computers are bad

https://computer.rip - me@computer.rip - fax: +1 (505) 926-5492

2023-11-25 the curse of docker

I'm heading to Las Vegas for re:invent soon, perhaps the most boring type of industry extravaganza there could be. In that spirit, I thought I would write something quick and oddly professional: I'm going to complain about Docker.

Packaging software is one of those fundamental problems in system administration. It's so important, so influential on the way a system is used, that package managers are often the main identity of operating systems. Consider Windows: the operating system's most alarming defect in the eyes of many "Linux people" is its lack of package management, despite Microsoft's numerous attempts to introduce the concept. Well, perhaps more likely, because of the number of those attempts. And still, in the Linux world, distributions are differentiated primarily by their approach to managing software repositories. I don't just mean the difference between dpkg and rpm, but rather more fundamental decisions, like opinionated vs. upstream configuration and stable repositories vs. a rolling release. RHEL and Arch share the vast majority of their implementation and yet have very different vibes.

Linux distributions have, for the most part, consolidated on a certain philosophy of how software ought to be packaged, if not how often. One of the basic concepts shared by most Linux systems is centralization of dependencies. Libraries should be declared as dependencies, and the packages depended on should be installed in a common location for use of the linker. This can create a challenge: different pieces of software might depend on different versions of a library, which may not be compatible. This is the central challenge of maintaining a Linux distribution, in the classical sense: providing repositories of software versions that will all work correctly together. One of the advantages of stable distributions like RHEL is that they are very reliable in doing this; one of the disadvantages is that they achieve that goal by packaging new versions very infrequently.

Because of the need to provide mutually compatible versions of a huge range of software, and to ensure compliance with all kinds of other norms established by distributions (which may range from philosophical policies like free software to rules on the layout of configuration files), putting new software into Linux distributions can be... painful. For software maintainers, it means dealing with a bunch of distributions using a bunch of old versions with various specific build and configuration quirks. For distribution and package maintainers, it means bending all kinds of upstream software into compliance with distribution policy and figuring out version and dependency problems. It's all a lot of work, and while there are some norms, in practice it's sort of a wild scramble to do the work to make all this happen. Software developers that want their software to be widely used have to put up with distros. Distros that want software have to put up with software developers. Everyone gets mad.

Naturally there have been various attempts to ease these problems. Naturally they are indeed various and the community has not really consolidated on any one approach. In the desktop environment, Flatpak, Snap, and AppImage are all distressingly common ways of distributing software. The images or applications for these systems package the software complete with its dependencies, providing a complete self-contained environment that should work correctly on

any distribution. The fact that I have multiple times had to unpack flatpaks and modify them to fix dependencies reveals that this concept doesn't always work entirely as advertised, but to be fair that kind of situation usually crops up when the software has to interact with elements of the system that the runtime can't properly isolate them from. The video stack is a classic example, where errant OpenGL libraries in packages might have to be removed or replaced for them to function with your particular graphics driver.

Still, these systems work reasonably well, well enough that they continue to proliferate. They are greatly aided by the nature of the desktop applications for which they're used (Snapcraft's system ambitions notwithstanding). Desktop applications tend to interact mostly with the user and receive their configuration via their own interface. Limiting the interaction surface mostly to a GUI window is actually tremendously helpful in making sandboxing feasible, although it continues to show rough edges when interacting with the file system.

I will note that I'm barely mentioning sandboxing here because I'm just not discussing it at the moment. Sandboxing is useful for security and even stability purposes, but I'm looking at these tools primarily as a way of packaging software for distribution. Sandboxed software can be distributed by more conventional means as well, and a few crusty old packages show that it's not as modern of a concept as it's often made out to be.

Anyway, what I really wanted to complain a bit about is the realm of software intended to be run on servers. Here, there is a clear champion: Docker, and to a lesser degree the ecosystem of compatible tools like Podman. The release of Docker lead to a surprisingly rapid change in what are widely considered best practices for server operations. While Docker images a means of distributing software first seemed to appeal mostly to large scalable environments with container orchestration, it sort of merged together with ideas from Vagrant and others to become a common means of distributing software for developer and single-node use as well.

Today, Docker is the most widespread way that server-side software is distributed for Linux. I hate it.

