Bundler is Still Vulnerable to Dependency Confusion Attacks (CVE-2020-36327)

29 April 2021 • zofrex

> **TL;DR:** Bundler was vulnerable to dependency confusion attacks if you had any implicit private dependencies, and was since version 1.16.0, released in October 2017, until version 2.2.18, released in May 2021.
>
> The latest version at the time of writing, **2.2.18**, fixes the issue.
>
> **Last revised: 25 May 2021.**

If you're using Bundler to manage your dependencies, you probably know that you shouldn't use multiple global sources, and should specify sources directly for private gems:

```
source "https://rubygems.org"

gem "rails", "~> 6.1" # public gems you depend on go here

source "https://private-packages.acme.example" do
    gem "acme-logger" # private gem
end
```

If you do this, the acme-logger gem will only ever be downloaded from `private-packages.acme.example`, and never from RubyGems, even if there is a package with the same name and higher version number on RubyGems.

Sounds fine, right?

But what if acme-logger has dependencies? And what if some of those dependencies are also private? For example, if acme-logger depends on another private gem, acme-util:

```
.
├── rails
└── acme-logger
    └── acme-util
```

The documentation has this to say:

> Bundler will search for child dependencies of this gem by first looking in the source selected for the parent, but if they are not found there, it will fall back on global sources using the ordering described in SOURCE PRIORITY.

and:

> Source Priority
>
> When attempting to locate a gem to satisfy a gem requirement, bundler uses the following priority order:
>
> - The source explicitly attached to the gem (using :source, :path, or :git)
> - For implicit gems (dependencies of explicit gems), any source, git, or path repository declared on the parent. This results in bundler prioritizing the ActiveSupport gem from the Rails git repository over ones from rubygems.org
> - The sources specified via global source lines, searching each source in your Gemfile from last added to first added.

According to the documentation, this is fine: it will search for acme-util in the same repository it got acme-logger from, and will only attempt to fetch it from the public repository if it doesn't find it.

However, the documentation is wrong.

The Vulnerability

Bundler will fetch implicit dependencies (dependencies of your explicit dependencies) from any declared source in the Gemfile, even if their parents are limited to a particular source. If the gem on RubyGems has a higher version number than the gem in your private repository, Bundler will fetch that one.

Looking back at our example Gemfile again:

```
source "https://rubygems.org"

gem "rails", "~> 6.1"

source "https://private-packages.acme.example" do
```

```
    gem "acme-logger"
end
```

If my private repository has acme-util version 1.0.0 in it, and someone uploads their own acme-util to RubyGems with version 1.0.1, this is what happens when I update:

```
% bundle update
Fetching gem metadata from https://private-packages.acme.example...
Fetching gem metadata from https://rubygems.org/.
Resolving dependencies...
Using bundler 2.2.16
Using rails 6.1.0
Fetching acme-util 1.0.1 (was 1.0.0)
Installing acme-util 1.0.1 (was 1.0.0)
Using acme-logger 0.1.0
Bundle updated!

% ruby main.rb
Hello from the public Gem that is masking the private Gem. Oh no!
```

Vulnerable versions

Affected: 1.7.0–2.2.16, except 2.2.10.

Nearly every version of Bundler released in the last three-and-a-half years is vulnerable, including 2.2.16, which is the latest version at the time of writing.

2.2.10 is a bit of a special case. It contains a fix rolled out for this issue, but the fix caused so many other problems it was reverted. I don't know if it's safe to use, generally. Maybe it's fine if none of the other bugs affect you?

Versions prior to 1.7 aren't applicable because they have no source scoping at all (which makes them even more vulnerable to dependency confusion).

It's worth noting that all 1.x versions are also vulnerable to CVE-2016-7954, although this is a slightly more niche issue, and is possible to mitigate.

Timeline

- August 13, 2014: Bundler 1.7.0 released, introducing non-global sources, fixing CVE-2013-0334
- October 31, 2017: Bundler 1.16.0 released, introducing this bug (CVE-2020-36327)
- September 30, 2020: First report of the bug (that I know of), although several reports have been made over the years that hint at its existence
- February 9, 2021: Alex Birsan publishes his post on dependency confusion, putting a spotlight on the issue
- February 15, 2021: Bundler 2.2.10 released, fixing the issue, and Bundler publish a blogpost saying the issue is fixed
- February 17, 2021: Bundler 2.2.11 released, rolling back the fix
- February 22, 2021: The Bundler blog post is updated to reflect that the fix has been rolled back (updating was delayed due to build failures of the Bundler website)
- April 29, 2021: CVE-2020-36327 is assigned

Mitigations

There are a few options for mitigating this vulnerability, but none of them are perfect.

