THE LINUX FOUNDATION

Open Source Software Supply Chain Security

A Publication of The Linux Foundation
February 2020

www.linuxfoundation.org

Introduction

As cybersecurity incidents have continued to grow in magnitude, frequency, and consequences, both public and private sector attention has turned to questions of what, if anything, organizations may do to better manage the risks of today's modern, connected world.

While innumerable strategies, frameworks, and "best practices" guides have emerged, few of which agree and some of which outright contradict each other, general consensus has grown around the need for increased diligence regarding the "software supply chain."

But even the term software supply chain raises its own host of questions. What exactly counts as "software?" Who are the "suppliers?" And, perhaps most importantly, what does the so-called "software supply chain" actually consist of?

Modern software development has become a significantly more complex process than in past decades, primarily due to its increasingly distributed nature. It is now exceedingly rare for organizations to write the majority of their software in-house. Instead, most organizations leverage high-quality and freely-available open source software (OSS) to create the bulk of their software products, with proprietary software either acting as the "glue" that holds the various pieces of OSS together, or as a unique service or function sitting on top of it.[1]

Consequently, a single piece of "software" is actually more properly understood as the sum of the various software packages of which it is comprised. Further, because OSS is designed and written not only by individuals or even individual companies, but often by multiple if not dozens of discrete and distributed developers working together, software "suppliers" run the gamut from hobbyists to multi-billion dollar companies. Every highly successful open source project has been built via an open framework of voluntary contributors by software engineers who devote their own time or their company's time to improve the project. Any policy discussion around a software supply chain must maintain this incredibly important open contribution framework.

Already, then, the "software supply chain" is massively complex. Adding further complications, there exist additional, more technical parts of the supply chain specifically involving how software is stored, retrieved, and analyzed that implicate additional factors. Where before software was typically delivered between businesses, or from businesses to customers, using physical mediums like CDs, today software—both OSS and proprietary—is more often stored in "repositories" and retrieved remotely over the Internet using tools such as project dependency managers (PDMs), more

1. While exact numbers vary, experts estimate that somewhere between 60-80% of modern software is comprised of OSS. See 2019 Open Source Risk and Analysis, Synopsys, available at https://www.synopsys.com/software-integrity/resources/analyst-reports/2019-open-source-security-risk-analysis.html, and The Forrester Wave™: Software Composition Analysis, Q1 2017, Forrester.

commonly referred to as the more general "package managers."[2]

Thus, the software supply chain can be understood to contain the following parts:

While the majority of recent attention regarding securing the software supply chain has focused on the first element in this chain—the developers—or the last part—the end users—weaknesses exist at all levels, leading to incidents like:

2015 Repackaging of Xcode for Malicious Code Distribution

In 2015, a security firm alerted Apple that thirty-nine applications available through the App Store were infecting iPhones and iPads. Once downloaded onto affected devices, the malicious applications connected to remote command-and-control servers and uploaded sensitive user information as part of a botnet. Further investigation revealed that the malicious code had been inserted into the applications through a "repackaged" version of Apple's official development platform Xcode. Dubbed "XcodeGhost," the Xcode-lookalike would add the botnet code alongside the otherwise legitimate apps developed using the platform.

While the apps were promptly removed and Apple took additional steps to ensure that legitimate developers had access to the official version of Xcode, the incident highlighted the risks to otherwise highly-curated software to vulnerabilities within its supply chain.[3]

2016 "left-pad" Dependency Incident

In 2016 and following a dispute about naming rights to an unrelated OSS package, a well-known developer removed all of their OSS packages from npm, the software registry used to distribute Node.js code. In total, the developer deleted 273 packages from npm altogether, but the problem rested primarily with one: left-pad. A deceptively simple package, left-pad right-justifies text for more human-readable text output. However, because it was relied upon by a number of critically important downstream packages—including Babel, a tool which "cleans up" and updates JavaScript code during the compilation process—its sudden disappearance "broke" many downstream pieces of code.

Another developer quickly replaced the vanished package with one that was functionally equivalent, but problems remained for some time as downstream developers scrambled to update their code and ensure that it referenced the new package rather than the

2. This paper uses the term "PDM" as defined in **this** article, instead of the more common "package manager," to differentiate between the types of package managers, as different types have wildly different security and operational practices.

