2021

State of the Software Supply Chain

The 7th Annual Report on Global Open Source Software Development

PRESENTED BY

sonatype

Contents

Introduction

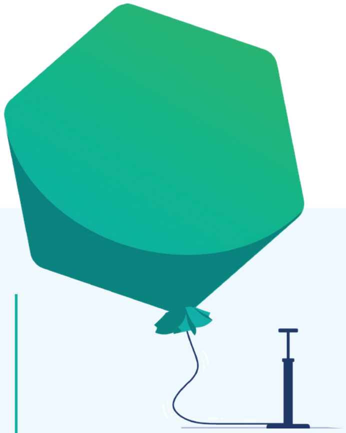
In 1849, French writer Jean-Baptiste Alphonse Karr famously said, “the more things change, the more they stay the same.” While he obviously wasn’t talking about open source software (OSS), digital supply chains, or application innovation during a global pandemic, he might as well have been. Indeed, in the year since we last published our State of the Software Supply Chain research, so much has changed in the world of software development, and yet, so much has stayed the same.

My oh my, how things have changed. COVID-19 fundamentally transformed how people live and work, how companies interact with customers, how customers shop and buy, and how physical and digital supply chains function. As the economic importance of digital innovation accelerated during the global pandemic, so too did the number of cyber-attacks aimed at exploiting software supply chains.

And yet, much has stayed the same. Top-performing companies like Apple, Goldman Sachs, and Amazon — and more recently, Zoom, Peloton, and Wayfair have mastered three key competitive advantages: knowing how to use open source and third-party innovation at scale, integrating security and risk controls into multiple phases of the software supply chain, and releasing higher quality code faster than their competitors.

Now in its seventh year, Sonatype’s *2021 State of the Software Supply Chain Report* blends a broad set of public and proprietary data to reveal important findings.

Together with our partners, we are proud to share this research. We hope that you find it valuable.



Open source supply is accelerating. The top four open source ecosystems released a combined 6,302,733 new versions and introduced 723,570 brand new projects. Collectively, these communities now contain a combined 37,451,682 different versions of components, representing a 20% year over year (YoY) growth in global supply of open source.

6 Million
new versions introduced in past year

37 Million
OSS component versions now available

73%
YoY growth of component downloads

Open source demand is exploding. In 2021 developers around the world will request more than 2.2 trillion open source packages from these same four ecosystems, representing a 73% YoY growth in developer downloads of open source components. Despite the growing volume of downloads, the percentage of available components utilized in production applications is shockingly low.

Open source vulnerabilities are most pervasive in popular projects. 29% of popular projects contain at least one known security vulnerability. Conversely, only 6.5% of non-popular projects do so. This dichotomy suggests that the vast majority of security research (blackhat and whitehat) is focused on finding and reporting vulnerabilities in projects that are most commonly utilized.

29% of popular projects contain known vulnerabilities, but only 6.5% of non-popular projects contain known vulnerabilities

650%
YoY increase in cyber-attacks aimed at open source suppliers

Supply chain attacks are increasing exponentially.

In 2021 the world witnessed a 650% increase in software supply chain attacks, aimed at exploiting weaknesses in upstream open source ecosystems. For perspective, the same statistic was 430% in the 2020 version of the report.

Standardizing architectural guidance is a path to huge efficiency gains.

Intelligent automation that standardizes engineering teams on exemplary open source projects could remove 1.6M hours and $240M of real world waste spread across our sample of 100,000[1] production applications. Extrapolated out to the entire software industry, the associated savings would be billions.

Intelligent automation could save companies
$192,000
per year

Some projects are better than others.

To avoid stale dependencies and minimize security risks associated with third-party open source, software engineering teams should actively embrace projects that consistently demonstrate low mean time to update (MTTU) values and high OpenSSF Criticality scores.

Dependency management practices vary widely among teams.

On average, enterprise Java applications utilize 10% of the components that are available for download in the Maven Central Repository. However commercial engineering teams actively update only 25% of components utilized. Such efforts are highly variable and frequently suboptimal, yet there is wisdom in the crowd that can be distilled. Newer versions of projects are generally better, but not always best.

Only 25%
of utilized components are updated actively

702
IT professionals surveyed

There is a disconnect between subjective survey feedback and objective data.

People believe they are doing a good job remediating defective components and indicate that they understand where risk resides. Objectively, research shows development teams lack structured guidance and frequently make suboptimal decisions with respect to software supply chain management.

[1] 100,000 anonymized, validated applications scanned by publicly available and commercial vulnerability analysis tools.

CHAPTER 1

Open Source Supply, Demand, and Security

The universal desire for faster innovation fundamentally requires that software developers reuse code frequently and efficiently. This, in turn, has led to a critical dependence on OSS libraries borrowed from third-party ecosystems. These third-party components and packages represent the building blocks of modern software development. But, what does open source supply look like? What are the demand dynamics? What is the relative quality and security of third-party code borrowed from open source suppliers?

Figure 1.1 summarizes statistics on supply, demand, usage, and security for the Java, JavaScript, Python, and .NET ecosystems. There are an order of magnitude more project versions than there are projects, with the average project having published eight to 12 versions, depending on the ecosystem. Older projects can have tens, or even hundreds of versions. Furthermore, a minefield of known (and unknown) security risks lurk within the 37 million available project versions. Such risk is far more prominent in popular projects.

