IEEE Spectrum

OPINION COMPUTING

Why Bloat Is Still Software's Biggest Vulnerability › A 2024 plea for lean software

BY BERT HUBERT

08 FEB 2024

DANIEL ZENDER

THIS POST IS DEDICATED TO THE MEMORY OF Niklaus Wirth, a computing pioneer who passed away 1 January 2024. In 1995 he wrote an influential article called "A Plea for Lean Software," published in Computer*, the magazine for members of the IEEE Computer Society, which I read early in my career as an entrepreneur and software developer. In what follows, I try to make the same case nearly 30 years later, updated for today's computing horrors. A version of this post was originally published on my personal blog, Berthub.eu.*

Some years ago I did a talk at a local university on cybersecurity, titled "Cyber and Information Security: Have We All Gone Mad?" It is still worth reading today since we *have* gone quite mad collectively.

The way we build and ship software these days is mostly ridiculous, leading to apps using millions of lines of code to open a garage door, and other simple programs importing 1,600 external code libraries—dependencies—of unknown provenance. Software security is dire, which is a function both of the quality of the code and the sheer amount of it. Many of us programmers know the current situation is untenable. Many programmers (and their management) sadly haven't ever experienced anything else. And for the rest of us, we rarely get the time to do a better job.

It is not just you; we are not merely suffering from nostalgia: Software really is very weird today

Let me briefly go over the terrible state of software security, and then spend some time on why it is so bad. I also mention some regulatory and legislative things going on that we might use to make software quality a priority again. Finally, I talk about an actual useful piece of software I wrote as a proof of concept that one can still make minimal and simple yet modern software.

I hope that this post provides some mental and moral support for suffering programmers and technologists who want to improve things. It is not just you; We are not merely suffering from nostalgia: Software really is very weird today.

The terrible state of software security

Without going all "Old man (48) yells at cloud," let me restate some obvious things. The state of software security is *dire*. If we only look at the past year, if you ran industry-standard software like Ivanti, MOVEit, Outlook, Confluence, Barracuda Email Security Gateway, Citrix NetScaler ADC, and NetScaler Gateway, chances are you got hacked. Even companies with near-infinite resources (like Apple and Google) made trivial "worst practice" security mistakes that put their customers in danger. Yet we continue to rely on all these products.

Software is now (rightfully) considered so dangerous that we tell everyone not to run it themselves.

Software is now (rightfully) considered so dangerous that we tell everyone not to run it themselves. Instead, you are supposed to leave that to an "*X* as a service" provider, or perhaps just to "the cloud." Compare this to a hypothetical situation where cars are so likely to catch fire that the advice is not to drive a car yourself, but to leave that to professionals who are always accompanied by professional firefighters.

The assumption is then that the cloud is somehow able to make insecure software trustworthy. Yet in the past year, we've learned that Microsoft's email platform was thoroughly hacked, including classified government email. (Twice!) There are also well-founded worries about the security of the Azure cloud. Meanwhile, industry darling Okta, which provides cloud-based software that enables user log-in to various applications, got comprehensively owned. This was their second breach within two years. Also, there was a suspicious spate of Okta users subsequently getting hacked.

Clearly, we need better software.

The European Union has launched three pieces of legislation to this effect: NIS2 for important services; the Cyber Resilience Act for almost all commercial software and electronic devices; and a revamped Product Liability Directive that also extends to software revamped Product Liability Directive that also extends to software. Legislation is always hard, and it remains to be seen if they got it right. But that software security is terrible enough these days to warrant legislation seems obvious.

Why software security is so bad

I want to touch on incentives. The situation today is clearly working well for commercial operators. Making more secure software takes time and is a lot of work, and the current security incidents don't appear to be impacting the bottom line or stock prices. You can speed up time to market by cutting corners. So from an economic standpoint, what we see is entirely predictable. Legislation could be very important in changing this equation.

The security of software depends on two factors—the *density* of security issues in the source code and the sheer *amount of code* accessible by hackers. As the U.S. defense community loved to point out in the 1980s, quantity has a quality all of its own. The reverse applies to software—the more you have of it, the more risks you run.