3. Apple scrambles after 40 malicious "XcodeGhost" apps haunt App Store, Dan Goodin, ArsTechnica (Sep. 25, 2015), https://arstechnica.com/information-technology/2015/09/apple-scrambles-after-40-malicious-xcodeghost-apps-haunt-app-store/.

old. In addition to highlighting in stark clarity the risks developers face in relying upon upstream packages over which they have little to no control, the incident also revealed a widespread "dependency" issue, where developers who had no idea and no intention to rely upon left-pad were also affected due to the package being nested within their upstream dependencies.

Though the left-pad incident happened over three years ago, many of these same problems remain.[4]

2017 Python Package (PyPI) Highjacking

In 2017 attackers created malicious libraries with names that "closely resembled" the names of built-in Python libraries, and unsuspecting developers downloaded the malicious ones instead. The malicious packages contained the same code as the originals, except for an installation script that was changed to include malicious code.[5]

2018 Python Package Highjacking

In 2018, a cryptocurrency-stealing Python package called "Colourama" was discovered in the Python software repository. The name of the package was deliberately meant to be associated with and/or confused for the legitimate package "Colorama," one of the top-20 most-downloaded pieces of software within the Python repository.

While the malicious package had only been downloaded 151 when it was discovered, clearing the infection from affected devices took significant effort, and highlighted the vulnerability of software repositories to such tactics.[6]

2018 Backdooring of "event-stream" Library

In 2018, one of the most widely-used JavaScript libraries was backdoored to insert cryptocurrency-stealing code into the package. Notable not only for its subsequent reach—the event-stream library had been downloaded 2 million times at the time of the backdoor's discovery—the insertion was significantly more sophisticated than similar incidents.

To begin with, the malicious actors behind the backdoor managed to obtain legitimate publishing rights to the event-stream package itself by offering help to the beleaguered original developer. Once they had gained said access, they used it to add a benign package to the npm registry, flatmap-stream, and added the package

4. Rage-quit: Coder unpublished 17 lines of JavaScript and "broke the Internet", Sean Gallagher, ArsTechnica (March 24, 2016), **https://arstechnica.com/information-technology/2016/03/rage-quit-coder-unpublished-17-lines-of-javascript-and-broke-the-internet/**.

5. Goodin, Dan. 2017-09-16. "Devs unknowingly use "malicious" modules snuck into official Python repository: Code packages available in PyPI contained modified installation scripts." Ars Technica. **https://arstechnica.com/information-technology/2017/09/devs-unknowingly-use-malicious-modules-put-into-official-python-repository/**

6. Two new supply-chain attacks come to light in less than a week, Dan Goodin, ArsTechnica (October, 23, 2018), **https://arstechnica.com/information-technology/2018/10/two-new-supply-chain-attacks-come-to-light-in-less-than-a-week/**.

as a dependency in event-stream itself. About a month later, the malicious actors added malicious code to flatmap-stream—and therefore to event-stream—that targeted users of a popular cryptocurrency wallet software.

The staged attack, along with the effort undertaken by the actors to gain publishing access to the event-stream package, shows not only that there are weaknesses in the way that new code and new developers are scrutinized, but also that there is increasing value for malicious actors in putting forth such effort. This suggests that not only will similar attacks continue, they are likely to grow in frequency and sophistication.[7]

July 2019 Account Takeover of Popular Ruby Gems Package

In July 2019, an astute developer updating their codebase noticed a missing changelog.md file in one of their dependencies. The affected package, strong_password, had been updated from 0.0.6 to 0.0.7 with no explanation of the changes, and with discrepancies in the code hosted on Github and the code hosted within the Ruby repository. The developer investigated further and discovered that the package had been updated to include code that, upon execution within a production environment, would contact a remote URL and retrieve additional code. Once retrieved, the new code presented the opportunity for remote code execution within the infected environment.

