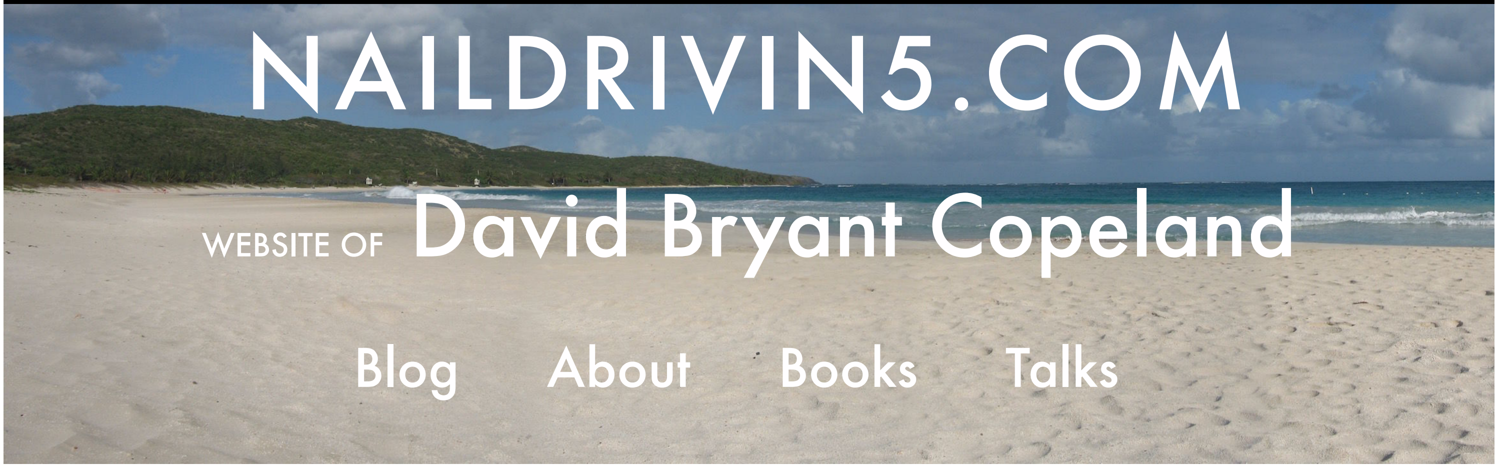
Black Lives Matter. See post



The Frightening State of Security Around NPM Package Management

JULY 10, 2019

Join my mailing list…

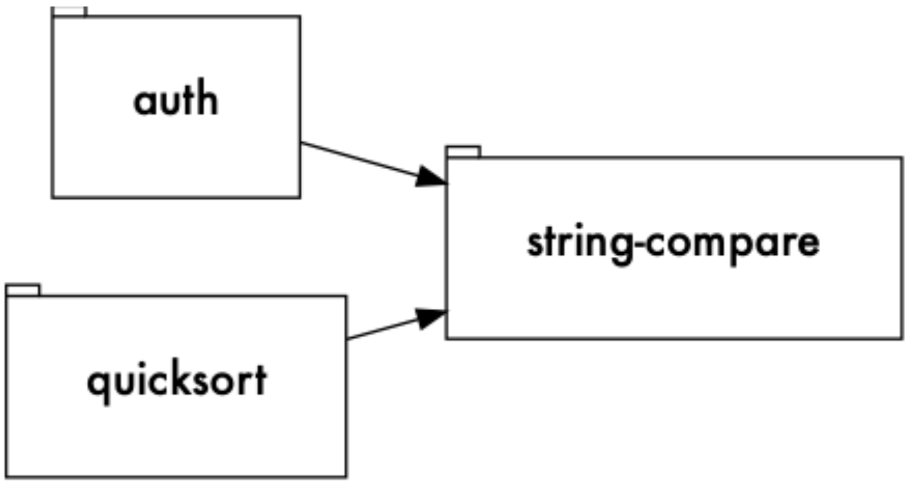
I take GitHub's new security vulnerability notifications seriously, and try to patch my apps whenever something comes up. I recently had trouble doing so for a JavaScript dependency, and uncovered just how utterly complex management of NPM modules is, and how difficult it must be to manage vulnerable packages. And I'm left wanting. I'm also left more concerned than ever that the excessive use of the NPM ecosystem is risky and dangerous.

The problem stems from three issues, each compounding the other:

- NPM's management of transitive dependencies that allows many versions of the same module to be active in one app.
- Core tooling lacking support to identify and remediate the inclusion if insecure modules.
- Common use of the same `package.json` for client and server side bundles.

