Fifty Years of Open Source Software Supply Chain Security

FOR DECADES, SOFTWARE REUSE WAS ONLY A LOFTY GOAL. NOW IT'S VERY REAL.

RUSS COX

In March 1972, the United States Air Force started a review of a Honeywell Multics system to understand whether it could be used in secure environments. The report was issued in mid-1974 and concluded that Multics, while not secure, was better than its peers and might be a reasonable starting point for a secure system.[23] The report raised the potential of adding a backdoor (it was called a "trap door") to an innocent system call. When passed a specific, very unlikely input, the system call allowed reading or writing an arbitrary word of kernel memory. That tiny change would completely undermine the security of the system, and the report investigated the mechanics of how such a change might be made and hidden.

In March 2024, Andres Freund, a Postgres developer working at Microsoft, noticed that his Debian Linux system's ssh daemon was taking more CPU than normal to handle the Internet's usual background attack traffic that was trying to brute force logins into his machine. Upon closer investigation, Freund discovered that the latest version of liblzma, a compression library linked into ssh on

The contours of the problems in software supply chain security have not changed in half a century because they are fundamental.

Debian systems, contained a backdoor that targeted ssh specifically.[19] Now, when passed a specific, very unlikely input, the ssh daemon would allow an attacker on the Internet to execute an arbitrary shell command. This tiny change completely undermined the security of bleeding-edge Debian systems, and for the next few weeks, security researchers all over the world investigated the mechanics of how the change was made and hidden. Because liblzma was distributed as part of the xz project, this attack is now widely known as the xz attack.

The contours of the problems in software supply chain security have not changed in half a century because they are fundamental. There are no easy answers in computer security; software supply chain security is no exception. The best we can aim to do is keep improving our defenses, and many promising reinforcements are not yet universally deployed. This article aims to highlight promising approaches that should be more widely used as well as point out areas where more work is needed.

I led the development of the Go programming language and environment[15] for more than a decade, and software supply chain security is one of the specific focuses of that effort. This article is informed by that work and draws some examples from my personal experience, in addition to examples drawn from across the software industry.

EXPLORING THE PROBLEM

Open source software supply chain security is a hot topic, especially after the xz attack, but what exactly does it mean? In the absence of an agreed-upon definition, I suggest this one, in three parts:

1. An open source software supply chain *attack* is the insertion of nefarious open source code into trusted software before delivery. (This definition adapts one by Kim Zetter.[35])
2. An open source software supply chain *vulnerability* is an exploitable weakness in trusted software caused by a third-party, open source component of that software.
3. Open source software supply chain *security* is the engineering of defenses against open source software supply chain attacks and vulnerabilities.

There are a few important nuances to this definition.

The first is that the hardware supply chain is not of concern here. For example, *Der Spiegel* reported in 2013 that the U.S. NSA (National Security Agency) can intercept a new computer ordered by a target and install backdoored software or hardware components into it.[1] That kind of physical attack is beyond the scope of this discussion of the software supply chain, although it may still be important to consider and defend against in certain contexts.

A second nuance is that closed source software components are not of concern. For example, in 2012, attackers broke into Juniper Networks and changed the VPN (virtual private network) source code, replacing some critical constants in a random number generator. The effect was to create a backdoor that made it possible for the attackers to decrypt all VPN traffic that Juniper's customers sent through those devices.[8] This is a software supply chain attack because it changed the software before its delivery to Juniper's customers. But it is not an open source software supply chain attack, because the nefarious change was not made in one of the open source components used in Jupiter's VPN. (The random-number generator was easily backdoored in the first place thanks to the NSA mounting what might be considered an algorithmic supply chain attack.[6])

As another example, developers in China often look for copies of Xcode on file-sharing sites hosted in China, which are faster to download. In 2015, security researchers discovered that attackers had posted a modified copy of Xcode and worked to make it the top Chinese-language search result for "Xcode download." This version, which researchers named XcodeGhost, had been nefariously altered to add malicious code to every iOS app it built. It was downloaded and used by many app developers, and the injected malware made it into at least two widely used apps.[34] This is a software supply chain attack on the distribution mechanism rather than the original software, but, once again, it is not targeting open source software.