Virtual Namespacing

The best mitigation, in my opinion, is to adopt the virtual namespacing pattern, which I wrote a post about last week. In short, this involves ensuring all your systems point at a mirror of RubyGems that you control, as well as your private Gem server. You then adopt a naming convention for your private gems (this is the "virtual namespace") and set policies on your repositories that ensure that the private gems will never be fetched from RubyGems.

This is the strongest mitigation, because it is a standalone fix for dependency confusion attacks, and it requires no ongoing work once implemented (unlike individually blocklisting your private gems on the RubyGems mirror).

It does have some significant downsides, however. If your existing gems don't already follow a naming pattern, you need to ensure you add all of them to the blocklist — missing a single one will leave you vulnerable — and if you aren't already using a RubyGems mirror, it will potentially require a lot of configuration changes in your environment and your apps.

Scope all gems (2.2.14+ Only)

You might be wondering if you can work-around this bug by having no global sources, which the documentation does recommend:

> Using the :source option for an individual gem will also make that source available as a possible global source for any other gems which do not specify explicit sources. Thus, when adding gems with explicit sources, it is recommended that you also ensure all other gems in the Gemfile are using explicit sources.

That would mean your Gemfile looks like this:

```
source "https://rubygems.org" do
    gem "rails", "~> 6.1"
end

source "https://private-packages.acme.example" do
    gem "acme-logger"
end
```

On Bundler version 2.2.13 and earlier, this doesn't change the results at all compared to above.

On versions 2.2.14, 2.2.15, and 2.2.16 this causes an error:

```
Fetching gem metadata from https://rubygems.org/.
Fetching gem metadata from https://private-packages.acme.example...
Resolving dependencies...
Could not find acme-util-1.0.1 in any of the sources
```

This isn't a great mitigation, in my opinion. The behaviour is out of spec, so it doesn't feel safe to rely on. There may be edge cases where it doesn't protect you from the vulnerability, or the behaviour may change in future versions of Bundler. As well as being unspecified behaviour, what Bundler does here is outright *weird*: it's aware that 1.0.1 exists because it found it in the public repository, but then only tries to download it from the private repository. Effectively, you'd be relying on a bug.

That said, I think it is worth making this change. It's the recommended way to use multiple repositories with Bundler, and writing your Gemfiles this way has mitigated other security issues in the past. I'm recommending doing this as a matter of best practice, however, and not as a mitigation for this specific issue.

It's also so easy to apply compared to the other mitigations here, you might consider doing it for extra protection while you work on implementing a more reliable mitigation.

Publicly register all your gems

To carry out the attack, someone needs to register gems with the same names as yours on the RubyGems repository. One way to prevent the attack from working is to do this yourself – register a dummy gem with the same name for every private gem you have.

The main downsides of this are:

- You need to make sure you register every Gem you have – if you miss a single one, you render the defence useless.
- You need to keep the list up to date as new Gems are created.
- This will publicly disclose the names of all your internal Gems, which may or may not be acceptable for you.

Explicitly provide source for each dependency

Bundler only gets confused about *implicit* dependencies, so you can make all your private dependencies *explicit*:

```
source "https://rubygems.org" do
    gem "rails", "~> 6.1"
end

source "https://private-packages.acme.example" do
    gem "acme-logger"
    gem "acme-util"
end
```

Note that acme-util is now declared in the Gemfile, when previously it was only installed due to being a dependency of acme-logger. Configured like this, Bundler will only fetch acme-util from the private repository.

This is a workable fix if you only have a very small number of implicit private dependencies and a small number of apps using them. If you have more than a few, it's not going to be tenable, because you have to ensure you include every implicit dependency in every Gemfile. You only have to miss one to be vulnerable. It's too much work to keep a handle on this, especially on an ongoing basis as dependencies are added or new apps created.

However, if you only have one or two implicit dependencies, and a small enough number of apps that you can keep an eye on all their Gemfiles, this might be enough to keep you safe until Bundler is fixed.

"What else can I do?"

Bundler is critical infrastructure for many people, and I am assuming that includes you if you are reading this, yet the team working on it lacks the resources they need to tackle issues like this in a timely manner.

I have worked with the Bundler team on this and other issues and they're a really great bunch. If you're unhappy that a critical bug has gone unfixed for such a long time, please, channel that energy towards financially supporting Ruby Together, the non-profit organisation that maintains Bundler and RubyGems.

Acknowledgements

Thank you to NeoThermic and Ben for their feedback on drafts of this post.

zofrex.com

James 'zofrex' Sanderson

Adventures in infosec, devops, and testing