How NPM Manages Transitive Dependencies

NPM manages transitive dependencies differently than other popular ecosystems. Suppose my application needs the hypothetical module `auth`. `auth` itself requires the module `string-compare`. Suppose further my application needs the module `quicksort`, which *also* requires `string-compare`:

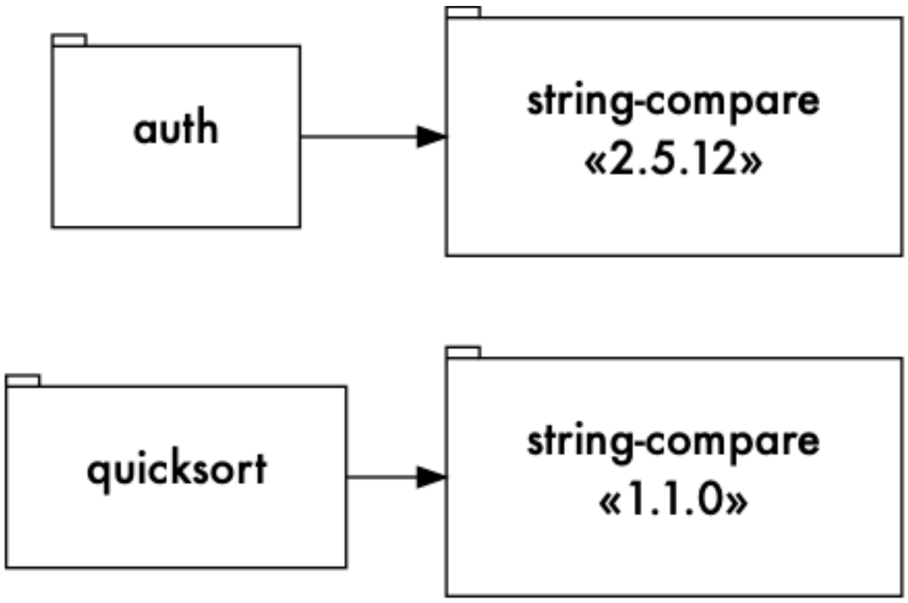


It's not this simple, as we usually specify the version of a dependency our application requires. Our dependencies themselves also specify the versions *they* require.

Suppose my app requires the latest versions of `auth` and `quicksort`, which are 4.3.0 and 2.1.1, respectively:

```
{
  "dependencies": {
    "auth": "^4.3.0",
    "quicksort": "^2.1.1"
  }
}
```

But, let's suppose that `auth` at 4.3.0 requires `string-compare` at 2.5.12, while `quicksort` requires 1.1.0, like so:



In a language like Ruby, Bundler would try to make sure we only had one version of any given package. It would try to find a set of dependencies that satisfied all packages, including transitively-included ones. If it couldn't, it would produce an error.

NPM instead allows *both* versions of `string-compare` to be present in the project. So, inside `auths` codebase, this code would load `string-compare` version 2.5.12:

```
const stringCompare = require("string-compare");
```

The same code inside `quicksort` would load version 1.1.0.

This can be viewed as a feature - since each NPM module is isolated and does not (normally) pollute the global namespace, you avoid thorny issues where an update to `string-compare` might break `quicksort`, or might force us to use a lower version of `auth` just so that both `auth` and `string-compare` can use the same version. Most Rubyists have experienced random downgrades when they `bundle update` because of this.

So what if we actually don't want that?

Security Vulnerabilities

Suppose that a security vulnerability is found in `string-compare`. *All* versions up to 2.5.12 have a serious exploit, and we need to make sure we are using 3.0.0 or greater. Suppose further that `auth` has released a new version depending on this, but `quicksort` has not.

If we `npm upgrade` (or `yarn upgrade`), we'll get version 4.3.1 of `auth`, which will then remove version 2.5.12 of string-compare, and bring in version 3.0.0. However, version 1.1.0 is still there, brought in my `quicksort`. Vulnerable code is still in our app!

You might think we could fix this by depending directly on `string-compare` so we can specify the version we need:

```
{
  "dependencies": {
    "auth": "^4.3.1",
    "quicksort": "^2.1.1",
    "string-compare": "^3.0.0"
  }
}
```

This won't work - the old version is still inside the `node_modules` of `quicksort`.