A third nuance is that the affected software does not itself need to be open source. For example, in 2021 security researchers discovered that the open source Java logging library Log4j would download and execute Java code from arbitrary URLs when logging text with a certain format.[33] Log4j is widely used in the Java ecosystem. As one example among millions of affected programs, in the popular game Minecraft, simply sending a chat message was sufficient to achieve remote code execution on the game server. Minecraft is, therefore, an example of a closed source program affected by an open source software supply chain vulnerability. Since almost all closed source programs use open source components,[2] they all depend on good open source software supply chain security.

A final nuance is that attacks that involve code written with nefarious intent are distinguished from vulnerabilities that involve innocent bugs. For example, in 2021 Apple fixed a bug that allowed so-called zero-click takeovers of an iPhone device by sending an iMessage with a specially crafted image attachment. The attachment identified itself as a GIF but was actually a PDF containing a JBIG2 image. Apple's software used the open source Xpdf JBIG2 decoder, written in C, and that decoder did not properly validate the encoded Huffman trees in the image; this made it possible to trigger bitwise operations on memory at attacker-controlled offsets beyond an allocated region. The attackers implemented an entire virtual CPU out of these bitwise operations and then implemented code in that virtual instruction set to scan process memory, break out of the iMessage sandbox, and take over the phone.[4] The JBIG2 bug was only accidentally (not nefariously) introduced, so it is an open source software supply chain vulnerability, not an attack. Vulnerabilities and attacks are distinct problems with distinct potential solutions.

As another example, in 2018 researchers discovered that the npm package event-stream contained obfuscated code that harvested bitcoin wallets when linked into the Copay mobile app.[21] The nefarious code explicitly targeted Copay, making that app an example of a closed source program affected by an open source software supply chain attack.

Although the last two examples ultimately impacted closed source applications, attacks on purely open source software stacks are also possible. The xz attack did exactly this, attacking OpenSSH through a component of its software supply chain—liblzma—instead of a direct attack on the OpenSSH source code or project itself. Even a purely open source attack can be devastating: If a few more months had passed before its discovery, the backdoored sshd (Secure Shell daemon) would have been deployed in sensitive contexts worldwide.

Although there is no silver bullet, the remainder of this article highlights the general themes for engineering better defenses as well as practical steps being taken today.

The software supply chain is all the places where a software supply chain attack might happen or vulnerability might be introduced.

UNDERSTAND THE SOFTWARE SUPPLY CHAIN

To secure your software supply chain, you first have to understand what it is. Let's start with the definition: The *software supply chain* is all the places where a software supply chain attack might happen or vulnerability might be introduced. The more important meaning of *understand*, however, is knowing what your specific software supply chain looks like, and that turns out to be quite difficult. The word *chain* sounds simple, but a supply chain is like a fractal: complex no matter how closely you look at it.

At the lowest level, you can look at the commands executed to build a single program and the dependencies between those commands. These build graphs align with the package dependency structure of the program itself. Even for simple programs, these graphs are so complex as to be unprintable. The Go project makes it a priority to avoid unnecessary dependencies and keep software simple,[15] and yet, as I write this article, building the go command executes 714 commands to build 297 packages, with 3,132 dependency edges in its package graph. The go command is unusual in that it has no external dependencies: All the packages it uses are part of the Go project itself. Looking at a slightly more complex command, Kubernetes's kubelet executes 3,289 commands in its build and depends on 1,581 packages from 137 Go modules, including many from outside the Kubernetes project. Both these examples are fairly small, low-level utilities. Higher-level programs have even more complex builds. In another article in this issue of *acmqueue*, Josie Anugerah and Eve Martin-Jones examine the complexity of open source build graphs in more detail, along with the surprise that most programs have many possible build graphs, depending on the exact context in which they are built.