As a case in point, Apple iPhone users got repeatedly hacked over many years because of the huge attack surface exposed by iMessage. It is possible to send an unsolicited iMessage to an Apple user. The phone will then immediately process that message so it can preview it. The problem is that Apple in its wisdom decided that such unsolicited messages needed to support a vast array of image formats, accidentally including PDFs with weird embedded compressed fonts using an ancient format that effectively included a programming language. So someone could send an unsolicited message to your iPhone that could probe for weaknesses in the rest of the phone.

In this way, attackers were able to benefit from security bugs in the phone's millions of lines of code. You don't need a high bug density to find an exploitable hole in millions of lines of code.

Wiping out all the bugs in your code won't save you from the decision to implement a feature to automatically execute code embedded in documents.

Apple could have prevented this situation by restricting previews to a far smaller range of image formats, or even a single "known good" image format. Apple could have saved themselves an enormous amount of pain simply by exposing fewer lines of their code to attackers. Incidentally, the E.U.'s Cyber Resilience Act explicitly tells vendors to minimize the attack surface.

Apple is (by far) not the worst offender in this field. But it is a widely respected and well-resourced company that usually thinks through what they do. And even they got it wrong by needlessly shipping and exposing too much code.

Could we not write better code?

There are those who think the biggest problem is the quality of the code, expressed in terms of the density of bugs in it. There are many interesting things happening on this front, like the use of memory safe languages like Rust. Other languages are also upping their security game. Fuzzers—test tools that automatically modify inputs to computer programs to find weaknesses and bugs—are also getting ever more advanced.

But many security problems are in the logic underlying the code. For example, the Barracuda email exploit originated in a third-party library that would actually execute code in Excel spreadsheets when they were scanned for viruses. Wiping out all the bugs in your code won't save you from the decision to implement a feature to automatically execute code embedded in documents.

The state of shipping software

Another problem is that we often don't know what code we are actually shipping. Software has gotten *huge*. In 1995 Niklaus Wirth lamented that software had grown to megabytes in size. In his article "A Plea for Lean Software," he went on to describe his Oberon operating system, which was only 200 kilobytes, including an editor and a compiler. There are now projects that have more than 200 KB for their configuration files alone.

A typical app today is built on Electron JS, a framework that incorporates both Chromium ("Chrome") and Node.JS, which provides access to tens of thousands of software packages for JavaScript. I estimate just using Electron JS entails at least 50 million lines of code if you include dependencies. Perhaps more. The app meanwhile likely pulls in hundreds or thousands of helper packages. Many packages used will also, by default, snitch on your users to advertisers and other data brokers. Dependencies pull in further dependencies, and exactly what gets included in the build can change on a daily basis, and no one really knows.

If this app controls anything in your house, it will also connect to a software stack over at Amazon, probably also powered by Node.js, also pulling in many dependencies.

We are likely looking at over 50 million active lines of code to open a garage door….

But wait, there's more. We used to ship software as the output of a compiler, or perhaps as a bunch of files to be interpreted. Such software then had to be *installed* and *configured* to work right. Getting your code packaged to ship like this is a lot of work. But it was good work since it forced people to think about what was in their "package." This software package would then integrate with an operating system and with local services, based on the configuration.

Since the software ran on a different computer than the one it was developed on, people really had to know what they shipped and think it through. And sometimes it didn't work, leading to the joke where a developer tells the operations people, "Well, it works on my system," and the retort "Then back up your email, we're taking your laptop into production!"

This used to be a joke, but these days we often ship software as containers, shipping not only the software itself but also including operating system files to make sure the software runs in a well-known environment. This frequently entails effectively shipping a complete computer disk image. This again vastly expands the amount of code being deployed. Note that you can do good things with containers like Docker (see below), but there are a lot of images over 350 MB on the Docker Hub.

Add it all up and we are likely looking at over 50 million active lines of code to open a garage door, running several operating-system images on multiple servers.

Now, even if all the included dependencies are golden, are we sure that their security updates are making it to your garage door opener app? I wonder how many Electron apps are still shipping with the image processing bug that had Google and Apple scramble to put out updates last year. We don't even know.

But even worse, it is a known fact that all these dependencies are *not* golden. The Node.js ecosystem has a comical history of package repositories being taken over, hijacked, or resurrected under the same name by someone else, someone with nefarious plans for your security. PyPI (a Python counterpart of Node.js) has suffered from similar problems. Dependencies always need scrutiny, but no one can reasonably be expected to check thousands of them frequently. But we prefer not to think about this. (Note that you should also not overshoot and needlessly reimplement everything yourself to prevent dependencies. There are very good modules that likely are more secure than what you could type in on your own.)