> A minefield of known (and unknown) security risks lurk within the 37 million available project versions.

Open Source Supply

The global supply of open source libraries continues to grow exponentially, fueled by new versions of existing projects constantly being released, and by the creation of altogether new projects. Currently, the top four open source ecosystems contain a combined 37,451,682 components and packages. These same communities released a combined 6,302,733 new versions of components / packages over the past year and have introduced 723,570 brand new projects in support of 27 million developers worldwide.

Java

As of July 31, 2021, there are 430,995 unique projects (group-artifacts) available in the Maven Central Repository, up 13% from last year. This public catalog of Java components now offers developers a total of 7.3 million different versions (group-artifact-versions) of projects to choose from, up 19% from last year.

In the past year alone, project owners released more than 1.1 million new versions of existing components, and introduced 136,000 brand new projects to service and support approximately eight million Java developers.

FIGURE 1.1

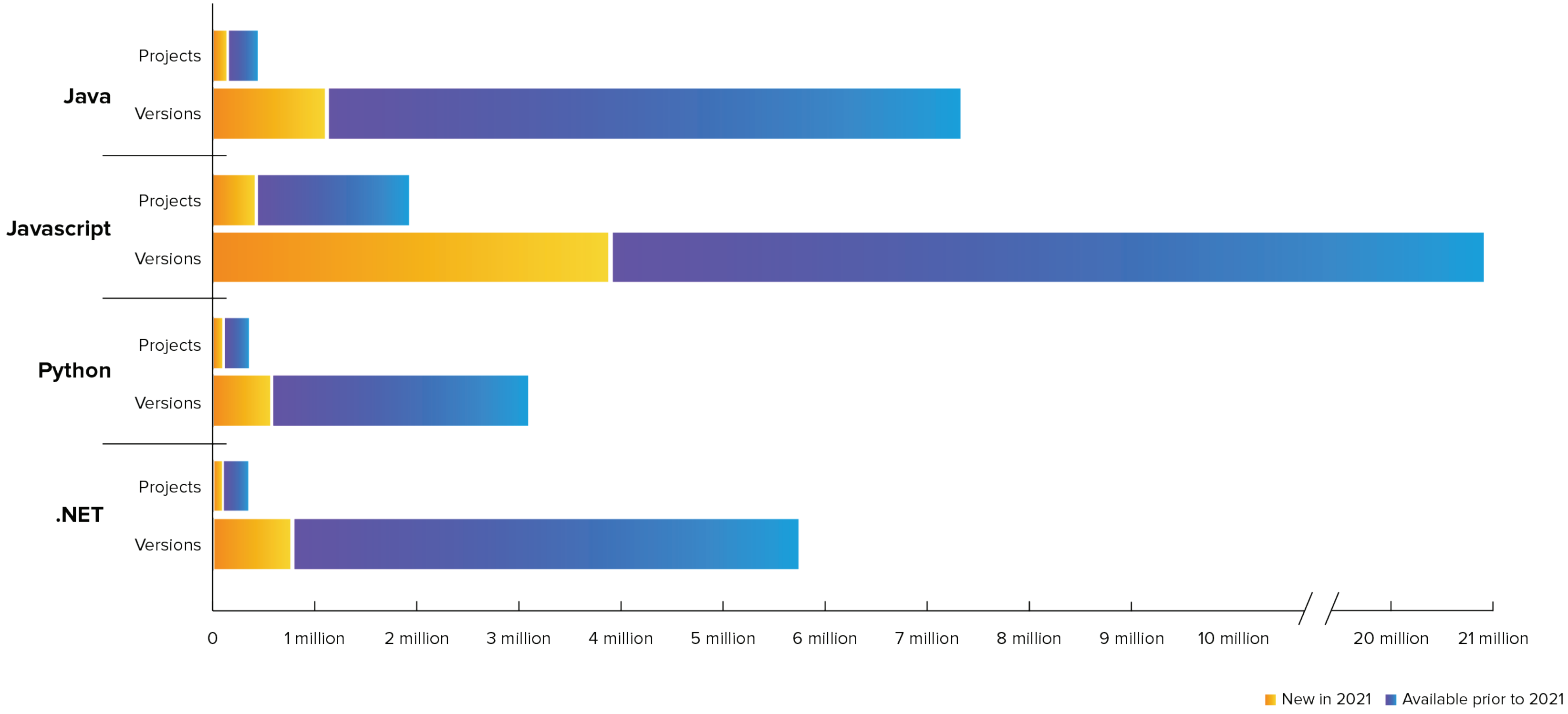
2021 SOFTWARE SUPPLY CHAIN STATISTICS

ECOSYSTEM	TOTAL PROJECTS	TOTAL PROJECT VERSIONS	ANNUAL DOWNLOAD VOLUME	YOY DOWNLOAD GROWTH	ECOSYSTEM PROJECT UTILIZATION	VULN DENSITY FOR UTILIZED VERSIONS *10% Most Popular*	VULN DENSITY FOR UTILIZED VERSIONS *90% Least Popular*
Java	431k	7.3M	457B	71%	15%	23%	4%
JavaScript	1.9M	21M	1.5T	50%	2%	39%	8%
Python	336K	3M	127B	92%[2]	4%	38%	8%
.NET	338K	5.6M	78B	78%	2%	15%	6%
Totals/ Averages	**3M**	**37M**	**2.2T**	**73%**	**6%**	**29%**	**6.5%**

[2] YoY growth estimated based on known PyPi downloads from March to August 2021 as queried from pypistats.org.