It is tempting to think these kinds of dependency graphs are the entire software supply chain, but they are only the most visible part. Each package or module in the dependency graph may be written by a different person or organization, with different security practices, code review standards, and so on. It would be helpful to know more about these details for every dependency you take on, but in general, this information is unavailable and can change over time.

Another kind of graph shows the computers and services that software passes through during a build and during distribution to users. A nefarious alteration could be made in any one of these computers or services, making each one a different potential attack site, not to mention a potential source of vulnerabilities. You should be concerned with who has access to each dependency project, who might have access in the future, what infrastructure they use, and so on. For the most part, without visibility into any of that, it goes ignored. But it's all still there.

Understanding the software supply chain is critical to identifying which links need to be reinforced. As an industry, we have much more work to do here, but for this article, let's move on to specific reinforcements already known to help.

AUTHENTICATE SOFTWARE

The Multics review contemplated inserting backdoors "during the distribution phase," taking advantage of "insecure telecommunications" as well as sending out nefarious updates "using forged stationery." The wording may be outdated, but the ideas are not. XcodeGhost is a modern example of exactly that approach. Fixing this problem is the nearest thing to a true success story in modern software supply chain security. Cryptographic signatures make it impossible to nefariously alter code between signing and verifying. The only problem left is key distribution: The verifier has to know who should have signed the code.

There are many possible answers to the key distribution problem. The simplest one is to ignore questions of identity and simply record and distribute the expected cryptographic hash of the specific dependency versions being used in a given build or package manager. Verification of these predistributed hashes completely removes download servers, proxies, and other network middleboxes as potential attack sites. Debian's system for packaging dependencies includes such a check, which meant the xz attacker could not simply modify an existing copy of xz; they needed to release a new version. That didn't prevent the attack, but it did make the attack harder.

On a larger scale, instead of predistributing all these hashes, they can be maintained in a trusted database. The Go checksum database[16] is a real-world example of this approach that protects millions of Go developers. The database holds the SHA256 checksum of every version of every public Go module. Each database entry is signed by the database server's private key. The corresponding public key is hard-coded in the go command source code, so key distribution piggybacks on the rest of the Go distribution.

Every time the go command downloads a new open source Go package, it looks up the expected checksum. There is a local checksum cache for dependencies in a given project, so network calls to the checksum server happen only for upgrades or adding new dependencies, but one way or another, every download is checked. This means all the proxies and other boxes between code hosting and a user's computer cannot be attack sites. Even an attack on the code hosting site cannot change old packages.

There is, of course, the question of what checksum to put into the database. For Go, if the database hasn't recorded a specific package version yet, it fetches the code directly and stores the checksum for the code it gets. This "trust on first use" approach doesn't mean the code is trustworthy, but it does mean that the code can't change if anyone else downloads it on a different computer tomorrow. This immutability ensures that the entire Go ecosystem agrees on the meaning of Kubernetes version

1.28.4, which serves as a foundation for any other analysis work.

To the extent that questions of identity can be solved, having authors sign their software can provide even stronger guarantees. In the case of xz, distribution packages were signed using the individual author's GPG (Gnu Privacy Guard) keys, making it possible to distinguish the packages signed by the original (trustworthy) maintainer of xz and the ones signed by the attacker who had taken control of the project.

MAKE BUILDS REPRODUCIBLE

The Multics review noted that a nefarious change might "best be hidden in changes to the binary code of a compiled routine," leaving the corresponding source code unmodified. Such a change would persist only until a rebuild from source code, but most installations would not rebuild source code without reason. This remains a concern today. For example, the critical lines of code that triggered the xz attack during its build were included only in the packaged distribution, not in the actual source control repository.[14]

The best and most obvious way to verify that binaries have not been modified is to rebuild them and check the result against the distributed binaries, but that assumes builds are reproducible. Since computers are deterministic, it sounds like this should be trivial, but it is far too easy for contextual information like the build machine architecture or machine name, the names of temporary directories, or the current time to end up in some build output, making the overall build nonreproducible. The Reproducible Builds[31]

Fifty years ago, there was some hope that software could be made completely secure by correct design and careful implementation. We know better now.

project aims to raise awareness of reproducible builds generally, as well as building tools to help make progress toward complete reproducibility for all Linux software.