The developer notified the original maintainer of the package, who then discovered that their account with the Ruby repository had been taken over. A malicious actor had compromised the maintainer's account, shifted ownership of the package, and then published the backdoored code. While unconfirmed, the original maintainer believes that a lack of two-factor or multi-factor authentication (2FA or MFA), along with potential password reuse, was to blame for the malicious actor's ability to gain access to their developer account.[8]

Because dependencies such as strong_password are deployed within a wide array of environments, and because they typically are associated with well-known developers who have developed a reputation for trustworthiness, the value in being able to take over such developer accounts is high. Similar attacks are likely to increase.

2018-2019 Webmin Compromise

Beginning in April 2018 and discovered in August 2019, the popular Webmin administration tool was backdoored by an unknown malicious actor. The change was relatively small, but allowed for significant impact: malicious actors who used the backdoor could leverage a specially-crafted URL to send commands to infected servers, which would then execute the commands with the highest level (root) privileges.

According to Webmin's developer, the server containing the Webmin source code was exploited in April 2018, allowing malicious code to be inserted. The attackers then altered the associated server logs so that it looked like the file had not been updated in some time, hiding

7. Widely used open source software contained bitcoin-stealing backdoor, Dan Goodin, ArsTechnica (November 26, 2018), **https://arstechnica.com/information-technology/2018/11/hacker-backdoors-widely-used-open-source-software-to-steal-bitcoin/**.

8. strong_password v0.0.7 rubygem hijacked, Tute Costa (July 3, 2019), **https://withatwist.dev/strong-password-rubygem-hijacked.html**.

the change from common detection mechanisms like code comparison tools. The altered code persisted either through lack of detection or additional malicious actions through August 17, 2019, when an outside party discovered that the backdoor had been released as part of a 0-day exploit.

While the infection was cleared and additional steps were undertaken by Webmin's maintainers, the incident serves as another example of both the vulnerability of such software and their continuing value to malicious actors.[9]

August 2019 Discovery of 11 Backdoored RubyGems Libraries

In August 2019, analysis by a developer examining Ruby libraries identified 11 backdoored packages. In each case, the backdoor allowed malicious actors in possession of pre-chosen credentials to remotely execute code on infected servers. The infected packages also allowed for the mining of cryptocurrencies.

While it is unclear how each of the libraries became infected, for at least one of the packages, the modification of the code was possible due to the compromise of a developer account. That account had been using a previously-cracked password, and was not protected by 2FA or MFA.[10]

In addition to the examples described above, some cybersecurity incidents in which the exact cause is unknown, but which are suspected to be connected to the software supply chain itself, have had consequences beyond the technical:

Steven Murdoch (sjmurdoch)

08/07/2019 06:13

We aren't certain how the malicious code got on the @British_Airways server, but I hope £183m is enough to revisit the development community's decision that build systems should download code from random Internet strangers and run it on your production environment

These incidents—among other, less notable incidents much like them—demonstrate the weaknesses inherent in the current policies, processes, and procedures used by package managers and repositories. Exacerbating the situation further, because these elements of the supply chain are indispensable to modern software development, organizations in nearly all cases must use them, thereby exposing them to high levels of risk that are typically beyond their control.

Finally, there exists one final element, separate from the software supply chain but indispensable to it: vulnerability databases. Given the distributed and overwhelmingly complex nature of modern software development, the identification, analysis, remediation, and tracking of vulnerabilities discovered in deployed software is critical. However, the world's most depended-upon vulnerability tracking database—the National Vulnerability Database (NVD),

9. The year-long rash of supply chain attacks against open source is getting worse, Dan Goodin, Ars Technica (August 21, 2019) **https://arstechnica.com/information-technology/2019/08/the-year-long-rash-of-supply-chain-attacks-against-open-source-is-getting-worse/**; Webmin page explaining exploit, Webmin, **http://www.webmin.com/exploit.html**.

10. Id.

fed by the Common Vulnerabilities and Exposures (CVE) program—continues itself to struggle with the growth, pace, and complexity of modern software development.[11] These struggles have direct impacts on developers and companies that rely on the CVE and NVD programs, and impact the security and reliability of the software supply chain as a whole.

This white paper will explore the security and reliability issues currently affecting the software supply chain, and identify where and how changes may be made to improve it overall.

