For Good Measure

Counting Broken Links: A Quant's View of Software Supply Chain Security

DAN GEER, BENTZ TOZER, AND JOHN SPEED MEYERS

Dan Geer is a Senior Fellow at In-Q-Tel and a security researcher with a quantitative bent. He has a long history with the USENIX Association, including officer positions, program committees, etc. dan@geer.org

Bentz Tozer is a Senior Member of Technical Staff in In-Q-Tel's Cyber Practice, where he identifies and works with startups with the potential for high impact on national security. In previous roles, he has performed security research and software development with a focus on IoT devices and embedded systems. He has a PhD in systems engineering from George Washington University. btozer@iqt.org

John Speed Meyers is a Data Scientist in IQT Labs and a researcher who focuses on cybersecurity, especially network traffic analysis and software supply chain security. He holds a PhD in policy analysis from the Pardee RAND Graduate School. He's ambivalent about computers. jmeyers@iqt.org

"Without data, you're just another person with an opinion."

—W. Edwards Deming

It is tempting to tune out the cyberattack news cycle, dismissing the seemingly random assortment of reported attacks as nothing more than chance encounters of lucky defenders with unlucky attackers. It is easy to see the noise. It takes more effort—what amounts to digital wading—to find the signal, especially when dealing with public reporting on cyberattacks, but wade we did to assess the extent of software supply chain attacks. These attacks prey on the trust that makes code reuse possible and that produces the modern software cornucopia enjoyed by software developers and consumers alike.

We read of the event-stream attack [1] where an individual with malicious intent took over a popular JavaScript library and slipped code that steals cryptocurrency wallet credentials into a dependency of the associated npm package; ShadowHammer [2] in which a back-doored update utility with a legitimate certificate was distributed through official channels; and of barrages of typosquatting attacks on package registries [3] such as npm, RubyGems, and PyPI. Learning about these incidents led us to collect and review reports of software supply chain attacks in order to better understand the characteristics of these incidents and trends. While doing so, we also noticed the emergence of more systematic research. There's been measurement of the susceptibility of package manager users to typosquatting [4], the creation of a sophisticated malware detection pipeline for package managers [5], the building of a package manager download client that protects users from malware [6], and other efforts to gather and classify reports of software supply chain compromises [7–9].

We collected our data set in order to answer basic questions about software supply chain attacks such as: How frequent are known instances of attacks? What is the relative occurrence of different attack types? What is the length of time from initial deployment of such attacks to public discovery? However, while attempting to obtain these quantitative metrics, we were also faced with more fundamental, qualitative questions, like: What is (and is not) considered a software supply chain attack? What are the definitions of different attack types? How should attack impact be defined and measured? We report on how we built this data set, answer the quantitative questions that we set out to understand, and then, based on these findings, offer some thoughts on how to use data to combat software supply chain attacks.

Software Supply Chain Compromises: Data Set and Analysis

We built a data set based on public reporting of software supply chain security compromises, which is available at https://github.com/IQTLabs/software-supply-chain-compromises. This data set defines software supply chain attacks as attacks that intentionally insert malicious functionality into build, source, or publishing infrastructure or into software components with the goal of propagating that malicious functionality through existing distribution methods. Exploiting a vulnerability found within a software's supply chain was insufficient