FIGURE 1.2

AVAILABLE SUPPLY OF OPEN SOURCE, 2021



JavaScript

As of July 31, 2021 there were 1,864,696 JavaScript packages available in the npm repository, up 16% from last year. This public catalog of JavaScript packages offers developers a total of 21,320,796 different versions. In the past year alone, npm project owners released 3,797,675 new versions of packages, and introduced 405,243 brand new projects to service and support approximately 12 million JavaScript developers.

Python

336,402 packages are available in the Python Package Index (PyPI), up 18% from last year. This public catalog of Python packages offers developers a total of 3,035,265 different versions. In the past year alone, python project owners released 556,327 new versions of packages, and introduced 93,032 brand new projects to service and support approximately eight million Python developers.

.NET

338,423 open source projects are available in the NuGet gallery, up 14% from last year. This public catalog of .net packages offers developers a total of 5,698,716 different versions. In the past year alone, the NuGet gallery released 756,444 new versions of packages, and introduced 87,268 brand new projects.

FIGURE 1.3

INCREASE IN DOWNLOADS

Year Over Year 2020 – 2021

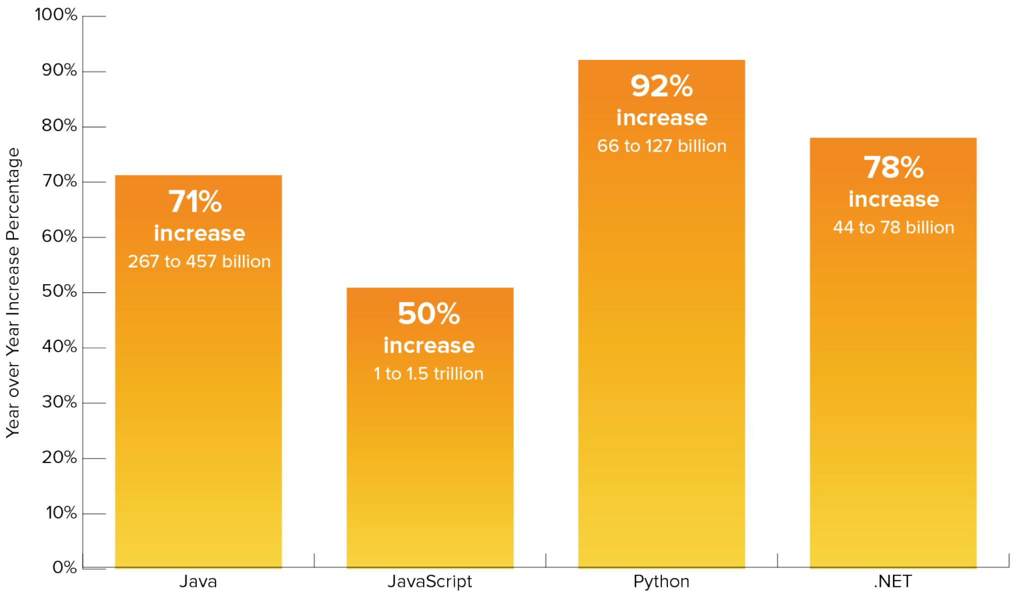
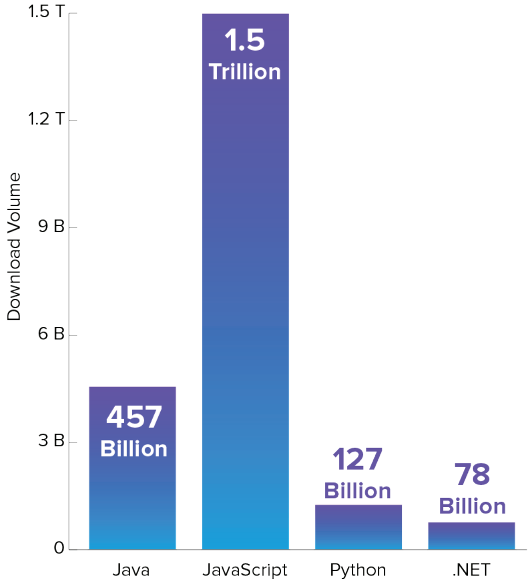


FIGURE 1.4

ANNUAL DOWNLOAD VOLUMES, 2021



Open Source Demand

In 2021, developers around the world will borrow more than 2.2 trillion open source packages or components from third-party ecosystems for two simple reasons: it makes life easier for software developers and it accelerates the pace of innovation.

Java

Through the first seven months of 2021, 267 billion Java components were downloaded from the Maven Central Repository. At this rate, the volume for 2021 is projected to be over 457 billion, a 71% YoY increase.

JavaScript

In 2020, JavaScript developers requested more than one trillion packages from npmjs. The volume for 2021 is expected to reach 1.5 trillion, a 50% YoY increase.