This is not a criticism of containers in general. Containerization is a wonderful thing with many advantages, even if the advantages over lightweight VMs are perhaps not as great as commonly claimed. I'm not sure that Docker has saved me more hours than it's cost, but to be fair I work as a DevOps consultant and, as a general rule, people don't get me involved unless the current situation isn't working properly. Docker images that run correctly with minimal effort don't make for many billable hours.

What really irritates me these days is not really the use of Docker images in DevOps environments that are, to some extent, centrally planned and managed. The problem is the use of Docker as a lowest common denominator, or perhaps more accurately lowest common effort, approach to distributing software to end users. When I see open-source, server-side software offered to me as a Docker image or--even worse---Docker Compose stack, my gut reaction is irritation. These sorts of things usually take longer to get working than equivalent software distributed as a conventional Linux package or to be built from source.

But wait, how does that happen? Isn't Docker supposed to make everything completely self-contained? Let's consider the common problems, something that I will call my Taxonomy of Docker Gone Bad.

Configuration

One of the biggest problems with Docker-as-distribution is the lack of consistent conventions for configuration. The vast majority of server-side Linux software accepts its configuration through an ages-old technique of reading a text file. This certainly isn't perfect! But, it is pretty consistent in its general contours. Docker images, on the other hand...

If you subscribe to the principles of the 12-factor-app, the best way for a Docker image to take configuration is probably via environment variables. This has the upside that it's quite straightforward to provide them on the command line when starting the container. It has the downside that environment variables aren't great for conveying structured data, and you usually interact with them via shell scripts that have clumsy handling of long or complicated values. A lot of Docker images used in DevOps environments take their configuration from environment variables, but they tend to make it a lot more feasible by avoiding complex configuration (by assuming TLS will be terminated by "someone else" for example) or getting a lot of their configuration from a database or service on the network.

For most end-user software though, configuration is too complex or verbose to be comfortable in environment variables. So, often, they fall back to configuration files. You have to get the configuration file into the container's file system somehow, and Docker provides numerous ways of doing so. Documentation on different packages will vary on which way it recommends. There are frequently caveats around ownership and permissions.

Making things worse, a lot of Docker images try to make configuration less painful by providing some sort of entry-point shell script that generates the full configuration from some simpler document provided to the container. Of course this level of abstraction, often poorly documented or entirely undocumented in practice, serves mostly to make troubleshooting a lot more difficult. How many times have we all experienced the joy of software failing to start, referencing some configuration key that isn't in what we provided, leading us to have to find have the Docker image build materials and read the entrypoint script to figure out how it generates that value?

The situation with configuration entrypoint scripts becomes particularly acute when those scripts are opinionated, and opinionated is often a nice way of saying "unsuitable for any configuration other than the developer's." Probably at least a dozen times I have had to build my own version of a Docker image to replace or augment an entrypoint script that doesn't expose parameters that the underlying software accepts.

In the worst case, some Docker images provide no documentation at all, and you have to shell into them and poke around to figure out where the actual configuration file used by the running software is even located. Docker images must always provide at least some basic README information on how the packaged software is configured.

Filesystems

One of the advantages of Docker is sandboxing or isolation, which of course means that Docker runs into the same problem that all sandboxes do. Sandbox isolation concepts do not interact well with Linux file systems. You don't even have to get into UID behavior to have problems here, just a Docker Compose stack that uses named volumes can be enough to drive you to drink. Everyday operations tasks like backups, to say nothing of troubleshooting, can get a lot more frustrating when you have to use a dummy container to interact with files in a named volume. The porcelain around named volumes has improved over time, but seemingly simple operations can still be weirdly inconsistent between Docker versions and, worse, other implementations like

Podman.

But then, of course, there's the UID thing. One of the great sins of Docker is having normalized running software as root. Yes, Docker provides a degree of isolation, but from a perspective of defense in depth running anything with user exposure as root continues to be a poor practice. Of course this is one thing that often leads me to have to rebuild containers provided by software projects, and a number of common Docker practices don't make it easy. It all gets much more complicated if you use hostmounts because of UID mapping, and slightly complex environments with Docker can turn into NFS-style puzzles around UID allocation. Mitigating this mess is one of the advantages to named volumes, of course, with the pain points they bring.