The Go project recently arranged for Go itself to be completely reproducible given only the source code, meaning that although a build needs some computer running some operating system and some earlier Go toolchain, none of those choices matters. A build for a given target produces the same distribution bits whether you build on Linux or Windows or Mac, whether the build host is X86 or ARM, and so on. Strong reproducibility makes it possible for others to easily verify that the binaries that are posted for download match the source code. Those binaries are also logged in the Go checksum database and verified when the go command downloads a new toolchain, so that downloads can't be modified in transit.[10]

Authenticating software and making builds reproducible remove potential attack vectors, although certainly not all. Let's turn our focus now to vulnerabilities.

FIND AND FIX VULNERABILITIES QUICKLY

Fifty years ago, there was some hope that software could be made completely secure by correct design and careful implementation. We know better now. Accepting that software will always have vulnerabilities, we must be ready to find and fix those vulnerabilities as quickly as possible when they arise.

Since attackers are looking for those vulnerabilities too, the best defense is to find and fix them first. The most trivial example is an out-of-date dependency with a known vulnerability. There are many available vulnerability

It is important to scan your software regularly, ideally daily, because even if your software is not changing, new entries are always being added to the database.

scanning tools that can identify this situation, whether language-specific tools like govulncheck and npm audit, general open source tools like osv-scanner,[30] or commercial tools. All of them work by cross-checking a list of the software inputs to a build—the *software bill of materials*—against a database of known vulnerabilities.

The specific choice of tool or database is not as important as it used to be. The open source software community has standardized on the OSV (Open Source Vulnerabilities) format[7] for individual vulnerability descriptions, including a precise, algorithmic description of the affected packages and versions. The OSV database[29] then aggregates all the language-specific databases. The CVE (Common Vulnerabilities and Exposures) database's JSON 5.0 schema also adopted OSV's precise information about affected packages and versions, enabling interchange between OSV and CVE. It benefits everyone for all tools to have access to the same, complete information about known vulnerabilities.

It is important to scan your software regularly, ideally daily, because even if your software is not changing, new entries are always being added to the database. And then you need to be ready to update to a fixed version of that dependency. This requires having comprehensive testing to make sure that the fixed version does not introduce any new bugs, as well as having automated deployment, so that a patched version of your software can go out in hours or days, not weeks or months.

Testing and deployment are standard software engineering concerns, not specifically about supply chain security, but without them, your security posture suffers, as can your legal exposure. When the Log4j vulnerability was discovered in 2021, it took most companies weeks or months (or more) to inventory all their software to determine what was affected and then update and redeploy that software. Even the U.S. FTC (Federal Trade Commission) issued a statement warning companies to update Log4j to "reduce the likelihood of harm to consumers, and to avoid FTC legal action,"[18] pointing to Equifax's previous liability for an intrusion enabled by unpatched software.[17]

Scanning for known vulnerabilities is the bare minimum. Effort should ideally also be spent looking for as-yet-unknown vulnerabilities in your open source dependencies. When you run security audits of your own source code, it is often worthwhile to identify critical open source dependencies and audit them too. Running bug-finding analysis tools or fuzzers on your software and its dependencies can also be effective. Attackers are going to use all these methods; you might as well use them first.

PREVENT VULNERABILITIES

Even if software will always have vulnerabilities, there are steps you can take to prevent certain kinds or to make them less likely.