11. House panel rips CVE contracting and oversight policies, Sean Lyngaas, Cyberscoop (Aug. 27, 2018), **https://www.cyberscoop.com/cve-mitre-house-energy-and-commerce-committee/**.

Examination of the Software Supply Chain

Developer Practices

In the graphic introduced earlier in this paper, developers are listed as the first link in the software supply chain. While this is true, developers also exist at and between every other link: they are the common element threaded through each piece of the software supply chain. Developers choose the programming languages they use, and therefore the repositories and PDMs upon which they must rely. They choose the libraries and packages and other OSS that become the building blocks of companies' completed products purchased by end users. They are, put simply, the single most indispensable element of the software supply chain.

However, even given their importance to and influence over that supply chain, many developers do not follow application security best practices when developing software. There are a variety of reasons for this. For one thing, as previously discussed, software development today is a massively complex process. Consequently, it is far easier said than done to "follow security best practices," especially given the sheer number of proposed "best practices," and the fact that one advocate's strategy might be another's fatal weakness. For another, security is often seen—and often in fact acts as—an impediment to developer and user experiences with software. As a result, many developers avoid or minimize their usage of what would otherwise be sound security practices.

This ignorance of or reluctance to fully embrace security practices has a number of consequences, many of which were highlighted in the supply chain incidents described above. Many of those incidents could have been avoided had the developers involved used well-known and widely accepted security practices, such as:

- Using two-factor or multifactor authentication (2FA or MFA) for developer accounts and other important accounts associated with a given project's design, deployment, and maintenance;
- Requiring that projects support change control tracking throughout the development process, to include who made changes and when those changes were made;
- Ensuring that projects have a unique version identifier for each release, thereby allowing downstream users to track new releases and establish controls and verification mechanisms around them;
- Integrating testing into the project's development lifecycle to check not only for common bugs and unexpected behavior, but also for malicious changes that may have been made without the developer's knowledge;
- Leveraging tools or other mechanisms to ensure that a project's dependencies are documented

and communicated in such a way as to be readily consumable by downstream users;

- Leveraging tools to ensure that dependencies are appropriately tracked, analyzed, and managed;
- Cryptographically signing or otherwise presenting verifiable proof of a project's integrity;
- Tracking and remediating vulnerabilities both within newly-developed code and in OSS dependencies integrated into a given project.

While it is certainly untrue to claim that all or even most developers fail to apply these and other best practices, many don't. Some may have valid reasons for not doing so—they don't have the resources, expertise, or support necessary—and others may simply be unaware that they should be leveraging them. In both cases and many more, however, the absence of these and similar best practices have severe downstream consequences not only for the developer, but for the end users of the impacted software.

Repositories

As information technology practices have evolved over the years to take advantage of rapid increases in network speeds, cloud computing, and other similar advances, so too has software development. Where before much software development was performed in-house, using code licensed from partners or vendors, now the majority of development involves ingesting large amounts of OSS retrieved free and often without restrictions over the Internet. While the exact locations of this stored software may vary, many developers rely on software storage sites known as "repositories."

At its most basic, a software repository is a server that contains a collection of software packages.[12] These packages may vary from small utility libraries up to full command line tools and development frameworks. Traditionally Linux systems rely on an Operating System repository to manage the applications—and dependencies of those applications—based on the Linux distribution that they are using. The developers of that distribution maintain all of the packages in a set of repositories and keep them up-to-date based on the releases of the upstream software packages, as well as fixing reported security and other bugs in those packages where needed.

With the growth of interpreted programming languages, starting with perl, it became advantageous for a language to offer an expanded repository of "helper" libraries for users of that programming language. Due to the size of these repositories, they typically have fallen outside of the main packaging ability of individual Linux distributions. Because of this growth in these language specific repositories, it is practically required that any developer using those languages also use the language repository tools to install needed dependencies, as well as for when developed software needs to be run on non-development systems.