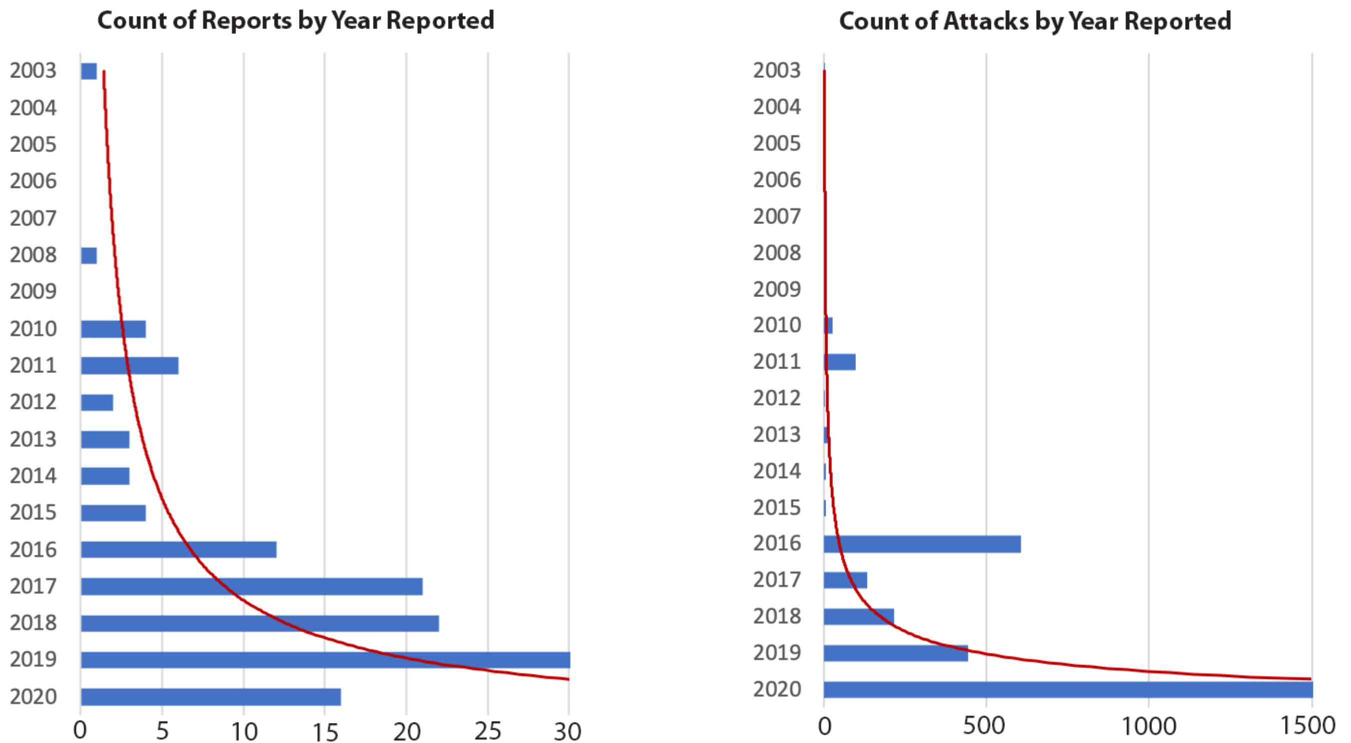

Figure 1: The number of reports and attacks by year reported

to merit inclusion. We attempted to count three different units of software supply chain security compromise: attacks, reports, and incidents. An "attack" is a distinct action to compromise a software supply chain, e.g., deliberate introduction of a vulnerability into source code. A "report" is a public disclosure of one or more software supply chain attacks, e.g., a blog post from a security researcher who has identified the existence of an attack in an open source library. An "incident" is a single instance of an attack reaching a target, e.g., the download of a compromised application from a download server. In effect, we hoped to use "incident" as a measurement of the impact of an "attack."

Figure 1 describes the trend of reports and attacks by the year in which the report or attack was announced, a decision that reflects the limited data available about the starting date of many of these attacks. The number of reports and attacks has been increasing over time; though included here, years before 2010 include only a count of one in 2003 and one in 2008. Because reporting can be delayed, 2020 and, perhaps, 2019 may be undercounted as yet. (The lines are power-law fits; the exponent is 1.2 for count of reports and 2.5 for count of attacks.)

Table 1 groups these reports and attacks into major and minor categories based on the actions of the attacker, not the perspective of the victim. These categories were influenced by the work of the "in-toto" project [7] but were adapted and extended organically while collecting this data set and do not represent any established classification scheme. The development of a standard taxonomy in the future would be beneficial.

Table 1 tentatively suggests that there is an inverse relationship between the estimated level of effort required to execute an attack type and the frequency of reported attacks of that type. For example, 41 percent of attacks in our data set are classified as typosquatting, which merely requires the attacker to create an account on a package registry, identify unclaimed package names that are plausible misspellings of legitimate packages, and publish the malicious package under those names. On the other end of the spectrum, a build system compromise is one of the least common attack types in our data set, perhaps because it involves several challenging steps, including obtaining access to a target's build environment and introducing a compromised component into the build process without being detected. As we discuss below, while the success of an attack is difficult to objectively define and measure, it seems possible that the effort required to successfully deploy an attack is directly proportional to the number of incidents, with typosquatting attacks affecting fewer individuals than a build system compromise like ShadowHammer. This indicates that increasing the level of effort required to successfully deploy attacks on software registries could significantly reduce the quantity of reported software supply chain attacks.