To start, omit needless dependencies. Gordon Bell once observed that "[t]he cheapest, fastest and most reliable components of a computer system are those that aren't there."[5] The most secure software dependencies are the ones not used in the first place: Every dependency adds risk.

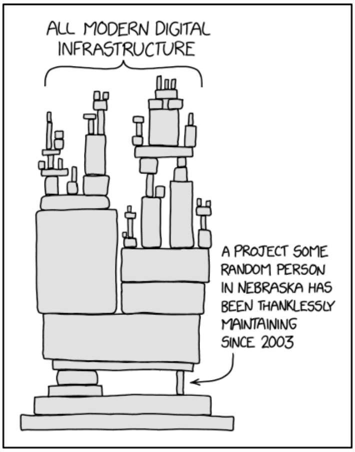
The OpenSSH project is careful about not taking on unnecessary dependencies, but Debian was not as careful. That distribution patched sshd to link against libsystemd, which in turn linked against a variety of compression packages, including xz's liblzma. Debian's relaxing of sshd's dependency posture was a key enabler for the attack, as well as the reason its impact was limited to Debian-based systems such as Debian, Ubuntu, and Fedora, avoiding other distributions such as Arch, Gentoo, and NixOS. Weeks before the xz attack was deployed, system developers had been discussing removal of the dependency on compressors such as liblzma, specifically to improve security. It is pure speculation, but those discussions may have accelerated the timeline for launching the attack.[3]

The same lesson applies to all projects, large and small. If it is possible to get by without a dependency, that's usually best. If not, small dependencies are better than large ones, and the number of transitive dependencies matters. Look not only at the one dependency being added but also at its impact on the overall dependency graph, using tools like Open Source Insights.[27]

Another good way to prevent vulnerabilities is to use safer programming languages that remove error-prone language features or make them needed less often. In 2022, the NSA released a recommendation on "Software Memory Safety" encouraging the use of memory-safe languages such as C#, Go, Java, or Rust instead of C and C++.[26] Among the many strikes against C and C++, manual memory management and the lack of any kind of bounds checking simply make programs too easy to get wrong in a way that creates a security vulnerability. Their reliance on "undefined behavior" adds another level of danger.[9] Of course, there is a substantial amount of C and C++ code in the world, and those programs cannot be abandoned overnight. For new efforts, however, adopting a safer language has significant security benefits.

FUND OPEN SOURCE

A famous xkcd comic from 2020 depicts "all modern digital infrastructure" built atop "a project some random person in Nebraska has been thanklessly maintaining since 2003."[25] The comic remains a disturbingly accurate assessment of the situation.


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

In 2014, researchers discovered that OpenSSL, a library widely used by Internet HTTPS servers, would respond to a specific kind of malformed packet by sending back arbitrary chunks of server memory. In some cases, that memory included the server's secret-key material. This was not an attack, but instead, an innocent coding mistake. (OpenSSL is written in C, so this mistake was incredibly easy to make and miss; in a memory-safe language with proper bounds checking, it would have been nearly impossible.)

The vulnerability was named Heartbleed and prompted a reckoning across the industry about exactly the situation described in the xkcd comic. At the time, OpenSSL was maintained by a handful of volunteers, with only one full-time developer. Researchers estimated that a security audit costing on the order of $100,000 would have caught the mistake, but the project received only $2,000 in annual donations, despite billions of dollars of commerce relying on the software each year. One outcome of this reckoning was the creation and funding of the Linux Foundation's Core Infrastructure Initiative, which evolved into the Open Source Security Foundation, or OpenSSF.[28]

The xz attack is the clearest possible demonstration that the problem is not fixed. It was enabled as much by underfunding of open source as by any technical detail.

OpenSSF is an important step forward, but it has not solved the problem of modern digital infrastructure depending on critical underfunded projects. More work is needed.

The xz attack is the clearest possible demonstration that the problem is not fixed. It was enabled as much by underfunding of open source as by any technical detail. Here's the story of the xz attack.