This seems to be a serious problem, because a developer might think they have fixed the problem (a cursory check in `node_modules/string-compare` would show the latest version), not realizing that the insecure version is still there, brought in my whatever module needed it. Nothing in the output of `yarn upgrade` or `npm ugprade` would indicate that you have an insecure version of a dependency still in your application, despite directly depending on the version with the fix (which makes sense, given that this behavior is viewed as a feature).

A developer might think they have fixed the problem, not realizing that the insecure version is still there

To catch this, you'd need to have configured an auditing system like the one GitHub uses and make sure it notified you before you considered the matter closed.

How, if you *did* notice, how would you fix this?

Remediation is Not So Easy

If the underlying vulnerability is serious enough, you might be able to rely on the owner of `quicksort` to see it and release an update. But consider the complexity of your dependencies when allowing all versions needed by all other versions in use. For example, suppose that we depend on the module `react-dropdown`, which requires the module `array-utils`, which requires the module `string-utils`, which requires version 1.3.0 of `string-compare`. In order to remove this new third vulnerable version, we'd need all of those modules to update. `string-utils` would need to release a new version, then `array-utils`, then `react-dropdown`.

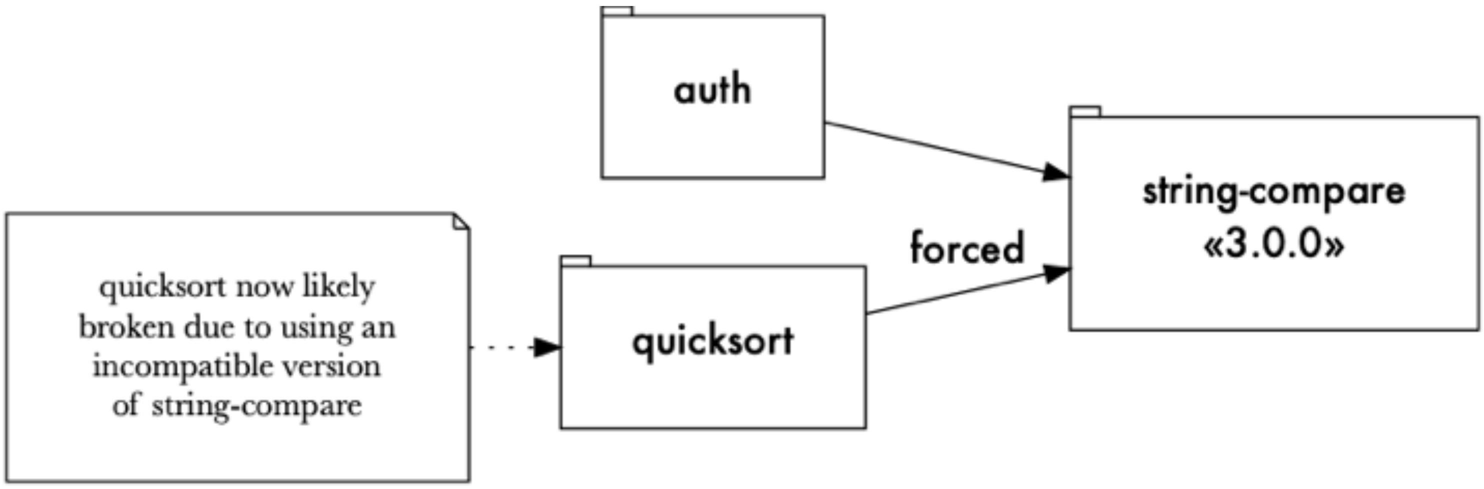
And now consider a tangle of many more dependency chains that include `string-compare`. Given that NPM modules tend to be small and single purpose, the web of dependencies in any application can be quite complex. Waiting for potentially hundreds of maintainers to coordinate so you can upgrade isn't a reliable strategy.

As of this writing, the `npm` CLI has no other solution. You'd have to somehow detect his is happening and figure out how to change your application to deal, or simply accept the vulnerability. Yarn has the concept of selective dependency resolutions. You can use this to tell Yarn to force a specific version on all other modules being brought in:

```
{
  "dependencies": {
    "auth": "^4.3.1",
    "quicksort": "^2.1.1"
  },
  "resolutions": {
    "string-compare": "^3.0.0"
  }
}
```

This will then prevent the download of multiple versions of `string-compare`. This solves the security vulnerability, but is this a good solution?