Non-portable Containers

The irony of Docker for distribution, though, and especially Docker Compose, is that there are a lot of common practices that negatively impact portability---ostensibly the main benefit of this approach. Doing anything non-default with networks in Docker Compose will often create stacks that don't work correctly on machines with complex network setups. Too many Docker Compose stacks like to assume that default, well-known ports are available for listeners. They enable features of the underlying software without giving you a way to disable them, and assume common values that might not work in your environment.

One of the most common frustrations, for me personally, is TLS. As I have already alluded to, I preach a general principle that Docker containers should not terminate TLS. Accepting TLS connections means having access to the private key material. Even if 90-day ephemeral TLS certificates and a general atmosphere of laziness have deteriorated our discipline in this regard, private key material should be closely guarded. It should be stored in only one place and accessible to only one principal. You don't even have to get into these types of lofty security concerns, though. TLS is also sort of complicated to configure.

A lot of people who self-host software will have some type of SNI or virtual hosting situation. There may be wildcard certificates for multiple subdomains involved. All of this is best handled at a single point or a small number of dedicated points. It is absolutely maddening to encounter Docker images built with the assumption that they will individually handle TLS. Even with TLS completely aside, I would probably never expose a Docker container with some application directly to the internet. There are too many advantages to having a reverse proxy in front of it. And yet there are Docker Compose stacks out there for end-user software that want to use ACME to issue their own certificate! Now you have to dig through documentation to figure out how to disable that behavior.

The Single-Purpose Computer

All of these complaints are most common with what I would call hobby-tier software. Two examples that pop into my mind are HomeAssistant and Nextcloud. I don't call these hobby-tier to impugn the software, but rather to describe the average user.

Unfortunately, the kind of hobbyist that deploys software has had their mind addled by the cheap high of the Raspberry Pi. I'm being hyperbolic here, but this really is a problem. It's absurd the number of "self-hosted" software packages that assume they will run on dedicated hardware. Having "pi" in the name of a software product is a big red flag in my mind, it immediately makes me think "they will not have documented how to run this on a shared device."

Call me old-fashioned, but I like my computers to perform more than one task, especially the ones that are running up my power bill 24/7.

HomeAssistant is probably the biggest offender here, because I run it in Docker on a machine with several other applications. It actively resists this, popping up an "unsupported software detected" maintenance notification after every update. Can you imagine if Postfix whined in its logs if it detected that it had neighbors?

Recently I decided to give NextCloud a try. This was long enough ago that the details elude me, but I think I burned around two hours trying to get the all-in-one Docker image to work in my environment. Finally I decided to give up and install it manually, to discover it was a plain old PHP application of the type I was regularly setting up in 2007. Is this a problem with kids these days? Do they not know how to fill in the config.php?

Hiding Sins

Of course, you will say, none of these problems would be widespread of people just made good Docker images. And yes, that is completely true! Perhaps one of the problems with Docker is that it's too easy to use. Creating an RPM or Debian package involves a certain barrier to entry, and it takes a whole lot of activation energy for even me to want to get rpmbuild going (advice: just use copr and rpkg). At the core of my complaints is the fact that distributing an application only as a Docker image is often evidence of a relatively immature project, or at least one without anyone who specializes in distribution. You have to expect a certain amount of friction in getting these sorts of things to work in a nonstandard environment.

It is a palpable irony, though, that Docker was once heralded as the ultimate solution to "works for me" and yet seems to just lead to the same situation existing at a higher level of configuration.

Last Thoughts

This is of course mostly my opinion and I'm sure you'll disagree on something, like my strong conviction that Docker Compose was one of the bigger mistakes of our era. Fifteen years ago I might have written a nearly identical article about all the problems I run into with RPMs created by small projects, but what surprises me about Docker is that it seems like projects can get to a large size, with substantial corporate backing, and still distribute in the form of a decidedly amateurish Docker Compose stack. Some of it is probably the lack of distribution engineering personnel on a lot of these projects, since Docker is "simple." Some of it is just the changing landscape of this class of software, with cheap single-board computers making Docker stacks just a little less specialized than a VM appliance image more palatable than they used to be. But some if it is also that I'm getting older and thus more cantankerous.