Lasse Collin started the xz project in 2005, using the LZMA compression algorithm, which compressed files to about 70 percent of what gzip did. Over time, this format became widely used for compressing tar files, Linux kernel images, and many other uses. For the most part, the software was stable and did not need significant ongoing attention. Collin was not paid for it, and it was not his full-time job.

In late 2021, an attacker using the (almost certainly not real) name "Jia Tan" started sending innocuous patches with small improvements to the xz development mailing list.[22] In mid-2022, the attacker started posting using other accounts and names on the mailing list, complaining about the slow pace of releases and new features, and pressuring Collin to cede control to someone with more time. They wrote things like, “The current maintainer lost interest or doesn’t care to maintain anymore. It is sad to see for a repo like this.” Also, “I get that this is a hobby project for all contributors, but the community desires more. Why not pass on maintainership…?”

The pressure campaign worked. Over the next year and a half, Collin turned over more and more development responsibility to the attacker, who built trust by making honest improvements and doing important maintenance work. In early 2023, the attacker built their first official xz release; as 2023 went on, they laid the technical groundwork for the actual attack, ultimately launched in early 2024.[13]

In the months after the attack was discovered, most speculation focused on the likelihood that the xz attack was carried out by nation-state hackers,[22] with no confirmation one way or the other. No matter who was responsible, it likely did not cost much. A competent software engineer working full-time on an open source project for two years to gain the trust of its maintainer probably costs less than a million dollars. The development of the exploit code itself, which was quite sophisticated, might cost another million dollars or so. A hidden backdoor into the vast majority of Linux ssh servers on the Internet would be worth many times more than that, possibly billions of dollars. The general underfunding of open source projects makes them directly susceptible to this kind of honest-seeming free help.

The social engineering of the xz attack is also not an isolated incident. In the event-stream attack mentioned earlier, the attacker simply asked whether the original author wanted someone to take over maintenance. In the aftermath of the xz attack, the OpenSSF and the OpenJS Foundation issued a warning about a similar campaign that had been unsuccessfully carried out against OpenJS.[20]

It is far from obvious how best to fund open source development. More than a decade after Heartbleed and the launch of the Core Infrastructure Initiative, the problem clearly remains unsolved.

We also need to find ways to fund open source development to make it less susceptible to takeover by the mere offer of free help.

CONCLUSION

We are all struggling with a massive shift that has happened in the past 10 or 20 years in the software industry. For decades, software reuse was only a lofty goal. Now it's very real.[12] Modern programming environments such as Go, Node, and Rust have made it trivial to reuse work by others, but our instincts about responsible behaviors have not yet adapted to this new reality.

The fact that the 1974 Multics review anticipated many of the problems we face today is evidence that these problems are fundamental and have no easy answers. We must work to make continuous improvements to open source software supply chain security, making attacks more and more difficult and expensive.

There are important steps we can take today, such as adopting software signatures in some form, making sure to scan for known vulnerabilities regularly, and being ready to update and redeploy software when critical new vulnerabilities are found. More and more development should be shifted to safer languages that make vulnerabilities and attacks less likely. We also need to find ways to fund open source development to make it less susceptible to takeover by the mere offer of free help. Relatively small investments in OpenSSL and xz development could have prevented both the Heartbleed vulnerability and the xz attack.

The xz attack seems to be the first major attack on the open source software supply chain. The event-stream attack was similar but not major, and Heartbleed and Log4j were vulnerabilities, not attacks. But the xz attack was discovered essentially by accident because it made sshd just a bit too slow at startup. Attacks, by their nature, try to remain hidden. What are the chances we would accidentally discover the very first major attack on the open source software supply chain in just a few weeks? Perhaps we were extremely lucky, or perhaps we have missed others.