Unfortunately, this subverts the dependency requirements that module authors have stated. Suppose that `string-compare`'s 2.0.0 had a breaking change from the 1.x.x line. This would be reasonable, since we assume semantic versioning. Suppose further that `quicksort` depends on this now broken API. Because we have forced `quicksort` to use a package it is not capable of using, our application is now potentially broken.


auth
string-compare «3.0.0»
forced
quicksort
quicksort now likely broken due to using an incompatible version of string-compare

How would we know that? Hopefully our tests would reveal this, but it's not clear they would. Most test suites assume the proper functioning of the libraries the application depends on. The best case scenario would be a cryptic error message from inside one of these dependencies.

Of course, it's not unique to NPM that an application can depend on two modules that have incompatible transitive dependencies. The difference between NPM and, say, Bundler, is that Bundler will not let you even start up your application without a single, consistent set of dependencies.

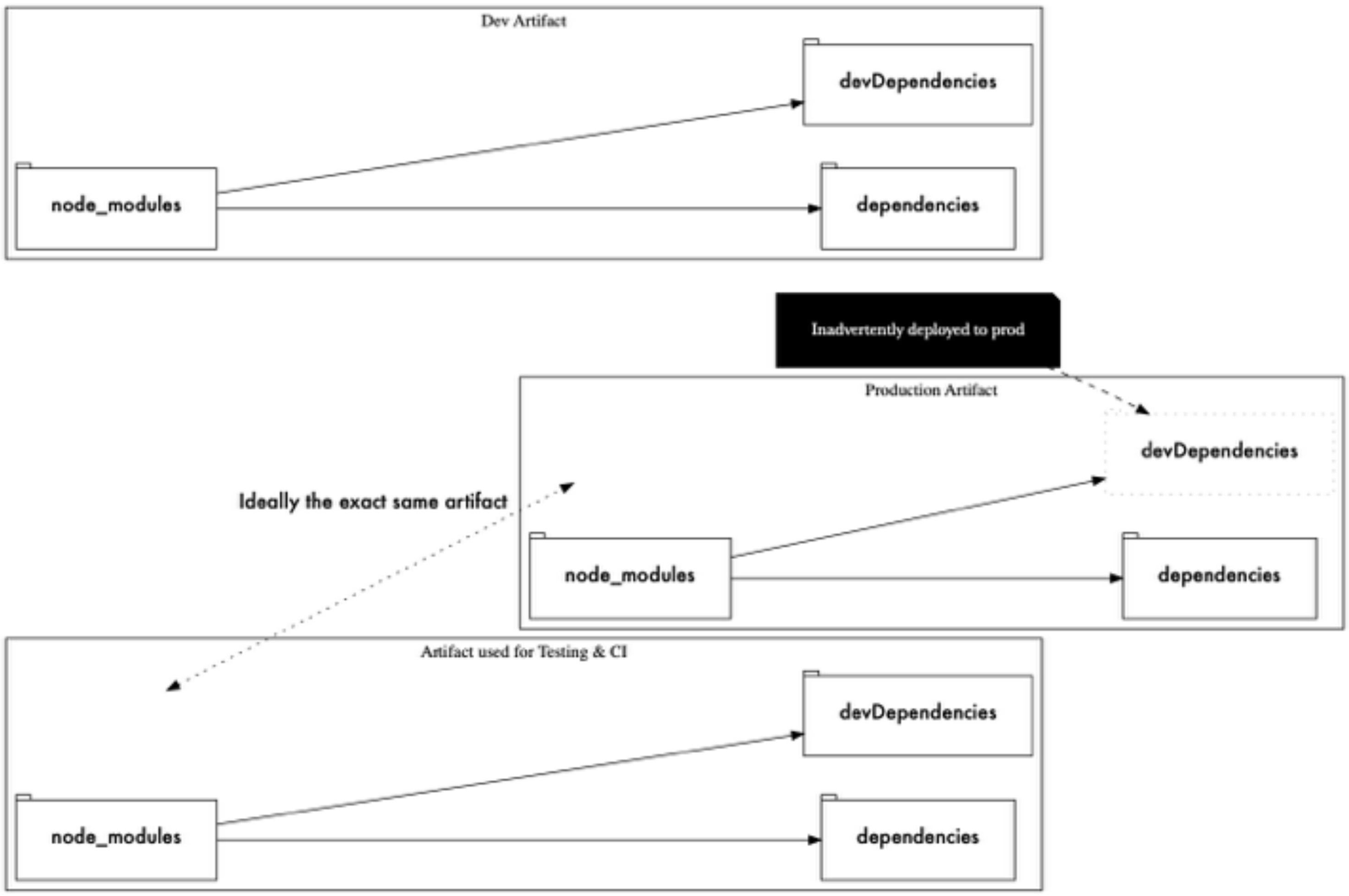
I think this difference is important.