Python

In 2020, Python developers downloaded 66 billion[3] packages from PyPi. For the full year of 2021, PyPi download volume is expected to be 127 billion packages. YoY growth of PyPi download volume is estimated to be 92%

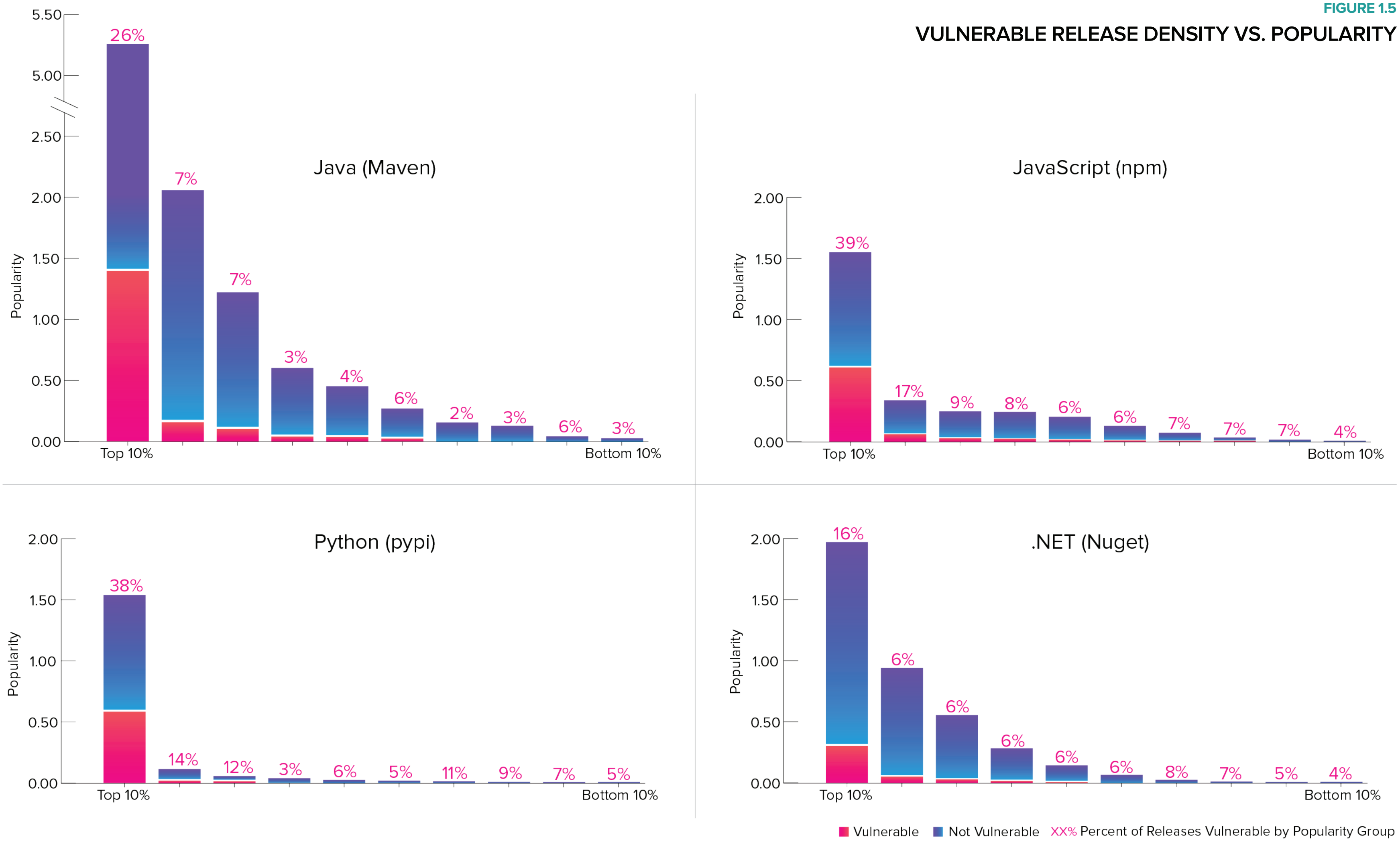
.NET

.NET developers were also eager to consume OSS packages over the past year. Developers downloaded 44 billion NuGet packages in 2020. In 2021 developers will download more than 78 billion packages, representing a 78% YoY growth.

[3] pypistats.org

FIGURE 1.5

VULNERABLE RELEASE DENSITY VS. POPULARITY



Open Source Security

The amount of third-party code flowing through software supply chains is massive. But is it secure? Are certain open source ecosystems more or less risky? Are certain projects safer than others? Are popular projects more or less likely to have known vulnerabilities? Here's what the data reveals.

When examining the top 10% of the most popular Java, JavaScript, Python, and .NET projects, 29% of them contain at least one known security vulnerability. Conversely, when examining the remaining 90% of less popular projects, only 6.5% of them contain known vulnerabilities. These findings indicate that the vast majority of security research (blackhat and whitehat) is focused on finding and reporting vulnerabilities in projects that are most commonly utilized.

In addition to studying the difference in vulnerability density between popular and non popular open source projects, we also present below the aggregate vulnerability density for each of the four ecosystems.