The Multics review is famous for pointing out the possibility of adding a backdoor to a compiler to insert backdoors in critical system programs during compilation as XcodeGhost later did, and then the possibility of having the compiler backdoor *itself*, so that the modifications would persist even after a full recompilation of the compiler. Reading the report inspired Ken Thompson to implement exactly that attack on an early Unix system, probably in early 1975. He later explained the attack in his 1983 Turing Award lecture, published in *Communications of the ACM* as "Reflections on Trusting Trust."[32] Thompson kept the original attack source code, and with his permission, I published an annotated copy in 2023.[11] Perhaps the most unsettling part is how short it is: 99 lines of C and a 20-line shell script.

In his lecture, Thompson said, "The moral is obvious: You can't trust code that you did not totally create yourself." But today, we do that all the time, whether the trust is warranted or not. We use source code downloaded from strangers on the Internet in our most critical applications; almost no one is checking the code.

Lawrence Kesteloot's excellent short story "Coding Machines"[24] imagines a computing world under attack from Thompson's backdoor writ large. In our actual world, the sophistication of this kind of backdoor is simply not necessary. There are far easier ways to mount a supply chain attack, such as asking a maintainer if they would like some help. It would be nice to live in a world where attacks require the level of sophistication described by Thompson and Kesteloot.

We all have more work to do.

References

1. Appelbaum, J., et al. 2013. Documents reveal top NSA hacking unit. Spiegel International; https://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html.
2. Bals, F. 2024. 2024 open source security and risk analysis report. Blackduck blog; https://www.blackduck.com/blog/open source-trends-ossra-report.html.
3. Beaumont, K. 2024. Inside the failed attempt to backdoor SSH globally — that got caught by chance. DoublePulsar; https://doublepulsar.com/inside-the-failed-attempt-to-backdoor-ssh-globally-that-got-caught-by-chance-bbfe628fafdd.
4. Beer, I., Groß, S. 2021. A deep dive into an NSO zero-click iMessage exploit: Remote Code Execution, Google

Project Zero blog; https://googleprojectzero.blogspot.com/2021/12/a-deep-dive-into-nso-zero-click.html.

5. Bentley, J. 1985. Programming pearls. *Communications of the ACM* 28(9), 896–901; https://dl.acm.org/doi/10.1145/4284.315122.
6. Bernstein, D. J., Lange, T., Niderhagen, R. 2015. Dual EC: a standardized back door. *LNCS Essays on the New Codebreakers 9100*, ed. P. Y. A. Ryan, D. Naccache, J.-J. Quisquater, 256–281; https://eprint.iacr.org/2015/767.
7. Chang, O., Catlin, K. 2023. Getting to know the Open Source Vulnerability (OSV) format, OpenSSF blog; https://openssf.org/blog/2023/05/02/getting-to-know-the-open source-vulnerability-osv-format/.
8. Checkoway, S., et al. 2018. Where did I leave my keys?: lessons from the Juniper Dual EC incident. *Communications of the ACM* 61(11), 148–155; https://dl.acm.org/doi/10.1145/3266291.
9. Cox, R., 2023. C and C++ prioritize performance over correctness. research!rsc blog post; https://research.swtch.com/ub.
10. Cox, R. 2023. Perfectly reproducible, verified Go toolchains. The Go Blog; https://go.dev/blog/rebuild.
11. Cox, R., 2023. Running the "Reflections on Trusting Trust" compiler. research!rsc blog post; https://research.swtch.com/nih.
12. Cox, R., 2019. Surviving software dependencies. *Communications of the ACM 62(9)*, 36–43; https://dl.acm.org/doi/10.1145/3347446.
13. Cox, R. 2024. Timeline of the xz open source attack. research!rsc blog post. https://research.swtch.com/xz-timeline.