Because a large percentage of software development today relies on OSS, and because a large amount of the world's most depended-on OSS is written in languages that rely on language repositories for their libraries, developers must retrieve some subset of software from these repositories. However, for a variety of historical and economic reasons, such language repositories in many cases lack even basic security or quality controls. For example:

12. Repositories, Ubuntu Documentation, **https://help.ubuntu.com/community/Repositories**.

- Few language repositories currently provides for a mechanism through which stored code is examined for its purpose, increasing consumer confusion and in some cases enabling malicious activity;
- Few language repositories perform systematic checks for vulnerabilities in stored code or for deprecated packages;
- No language repository currently provides for a mechanism through which a consumer can tell if one piece of stored code is derived from another, which limits their ability to discover whether vulnerabilities or other issues are being inherited from dependencies;
- In most language repositories, weak or missing authentication and publisher verification mechanisms create uncertainty and risk over the provenance of stored code;
- Some language repositories do not provide two-factor or multi-factor authentication of developer accounts, and those that do often do not require it, encourage it, or indicate to others that the developer account (and packages that account controls) is weakly protected;
- While many language repositories provide for code signing, few, if any, provide for or enable strong mechanisms for verifying the validity of those signatures;
- Some language repositories contain restrictive End User License Agreements (EULAs) that limit the ability of conscientious consumers from attempting to perform their own security and quality analysis on stored code.
- Many language repositories do not verify, or do not make it easy for others to verify, that compiled or generated packages are necessarily generated from the expected, publicly-available source that is inspectable by others.

While some language repositories have taken steps to address subsets of these concerns, no repository has developed mechanisms for addressing them all. Further, for some language repositories that have attempted to address these concerns, many have chosen to do so by "commercializing" the repository itself, whereby paying customers receive "premium" features like those listed above. Consequently, many basic and necessary security and quality controls remain outside the reach of many everyday consumers.

Project Dependency Managers ("Package Managers")

As a result of software having now "eaten the world," users, developers, and maintainers of software require straightforward and robust tools through which they may interact with and efficiently manage the large—and growing ever-larger—amounts available today. While many such tools exist, the most popular and widespread of them are "package managers"—software tools that themselves automate the process of installing, upgrading, configuring, and removing software packages, libraries, and other such files from a given system.[13] In particular, a certain

13. Package manager, Wikipedia, https://en.wikipedia.org/wiki/Package_manager; What is a package manager, Debian, https://www.debian.org/doc/manuals/aptitude/pr01s02.en.html.

type of package manager called a "project/application dependency manager" (PDM) is the tool of choice.[14]

By leveraging PDMs, users are able to turn the previously complex, multi-step process of locating, installing, and configuring software into a single step. Under the hood, PDMs make connections to language repositories like those described above, retrieve the software specified by the user, including all the software it indirectly depends on, and—where applicable—configure that software as desired. In simplifying software retrieval and management in this way, PDMs have greatly reduced the levels of expertise and resources necessary for modern software development.

However, PDMs are simply software retrieval tools. They do not, nor is there currently any feasible way to modify them to, check whether the retrieved software:

- Has known security or reliability issues;
- Contains unexpected or malicious behavior;
- Has a misleading package name that suggests "typosquatting" and/or is the name of a built-in library, nor do many implement defenses such as obscurity alerts.[15]

Instead, these practices should be—but as discussed above, usually aren't—performed within other parts of the software supply chain, frustrating efforts by PDM users and PDM maintainers themselves to ensure some degree of security and quality within retrieved software. This is especially problematic since, as evidenced by the increasing frequency of security incidents involving PDMs, the weaknesses inherent within their current procedures are becoming popular avenues of exploitation for malicious actors.

Vulnerability Databases

As discussed above, a piece of modern "software" is almost guaranteed to be a composition of many software packages woven together. These "building block" packages may be proprietary code, licensed code, or OSS, but the bottom line remains that they often number from dozens to thousands for each discrete software product. While this method—as shown by development trends and analyses of the ecosystem—provides significant benefits, it also leads to a notable risk: developers and companies today must worry not only about vulnerabilities and bugs discovered in their own code, but those discovered in each and every one of the software packages on which their product depends.