Java (Maven)

As of July 31, 2021, 612,988 (8.4%) of all component versions housed in Maven Central contain at least one known security vulnerability. To exclude low level security issues, we determined severity based on the Common Vulnerability Scoring System (CVSS), for medium (5), high (7), and critical (9), Of the issues identified:

- 356,808 (4.9%) had a CVSS of 9 or higher
- 488,826 (6.7%) had a CVSS of 7 or higher
- 598,364 (8.2%) had a CVSS of 5 or higher

For the past eight years, Sonatype has also analyzed the patterns and practices associated with Java components being downloaded from Maven Central. In 2020 and through the first seven months of 2021, 8% of the downloads had at least one known vulnerability.

JavaScript (npm)

As of July 31, 2021, 459,576 (2.2%) project versions housed in npm contain at least one known security vulnerability. Of the issues identified:

- 250,002 (1.2%) had a CVSS of 9 or higher
- 350,737 (1.6%) had a CVSS of 7 or higher
- 450,734 (2.1%) had a CVSS of 5 or higher

Notably, however, of the nearly 1.9 million JavaScript top level projects available, only 2% of those are being used with any regularity.

Python (pypi)

As of July 31, 2021, 147,994 (0.5%) package versions housed in the PyPI repository contained at least one known security vulnerability. Of the issues identified:

- 81,731 (.4%) had a CVSS of 9 or higher
- 111,970 (.4%) had a CVSS of 7 or higher
- 143,902 (.5%) had a CVSS of 5 or higher

.NET (Nuget)

As of July 31, 2021, 112,031 (2%) of package versions housed in the NuGet Gallery contained at least one known security vulnerability. Of the issues identified:

- 27,288 (.5%) had a CVSS of 9 or higher
- 99,096 (1.7%) had a CVSS of 7 or higher
- 110,764 (1.9%) had a CVSS of 5 or higher

Software Supply Chain Attacks Increase 650%

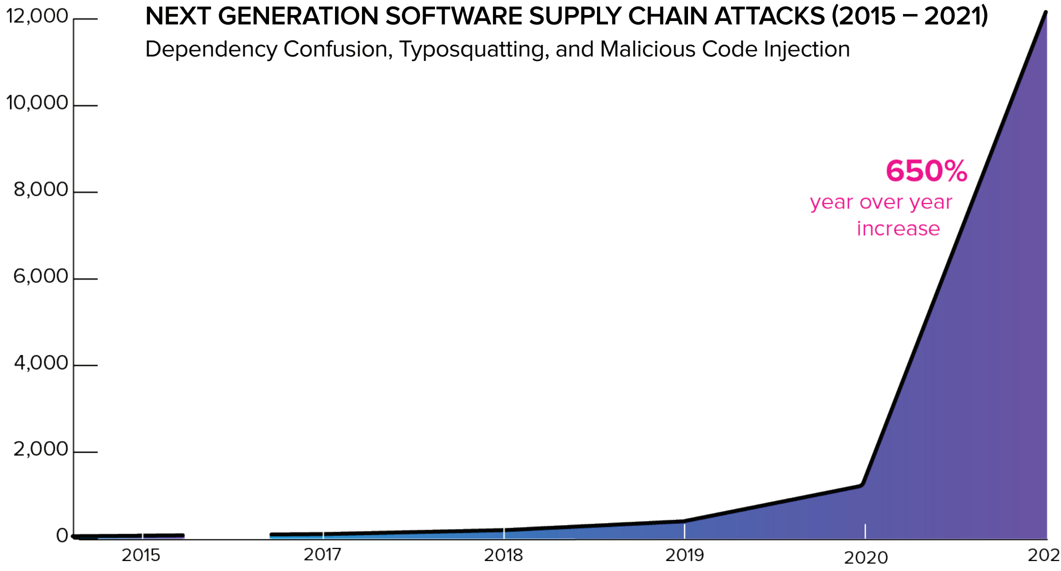
Members of the world's open source community are facing a novel and rapidly expanding threat that has nothing to do with passive adversaries exploiting known vulnerabilities in the wild — and everything to do with aggressive attackers implanting malware directly into open source projects to infiltrate the commercial supply chain.

Legacy software supply chain "exploits," such as the now infamous 2017 Struts incident at Equifax, prey on publicly disclosed open source vulnerabilities that are left unpatched in the wild. Next-generation software supply chain "attacks" are far more sinister, however, because bad actors are no longer waiting for public vulnerability disclosures to pursue an exploit. Instead, they are taking the initiative and injecting new vulnerabilities into open source projects that feed the global supply chain, and then exploiting those vulnerabilities

FIGURE 1.6

NEXT GENERATION SOFTWARE SUPPLY CHAIN ATTACKS (2015 – 2021)

Dependency Confusion, Typosquatting, and Malicious Code Injection



before they are discovered. By shifting their attacks "upstream," bad actors can gain leverage and the crucial benefit of time that that enables malware to propagate throughout the supply chain, enabling far more scalable attacks on "downstream" users.