Major Type	Build, Source, and Publishing Infrastructure					Software Registry			
Minor Type	Build System Compromise	Firmware Implant	Source Code System Compromise	Publishing: Certificate Attack	Publishing: Delivery System Compromise	Account Takeover	Dependency Compromise	Malicious Package	Typosquatting
Count	11:13	7:32	9:39	6:18	29:35	11:14	12:333	51:1,373	15:1,247

Table 1: Count of Reports:Attacks by major and minor categories. (Note: Both reports and attacks can be assigned to multiple categories.)

Data on incidents is not consistently available, and metrics are not consistent across attack types, making quantitative analysis across this data set infeasible. For example, download metrics were sometimes available for a malicious package accessible on a software registry, but the number of victims and number of times the package was executed after download are generally unknown. In other cases, a lower bound on the number of compromised applications has been reported, but the extent of propagation is unknown. However, the limited data that is available indicates the potential for widespread impact. In the case of the event-stream attack, there were over 7 million package downloads reported for the 53 days it was available on npm, and some unknown number of those downloads were of the compromised version, rather than older, non-malicious versions. For ShadowHammer, Kaspersky, which identified and reported the attack, stated that the attack affected over 57,000 of their users and estimated that the attack was distributed to over 1 million people. In the case of the typosquatting attacks identified by Perica and Zekić, the one package where information is reported was downloaded over 1700 times over nearly two years. While details are limited, it is clear that the potential force multiplication caused by the propagation of an attack through existing software delivery methods is highly appealing to attackers.

Another way to measure the success of a software supply chain compromise is the length of time it is active. Known as "dwell time," it is the number of days a threat remains undetected within a given environment; if the detection date is not available, we use the public announcement date. Figure 2 displays the distribution of dwell time for all reports with available data (n=59).

Defending the Supply Chain

Our analysis of known software supply chain attacks indicates that weaknesses in the software supply chain are numerous and are being appropriated by cybercriminals with increasing frequency. Per the usual, attacks as yet unknown are surely present, so you should assume that we are undercounting. Counting is hard. The spike in attacks in 2016 includes two special cases: a research project where a student intentionally uploaded 214 typosquatting packages to various software registries to measure download frequency, and the intentional deletion of 273 npm packages by a developer who was angry that npm took a package namespace away from him and wanted to wreak havoc. Earlier this year, ReversingLabs found 700+ malicious packages in RubyGems, while Duo found 500+ malicious Chrome extensions, both evidently the first time anyone had looked into such unknown unknowns. Counting is hard.

We believe there exist at least three major obstacles that prevent software developers, security teams, and software users from adequately protecting the software supply chain and from shielding themselves or their organization from such attacks.

First, there is a striking absence of data collection and analysis that would help identify and assess risks associated with these attacks, especially those involving open source software. This absence is surprising given the inherently public nature of open source software development and the ubiquity of open source dependencies within modern software applications. In other words, much of the data needed to identify potential and actual risks associated with a software component is hosted on publicly accessible development platforms like GitHub and is thus available to any interested party. Unfortunately, much of this information is not analyzed, allowing attackers to hide in plain sight.

To identify attacks, defenders will need to cull and analyze software development-related data. To start, a software bill of material that describes all dependencies of an application provides transparency and allows for investigation of direct and indirect dependencies. Other rich data sources are open source code repositories and package registries, which contain information about developer turnover, code commits, and version releases stored in these repositories. Defenders can also expose inconsistencies between independent data sources, verifying, for instance, the relationships between source code stored in

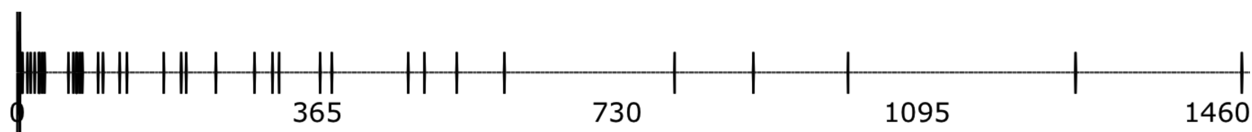

Figure 2: Distribution of dwell time in days for reports; dwell time for 12 reports was zero days; the median=34.

a repository and a released library or executable stored in a package registry. Countermeasures will also likely require an understanding of the individuals and organizations that directly or indirectly contribute to the development and distribution of software, especially the individuals or organizations that can publish changes. Importantly, defenders will need this information for all dependencies of a distributed software application, whether those dependencies are part of the build process, release process, or are included at runtime. As always, the wellspring of risk is dependence, and risk, unlike benefit, is transitive.