Just as modern software development outpaced strictly in-house development strategies, the sheer number, variety, and uniqueness of vulnerabilities and bugs discovered in modern software means that strictly in-house vulnerability tracking is impossible. This was a reality recognized early-on by the software community, and led to the creation of a standardized, United States-based set of programs for naming, describing, and tracking vulnerabilities and bugs: the Common Vulnerabilities and Exposures (CVE) program and the National Vulnerability Database (NVD) program.[16]

14. See supra note 2.

15. Vaidya et al, "Security Issues in Language-based Software Ecosystems, March 6, 2019, **https://arxiv.org/abs/1903.02613**

16. While the CVE and NVD programs are generally relied-upon worldwide, it is important to keep in mind that they do not encompass all countries' programs in all cases.

These two programs have existed for over two decades and have become the foundation for many modern cybersecurity tools, products, and practices.[17] However, in recent years both programs have publicly struggled with the staggering growth of new technologies, which has led and continues to lead to a significantly increased influx of requests for inclusion in the NVD. These struggles have created a number of downstream issues, including:

- Missing or rejected vulnerabilities, leading to incomplete coverage in the NVD;[18]
- Severely delayed assignment of vulnerability identifiers, creating risks for downstream parties who remain unaware and likely unprotected from the issue;
- Poorly contextualized descriptions of vulnerabilities, increasing the difficulty of mitigation and vulnerability management;
- Overinflated and/or underplayed vulnerability scores, leading to misallocated resources and in some cases vulnerability "fatigue."
- Abuse by developers who claim inflated numbers of vulnerabilities in order to pad resumes, creating "false positives."
- Difficulty in revoking assigned vulnerabilities when they are found to be invalid, creating confusion and lack of trust in overall program.
- Abuse by engineers in organizations who see CVE assignments as a way to circumvent difficult management procedures preventing them from doing normal software upgrades.
- Discomfort with the CVE program because it is managed by a US federal agency.
- Inability to handle ongoing and complex vulnerabilities that require multiple fixes across multiple packages over extended periods of time.

Consequently, many stakeholders who rely on the CVE and NVD programs—stakeholders which include nearly all modern companies, federal agencies, and other organizations—are left with an incomplete picture of their vulnerability exposure. Worse still, the lack of coverage in the NVD can lead to a false sense of security, where stakeholders believe that their products remain relatively secure and reliable, since there are fewer or no associated CVE entries in the NVD.

End User Practices

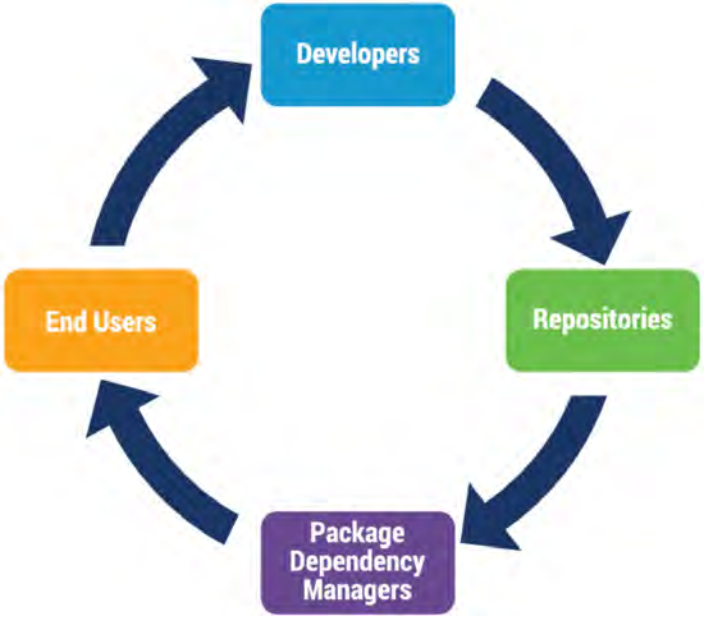
Given their place at the end of the software supply chain, end users arguably have the least control over security—or any—practices undertaken by the parties responsible for the parts of the supply chain described in the earlier sections. That understanding, however, misses the fact that while the software supply chain is most often discussed as a straight-line continuum starting with developers writing code and terminating when end users acquire products, there remains a