According to security researchers at the University of Bonn, SAP Labs France, and Fraunhofer FKIE, "From an attacker's point of view, [large scale, public internet-based] package repositories represent a reliable and scalable malware distribution channel. Thus far, Node.js (npm) and Python (PyPI) repositories have been the primary targets of malicious packages, supposedly due to the fact that malicious code can be easily triggered during package installation."[4]

From February 2015 to June 2019, 216 software supply chain attacks were recorded. Then, from July 2019 to May 2020, the number of attacks increased to 929 attacks. However, in the past year, such attacks represented a 650% YoY increase (see Figure 1.6 above).

Dependency Confusion

The most common type of attack in 2021 has been Dependency Confusion (aka namespace confusion).

The novel, highly targeted, attack vector allows unwanted or malicious code to be introduced downstream automatically, without relying on typosquatting or brandjacking techniques. The technique involves a bad actor determining the names of proprietary (inner source) packages utilized by a company's production application. Equipped with this information, the bad actor then publishes a malicious package using the exact same name, and a newer semantic version, to a public repository, like npmjs, that does not regulate namespace identity. At this point, certain pipeline build tools will automatically fetch the newer, intentionally malicious version. In the past year, namespace confusion has alone accounted for instances of attempted or confirmed supply chain attacks.[5] This attack vector relies on the long established convention in some programming languages to fetch the "LATEST" version of any package.

Typosquatting

Typosquatting was the second most common technique over the past year. This indirect attack vector preys on developers making otherwise innocent typos when searching for popular components. For example, if a developer accidentally types "electorn" when their intention is to source "electron," they might accidentally install a malicious component of a similar name (see **electorn**, September 2020).

Malicious Source Code Injections

Malicious Source Code Injections are another type of attack that have been less prevalent in the past year compared to previous years. Such attacks involve injecting malicious source code directly into an open source project's repository, and have been conducted in various ways, including:

- stealing credentials from a project maintainer (see **rest-client**, 8/19)
- releasing new versions of a project to a public repository (see **bootstrap-sass**, 4/19)
- contributing pull requests to a project that includes malicious code (see **event-stream**, 11/18)
- tampering with open source developer tools that inject malicious code into downstream applications (see **Octopus Scanner**, 5/20).

With code injections, it is likely that no one knows the malware is there, except for the person that planted it. This approach allows adversaries to surreptitiously "set traps" upstream, and then carry out attacks downstream once the vulnerability has moved through the supply chain and into the code bases of thousands of companies.

Front Page News

In the past year, numerous high-profile and prominent attacks demonstrated how supply chain threats affect not only third-party application level libraries and tools, but also first-party source code. The European Union's Cybersecurity Agency (ENISA) predicts these types of supply chain attacks are **expected to increase 4x in 2021**.

SolarWinds — December 2020

The massive SolarWinds Orion attack publicized in December 2020 marked the most notable supply chain attack of the past year. The attack started with threat actors gaining access to SolarWinds' internal development tools to inject malicious code into SolarWinds' Orion update binaries. These trojanized updates delivered a backdoor known as SUNBURST and Solorigate, to systems running Orion platform versions. The impact?

[4] arxiv.org/pdf/2005.09535.pdf

[5] 12,000 statistic counts PyPI and npm 5k package flood as a single attack; not multiple attacks.

FIGURE 1.7

NEXT GENERATION SOFTWARE SUPPLY CHAIN ATTACKS

July 2019 – July 2021

JUL 2019

PyPI package discovered with a back-door vulnerability.
The package had been reported as containing a known vulnerability but was not removed from the public repository.

230 RubyGems pulled for typosquatting or impersonating popular open source packages.

AUG 2019

Adversaries compromised the account of a rest-client maintainer to install crypto miners.
Affected versions (1.6.10 to 1.6.13) were downloaded about 1,000 times. Similar vulnerabilities were found in Gem packages: coming-soon and cron_parser.

bb-builder removed from the npm repository.
The component stole login information from the computers it was installed on, sending it to a remote server.

OCT 2019

basic_authable
Three versions of this Gems package released in 2017 were yanked from the Gems repository due to their malicious nature.

NOV 2019

sj-tw-test-security
All versions of the component "contain malicious back-door code that downloads and runs a script that opens a reverse shell in the system allowing a remote attacker to compromise the affected system.

lodahs, web3b, and web3-eht
Taking advantage of a typosquatting exploit for lodash npm packages, all versions of the "lodahs" package contain malware designed to find and exfiltrate cryptocurrency wallets. Web3b and web3-eht were removed for the same exploit pattern.

DEC 2019

Python3-dateutil and jellyfish
Two trojanized Python libraries were caught stealing SSH and GPG keys from the projects or infected developers.

JAN 2020

1337qq-js
The malicious npm package exfiltrates sensitive information such as hard-coded passwords or API access tokens through install scripts and targets UNIX systems only.

FEB 2020

Hundreds of RubyGems packages yanked from the public repository as a result of typosquatting concerns.

APR 2020

atlas-client
400 Gems were removed from the public repository for typosquatting and crypto mining malware. They include "atlas-client" (downloaded 2,100 times by developers).

MAY 2020

Octopus Scanner
26 open source packages were found to be compromised through malicious code injection. The malware was designed to enumerate and back door NetBeans projects through the NetBeans IDE.

AUG 2020

Multiple npm packages vulnerable to typosquatting attacks.
Electron, loadyaml, lodashs, and loadyml packages have all been identified as vulnerable. Once installed, the packages collect and expose sensitive information, including IP address, geolocation, and device fingerprint, publishing this data to a public GitHub page.