The world is shipping far too much code where we don't even know what we ship and we aren't looking hard enough (or at all) at what we *do* know we ship.

You *can* write lean code today

Writing has been called the process by which you find out you don't know what you are talking about. Actually doing stuff, meanwhile, is the process by which you find out you also did not know what you were writing about.

In a small reenactment of Wirth's Oberon Project, I too wrote some code to prove a point, and to reassure myself I still know what I am talking and writing about. Can you still make useful and modern software the old way? I decided to try to create a minimalistic but full-featured image-sharing solution that I could trust.

Trifecta is the result. It is actual stand-alone software that lets you use a browser to drag and drop images for easy sharing. It has pained me for years that I had to use imgur for this purpose. Not only does imgur install lots of cookies and trackers in my browser, I also force imgur install lots of cookies and trackers in my browser, I also force these trackers onto the people who view the images that I share. If you want to self-host a Web service like this, you also don't want to get hacked. Most image-sharing solutions I found that you could run yourself are based on huge frameworks that I don't trust too much for the reasons outlined above.

So, also to make a point, I decided to create a minimalistic but also useful image-sharing solution that I could trust. And more important, that other people could trust as well, because you can check out all Trifecta's code within a few hours. It consists of 1,600 lines of new source code, plus around five important dependencies.

You end up with a grand total of 3 megabytes of code.

To contrast, one other image-sharing solution ships as a 288-MB Docker image, although admittedly it looks better and has some more features. But not 285 MB worth of them. Another comparison is this Node-based picture-sharing solution, which clocks in at 1,600 dependencies, apparently totaling over 4 million lines of JavaScript.

The world ships too much code, most of it by third parties, sometimes unintended, most of it uninspected.

Note that Trifecta is not intended as a public site where random people can share images, as that does not tend to end well. It is however very suitable for company or personal use. You can read more about the project here, and there is also a page about the technology used to deliver such a tiny self-contained solution.

Response to Trifecta

This has been rather interesting. The most common response to Trifecta so far has been that I should use a whole bag of Amazon Web Services to deploy it. This is an exceedingly odd response to a project with the clearly stated goal of providing stand-alone software that does not rely on external services. I'm not sure what is going on here.

Another reaction has been that I treat Docker unfairly, and that you could definitely use containers for good. And I agree wholeheartedly. But I also look at what people are actually doing (also with other forms of containers or virtual machines), and it's not so great.

I want to end this post with some observations from Niklaus Wirth's 1995 paper:

> *"To some, complexity equals power. (…) Increasingly, people seem to misinterpret complexity as sophistication, which is baffling—the incomprehensible should cause suspicion rather than admiration."*

I've similarly observed that some people prefer complicated systems. As Tony Hoare noted long ago, "[T]here are two methods in software design. One is to make the program so simple, there are obviously no errors. The other is to make it so complicated, there are no obvious errors.” If you can’t do the first variant, the second way starts looking awfully attractive perhaps.

Back to Wirth:

> *“Time pressure is probably the foremost reason behind the emergence of bulky software. The time pressure that designers endure discourages careful planning. It also discourages improving acceptable solutions; instead, it encourages quickly conceived software additions and corrections. Time pressure gradually corrupts an engineer’s standard of quality and perfection. It has a detrimental effect on people as well as products.”*

Why spend weeks paring down your software when you can also ship a whole pre-installed operating-system image that just works?

> *“The plague of software explosion is not a ‘law of nature.’ It is avoidable, and it is the software engineer’s task to curtail it.”*

If this is indeed on the shoulders of software people, we should perhaps demand more time for it.

The world ships too much code, most of it by third parties, sometimes unintended, most of it uninspected. Because of this, there is a huge *attack surface* full of mediocre code. Efforts are ongoing to improve the quality of code itself, but many exploits are due to logic fails, and less progress has been made scanning for those.

Meanwhile, great strides could be made by paring down just how much code we expose to the world. This will increase time to market for products, but legislation is around the corner that should force vendors to take security more seriously.

Trifecta is, like Wirth’s Oberon Project mentioned above, meant as a proof that you can deliver a lot of functionality even with a limited amount of code and dependencies. With effort and legislation, maybe the future could again bring sub-50-million-line garage-door openers. Let’s try to make it happen.