If I'm on the application security team of a JavaScript-based application, I'm gonna be worried (though not about my continued employment :). I will have to manage a potentially significant amount of external tooling just to get his visibility, and navigate potentially difficult political situations to get teams to take more care when managing and upgrading their dependencies.

But, it gets worse.

Dev, Production, Client, Server

A *development* dependency is something you only need when building or testing the software. NPM allows specifying this via `devDependencies`. The assumption is that you run `npm install --only=prod` when building the final deployment. This is slightly problematic, because you must have dev dependencies installed in order to run tests, and many DevOps workflows favor deploying the exact artifact that was tested to ensure reproducibility, thus deploying dev dependencies to prod.



Of course, this problem isn't specific to NPM, but it's made more confusing by systems that have both client and server components and share a single `package.json`. You'd then use something like Webpack to create a client bundle. Webpack would theoretically understand exactly what is needed on the client and ship only that. The server bears no penalty for a large dependency tree, so it would use anything in `dependencies`.

The confusion here is in analyzing your application for the effect of a security vulnerability. Suppose one is found in a core package like `is-number`. Suppose the exploit is only relevant for a browser-based application.

You'd need to then understand if your client package includes `is-number` or not. With a shared `package.json`, it's very difficult to figure this out. You would need to produce your production bundle (e.g. with Webpack) and analyze that (which would be obfuscated and minified, making analysis potentially impossible).

Supposing that your client bundle contains insecure code, you now have the situation where perhaps you could upgrade your client-side code to safely use the patched version of `is-number`, but not the server side code because of the issues stated above.

Where Does This Leave Us?

These problems *can* certainly be fixed, but detecting them and remediating them is more expensive with NPM than with other language ecosystems, because NPM allows multiple versions of any given package along with the complexities around having multiple deploy targets (a client-side bundle and a server-side bundle).

Small teams, or teams without the benefit of security professionals, have to rely on tools and manual process to help understand their applications and fix problems. NPM does not provide the tools to help here, thus allowing developers to inadvertently ship insecure or broken software or both.

What this tells me is that the use of JavaScript comes with a higher security risk than other popular ecosystems, since the design of the tooling and systems it supports make analysis and remediation more difficult, while also allowing insecure configurations to be shipped to production.

How do we deal with risks? We mitigate them through tooling, process, and education.

Mitigating These Risks

NPM would be wise to ship a better-implemented version of Yarn's resolution feature. Ideally, it could be more focused around security, allowing a developer to specify actual requirements of certain modules to avoid security issues and for `npm` to then refuse to install a set of modules that would violate those requirements.

Aside from that, what you can do:

- Educate the team on how dependency resolution actually works.
- Reduce dependencies on NPM modules in general:
  - Product-wise, do you need a highly interactive UI? Do you need to model your view as a single-page app? You probably don't, especially when you consider the security tradeoffs and carrying costs. You have to educate your product managers on these issues.
  - Can you solve the problem by writing your own code? You probably can.
  - Is a module unmaintained? Perhaps you should inline it into your application, taking the parts you need and avoiding a dependency unlikely to be updated.
- Aggressively use `devDependencies`. I see `package.json` files with Webpack in `dependencies` and this makes no sense to me (I have done this myself many times, because it's all to easy to `yarn add webpack`).
- Do not conflate server-side code with client-side bundles. Make separate `package.json` files.
- Configure your client-side bundles to have production and development mirror each other in terms of what code is included. Or, run your client-side tests against the production bundle.
- Consider static analysis on all bundles to both understand what is in them and cross-reference with known security vulnerabilities.
- Carefully arrange your CI system so that you run your application with only its runtime dependencies when testing.

No ecosystem is free of issues like this, but the massive popularity and proliferation of JavaScript makes it a singularly-interesting ecosystem. And great care is needed. ❺

You Might Also Like

Creating a Culture of Consistency
MAY 06, 2018

Choosing Technology
AUGUST 08, 2019

Get occasional emails on being a senior engineer

Email:

Name:

Just a short list of interesting links as well as some exclusive content from me, all about being a well-rounded, trusted engineer who gets things done.

SUBSCRIBE

Typos or other issues? Feel free to tweet at me. To discuss the content, consider doing so on Hacker News, Reddit, Twitter, or lobste.rs