OCT 2020

Twilio-npm malware can be compromised through brandjacking.
The counterfeit package opens a backdoor on a user's device, giving attackers control of the compromised machine and Remote Code Execution (RCE) capabilities.

NOV 2020

Counterfeit components discovered in the npm ecosystem.
The series of components were identified as a successor to "fallguys" malware: *discord.dll, discord.app, wsbd.js*, and *ac-addon*. These components exfiltrate Discord and web browser's "leveldb" files, as well as collect data such as IP address, "PC username", "discordcanary" files, etc.

xpc.js" malware
Discovery of the malware in the npm registry confirmed to be a part of the newly identified family of Discord malware named CursedGrabber. This malware targets Window hosts and steals Discord information, sending user information via webhook to the attacker.

jdb.js and db-json.js
The malicious typosquatted packages are found laced with a popular Remote Access Trojan (RAT), njRAT aka Bladabindi. Upon install, the malicious script engaged in data gathering and reconnaissance, ultimately launching *patch.exe* which is an njRAT written in .NET. This allows a remote attacker to log keystrokes, modify registry values, initiate system shutdown or restart at will, among other nefarious activities.

DEC 2020

SolarWinds
Threat actors gained access to SolarWinds dev infrastructure, and injected malicious code into Orion update binaries. 18,000 customers automatically pulled trojanized updates, planting backdoors into their systems, and allowing bad actors to exploit private networks at will.

FEB 2021

Namespace Confusion
Three days after news broke of an ethical researcher hacking over 35 big tech firms in a novel supply-chain attack, more than 300 malicious copycat attacks were recorded. Within one month, more than 10,000 dependency confusion copycats had infiltrated npm and other ecosystems.

APR 2021

Codecov
An attacker was able to gain access to a credential via a mistake in how Codecov were building Docker images. This credential then let them modify Codecov's bash uploader script which was either used directly by customers or via Codecov's other uploaders like their Github Action. The attacker used this modified script to steal credentials from the CI environments of customers using it.

MAY 2021

Microsoft's WinGet
The weekend after launching, Winget's software registry was flooded with pull requests for apps that were either duplicates or malformed. Some newly added duplicate packages were corrupted and ended up overwriting the existing packages, raising serious concerns about the integrity of the Winget ecosystem.

JUL 2021

Kaseya
A ransomware group discovered and exploited a zero-day vulnerability in a remote monitoring and management software platform used by dozens of managed security providers (MSP). Because these MSPs service thousands of downstream customers, the hackers were able to conduct a ransomware attack against 1,500 victims.

Roughly, 18,000 customers automatically pulled these malicious updates, exposing the networks of large companies and government entities like the National Nuclear Security Administration and enabling the bad actors to explore and exploit their networks at will over the course of many months.

By attacking the SolarWinds software supply chain and mingling their malicious code with the legitimate, trusted code that was delivered to their clients, attackers were able to plant backdoors on the systems of tens of thousands of SolarWinds' customers.

Namespace Confusion — February 2021

In February 2021, news broke of a researcher, Alex Birsan, hacking over 35 big tech firms in a novel supply chain attack dubbed “dependency confusion.” The name of this attack refers to the inability of your development environment to distinguish between a private, internally-created dependency in your software build, and a package by the same name available in a public software repository.

In other words, should an attacker register the name of your private, internally-used dependency on a public repository, such as npmjs, your software development tool may inadvertently pull in the attacker's malicious dependency as opposed to your legitimate one.

Within 72 hours after news of the namespace attack vector became public, automated malware detection services observed 300+ copycats emerging from other researchers interested in earning a bug bounty. One week later, the number of copycat attacks increased to 575. The following week, it was 750. By March 15, 2021, Sonatype's automated malware detection service had observed more than 10,000 dependency confusion copycats having infiltrated npm and other ecosystems.