14. Cox, R., 2024. The xz attack shell script. research!rsc blog post; https://research.swtch.com/xz-script.
15. Cox, R., Griesemer, R., Pike, R., Taylor, I. L., Thompson, K. 2022. The Go programming language and environment. *Communications of the ACM 65(5),* 70–78; https://dl.acm.org/doi/10.1145/3488716.
16. Cox, R., Valsorda, F. 2019. Proposal: secure the public Go module ecosystem. Go design document; https://go.dev/design/25530-sumdb.
17. Federal Trade Commission. 2019. Equifax to pay $575 million as part of settlement with FTC, CFPB, and states related to 2017 data breach. FTC press release; https://www.ftc.gov/news-events/news/press-releases/2019/07/equifax-pay-575-million-part-settlement-ftc-cfpb-states-related-2017-data-breach.
18. Federal Trade Commission. 2022. FTC warns companies to remediate Log4j security vulnerability. FTC Office of Technology blog; https://www.ftc.gov/policy/advocacy-research/tech-at-ftc/2022/01/ftc-warns-companies-remediate-log4j-security-vulnerability.
19. Freund, A., 2024. Backdoor in upstream xz/liblzma leading to ssh server compromise. oss-security mailing list, Openwall; https://www.openwall.com/lists/oss-security/2024/03/29/4.
20. Ginn, R. B., Arasaratnam, O. 2024. XZ Utils cyberattack likely not an isolated incident. OpenSSF blog; https://openssf.org/blog/2024/04/15/open source-security-openssf-and-openjs-foundations-issue-alert-for-social-engineering-takeovers-of-open source-projects/.
21. Goodin, D., 2018. Widely used open source software contained bitcoin-stealing backdoor. *Ars*

Technica; https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open source-software-to-steal-bitcoin/.

22. Greenberg, A., Burgess, M. 2024. The Mystery of "Jia Tan," the XZ backdoor mastermind. *Wired;* https://www.wired.com/story/jia-tan-xz-backdoor/.
23. Karger, P. A., Schell, R. R. 1974. Multics security evaluation: vulnerability analysis. U.S. Air Force Electronic Systems Division report ESD-TR-74-193, Vol. II; https://seclab.cs.ucdavis.edu/projects/history/papers/karg74.pdf.
24. Kesteloot, L. 2009. *Coding Machines*; https://www.teamten.com/lawrence/writings/coding-machines/.
25. Munroe, R. 2020. xkcd: Dependency. Webcomic; https://xkcd.com/2347/.
26. National Security Agency. 2022. Software memory safety. NSA Cybersecurity Information Sheet version 1.1 (updated April 2023); https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.
27. Open Source Insights website; https://deps.dev/.
28. Open Source Security Foundation website; https://openssf.org/about/.
29. OSV website. A distributed vulnerability database for Open Source; https://osv.dev/.
30. Pan, R. 2022. Announcing OSV-Scanner: vulnerability scanner for open source. Google Security blog; https://security.googleblog.com/2022/12/announcing-osv-scanner-vulnerability.html.
31. Reproducible Builds website; https://reproducible-builds.org/.

32. Thompson, K. 1984. Reflections on trusting trust. *Communications of the ACM* 27(8), 761–763; https://dl.acm.org/doi/10.1145/358198.358210.
33. Turton, W., Gillum, J., Robertson, J. 2021. Inside the race to fix a potentially disastrous software flaw. Bloomberg News; https://finance.yahoo.com/news/inside-race-fix-potentially-disastrous-234445533.html.
34. Xiao, C. 2015. Novel malware XcodeGhost modifies Xcode, infects Apple iOS apps and hits app store. Palo Alto Networks; https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/.
35. Zetter, K. 2023. The untold story of the boldest supply-chain hack ever. *Wired*; https://www.wired.com/story/the-untold-story-of-solarwinds-the-boldest-supply-chain-hack-ever/.

Russ Cox *works at Google on the Go programming language team. He led the development of Go for more than a decade, with a particular focus on improving the security and reliability of using software dependencies. With Jeff Dean, he created Google Code Search, which let developers grep the world's public source code. He also worked for many years on the Plan 9 operating system from Bell Labs and holds degrees from Harvard and MIT.*