Second, existing application security products are unable to identify the distinctive characteristics of software supply chain attacks. Moreover, there has been limited adoption of what tools and processes do exist in order to prevent instances of supply chain attacks within released software. These issues force software developers and users to trust—but not verify—vendors and their products, rendering judgments about product software supply chain quality impossible and compelling acceptance of unknown risks within critical software.

These deficiencies should be a rallying cry for those who want to develop and build a new class of application security products, tools designed to uncover instances of software supply chain attacks. Existing application security tools are designed to identify defects in source code or executables and determine the conditions under which those defects are exploitable. These tools will not, however, identify a well-written software supply chain compromise. These attacks arise from the existence of undesired functionality with respect to the intended purpose of the software. This new breed of application security products will need context and an understanding of the expected use case of the application, concepts lacking in current application security products. This will not be easy.

Third, reducing the software supply chain attack surface also requires adopting existing technologies and processes that provide the information needed to verify the origin and content of source code and binaries, eliminating or mitigating many of the risks of compromise. One practical step is using digital signatures and certificates to verify file integrity. Employing reproducible builds and publishing relevant metadata to an immutable distributed ledger can also allow consumers to independently verify the integrity of a software component. Best of breed tools for network and endpoint protection should also be deployed within the development, publication, and operational environment to limit opportunities for compromise pre-commit.

Ultimately, securing the software supply chain of any product requires continuous assessment of components, vendors, and operational environments in addition to orchestration and analysis of relevant data. These processes, to be successful, require significant investment in automation and collaboration between all participants in the software supply chain. Nothing less is needed if sharing common software dependencies is to be a strength, the topic of this column two issues ago [10], rather than the liability it appears to be today.

References

[1] T. Hunter II, "Compromised npm Package: Event-stream," Intrinsic/VMware, November 26, 2018: https://medium.com/intrinsic/compromised-npm-package-event-stream-d47d08605502.

[2] "ShadowHammer: Malicious Updates for ASUS Laptops," Kaspersky Daily, March 25, 2019: https://www.kaspersky.com/blog/shadow-hammer-teaser/26149/.

[3] R. Perica and A. Zekić, "Mining for Malicious Ruby Gems," ReversingLabs, July 17, 2019: https://blog.reversinglabs.com/blog/suppy-chain-malware-detecting-malware-in-package-manager-repositories.

[4] N. P. Tschacher, "Typosquatting in Programming Language Package Managers," University of Hamburg, Bachelor Thesis, 2016: https://incolumitas.com/data/thesis.pdf.

[5] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Measuring and Preventing Supply Chain Attacks on Package Managers," arXiv, February 4, 2020: https://arxiv.org/abs/2002.01139.

[6] M. Taylor, R. K. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, "SpellBound: Defending against Package Typosquatting," arXiv, March 6, 2020: https://arxiv.org/pdf/2003.03471v1.pdf.

[7] S. Torres, H. Afzali, A. Sirish, and L. Puehringer, "Software Supply Chain Compromises," November 11, 2019: https://github.com/in-toto/supply-chain-compromises.

[8] M. Ohm, H. Plate, A. Sykosch, and M. Meier, "Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks," in *Proceedings of the 17th Conference on Detection of Intrusions and Malware,* and *Vulnerability Assessment (DIMVA 2020)*: https://link.springer.com/chapter/10.1007/978-3-030-52683-2_2.

[9] T. Herr, J. Lee, W. Loomis, and S. Scott, "Breaking Trust: Shades of Crisis Across an Insecure Software Supply Chain," Atlantic Council, July 6, 2020: https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/.

[10] D. Geer and G. Sieniawski, "Who Will Pay the Piper for Open Source Software Maintenance?" *;login:*, vol. 45, no. 2 (Summer 2020): https://www.usenix.org/publications/login/summer2020/geer.