17. The NVD is considered so important that in 2018 it was exempted from the U.S. government shutdown. See "Closed Down: Government Shutdown Impacts Enterprise Security, December 31, 2018, **https://duo.com/decipher/government-shutdown-impacts-enterprise-security**

18. Over 6,000 vulnerabilities went unassigned by MITRE's CVE project in 2015, Steve Ragan, CSO Online (Sep. 22, 2016), **https://www.csoonline.com/article/3122460/over-6000-vulnerabilities-went-unassigned-by-mitres-cve-project-in-2015.html**.

distinct, better method for interacting with it: as a loop, rather than a chain. In other words, while the graphic introduced early in this white paper presented the software supply chain as this:

It can and arguably should be approached as this:



For end users, there are typically two ways in which they interact with software generally and the software supply chain in particular. In many cases, end users will source solutions from technology vendors providing commercial support, in which case they may never make any decisions about selection of PDMs or OSS packages. What end users do have, but which many underutilize, is control over their acquisition requirements.

This paper has described numerous practices which segments of the software supply chain either aren't but should be following, or practices that those segments are following, but shouldn't. While some of these practices are highly technical and relate to specific, highly nuanced parts of the software supply chain, others are more general and easier to isolate as standalone "best practices." And these standalone best practices can easily become acquisition requirements that end users insert into their contracts as they negotiate with technology providers. For example, end users may require that:

- Dependency lists, software bills-of-material, or other such component tracking mechanisms are provided in a robust and transparent way.
- Vulnerabilities within products maintained by a technology provider that are judged to have specific impacts must be remediated within certain timeframes.
- Developers must use 2FA or MFA with any accounts related to the development of the software being acquired.

There remains the other case, however, where end users will choose to self support their own solution with open source packages. requiring that they then are capable of applying the same practices discussed in the Developer Practices section of this paper. In addition, beyond those practices and the acquisition practices outlined above, there are practical steps that end users can take to: inspect and verify the trustworthiness of software, download their software from trustworthy locations, ensure that the software have requested is the software they wanted, verify that the software

that they received is the software that they wanted, and limit the privilege they give software to reduce the impact of supply chain problems. However, the following still remain true:

- It is difficult to determine if software is trustworthy, both because of a lack of agreed-upon understanding as to what it means for software to in fact be "trustworthy," and because of a lack of effective tooling.
- Similarly, it is challenging to determine if download locations, such as the repositories discussed earlier, are trustworthy.
- Users often fail to ensure that the software they request is the software is in fact the package they believe it to be, and is not malicious, fraudulent, or otherwise incorrect.
- Similarly, users often fail to verify that the software they received is the software that they wanted by, for example, checking digital signatures, and some users run code immediately upon receipt, without performing security, quality, or other checks.

End users are arguably in the best and worst positions to influence the software supply chain. For those that are acquiring technologies from vendors, they may be able to leverage their acquisition practices to encourage them to apply security best practices that they might not otherwise, but they still have less ability to correct or even have visibility into deficiencies in the products they receive. For those that choose to self-source their software, these end users need recognize that they then become, in essence, developers, and behave appropriately. In both cases, they must recognize that changes in modern software development require changes in their own behavior.

Conclusion

Modern software development is a massively distributed process, with a "supply chain" that often involves dozens, if not thousands, of individual developers, organizations, pieces of software, and the tools, policies, and procedures to weave them altogether. While this trend—which only continues to increase—has created significant value by reducing barriers to entry for new programmers, decreasing mean-time-to-market for products, and ensuring a global community of expertise, it also creates opportunities for risk and exploitation.

Software repositories, package managers, and vulnerability databases are all necessary components of the software supply chain, as are the developers and end users who leverage them. Unless and until the weaknesses inherent within their current designs and procedures are addressed, however, they will continue to expose the companies and developers who rely upon them to significant risk. This white paper was written to highlight known problems within the software supply chain, and serve as a call to action to address them. The Linux Foundation will be convening a meeting of global technology leaders in working across application and product security groups in order to design collective solutions to address these problems.

The Linux Foundation promotes, protects and standardizes Linux by providing unified resources and services needed for open source to successfully compete with closed platforms.

To learn more about The Linux Foundation or our other initiatives please visit us at **www.linuxfoundation.org**