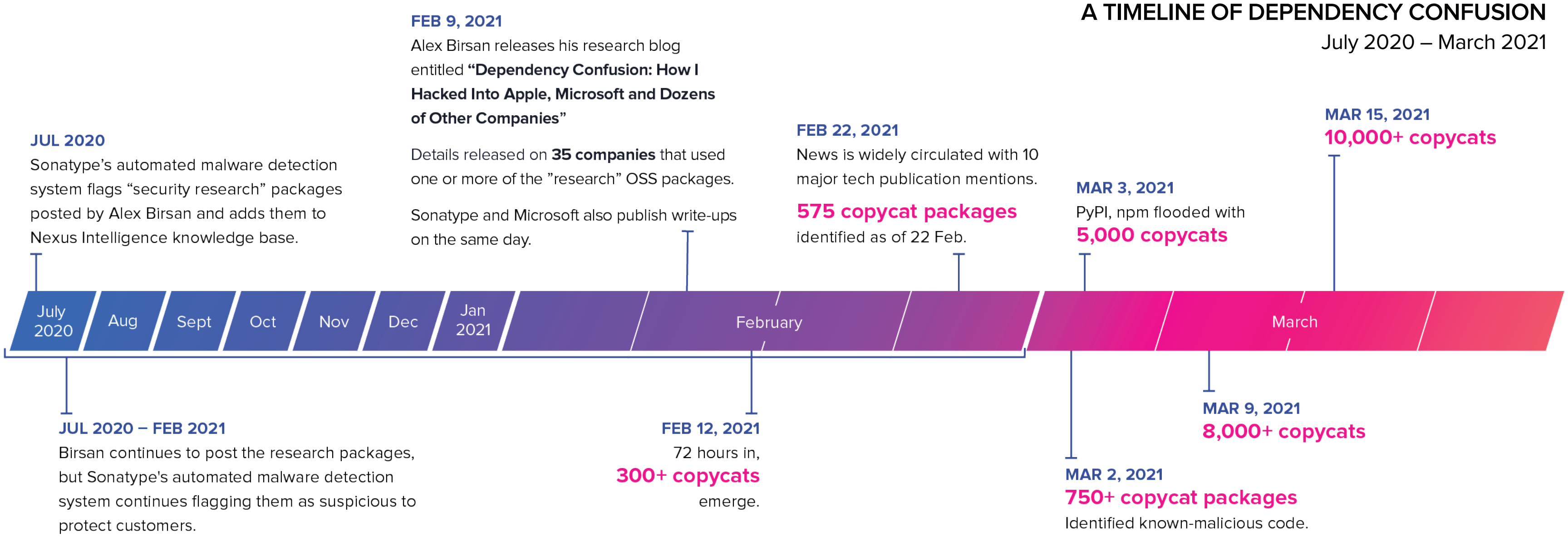
Not all copycats were benign proof of concepts. In search of bug bounty payouts, thousands were published by bad actors with malicious intent. Some of the copycats were even aimed as “vigilante vandalism” on the open source repositories.

“The fact that so much of the npm ecosystem is not namespaced has actually created potential build time malware injection possibilities. If I know

FIGURE 1.8

A TIMELINE OF DEPENDENCY CONFUSION

July 2020 – March 2021



JUL 2020
Sonatype's automated malware detection system flags “security research” packages posted by Alex Birsan and adds them to Nexus Intelligence knowledge base.

FEB 9, 2021
Alex Birsan releases his research blog entitled **“Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies”**

Details released on **35 companies** that used one or more of the ”research” OSS packages.

Sonatype and Microsoft also publish write-ups on the same day.

FEB 22, 2021
News is widely circulated with 10 major tech publication mentions.
575 copycat packages
identified as of 22 Feb.

MAR 3, 2021
PyPI, npm flooded with
5,000 copycats

MAR 15, 2021
10,000+ copycats

JUL 2020 – FEB 2021
Birsan continues to post the research packages, but Sonatype's automated malware detection system continues flagging them as suspicious to protect customers.

FEB 12, 2021
72 hours in,
300+ copycats
emerge.

MAR 9, 2021
8,000+ copycats

MAR 2, 2021
750+ copycat packages
Identified known-malicious code.

the name of a package in use by a company, I could go publish a malicious package using the exact same name with a new version number and know that it's very likely that it would be ingested over the intended, internally developed package," said Sonatype CTO Brian Fox **in 2017**.

Public repositories that do not strictly enforce namespace rules, including npm, PyPI, RubyGems, and NuGet, are susceptible to namespace confusion. In contrast, the Maven Central and Golang's pkg.go.dev repositories enforce strict namespacing and verify namespace ownership before artifacts can be published.

Codecov — April 2021

The Codecov supply chain attack publicized in April 2021 was similar to the SolarWinds attack. In this case, bad actors compromised a Codecov server to **inject their malicious code** into a Bash Uploader script that was then downloaded by Codecov's customers over the course of two months.

Using the Bash uploader script used by Codecov customers, the attackers exfiltrated sensitive information including keys, tokens, and credentials from those customers' Continuous Integration/ Continuous Delivery (CI/CD) environments. Using these harvested credentials, Codecov attackers **reportedly breached** hundreds of customer networks, including **HashiCorp**, **Twilio**, **Rapid7**, **Monday.com**, and e-commerce giant **Mercari**.

Although much focus has been on the compromised Bash Uploader script, the credentials used to modify the script were originally obtained by the attackers from a flawed Docker image creation process, **according to** Codecov. In aggregate, the incident highlighted the importance of securing CI/CD pipelines, including scrutinizing the secrets filed in these environments, and stepping up container security.

Microsoft's Winget — May 2021

In May 2021, Microsoft released the first stable version of its Windows 10 package manager, Winget, which enabled users to manage apps via the command-line. Users were able to propose or add new packages to Winget on the project's **GitHub** repository. But, over the weekend after its launch, many **flooded Winget's software registry** with pull requests for apps that were either duplicates or malformed. Moreover, some newly added duplicate packages were corrupted (incomplete) and ended up overwriting the existing packages. Over 60 such instances were seen. The incident raised serious **concerns** among developers about the integrity of the Winget ecosystem.

Kaseya — July 2021

In July 2021, the world witnessed another form of upstream software supply-chain attack. In this case, the REvil ransomware group aka Sodinokibi discovered and exploited a **zero-day vulnerability** in Kaseya's Virtual System Administrator (VSA). The VSA tool is a remote monitoring and management software platform used by dozens of managed security service providers who in turn service thousands of downstream customers.

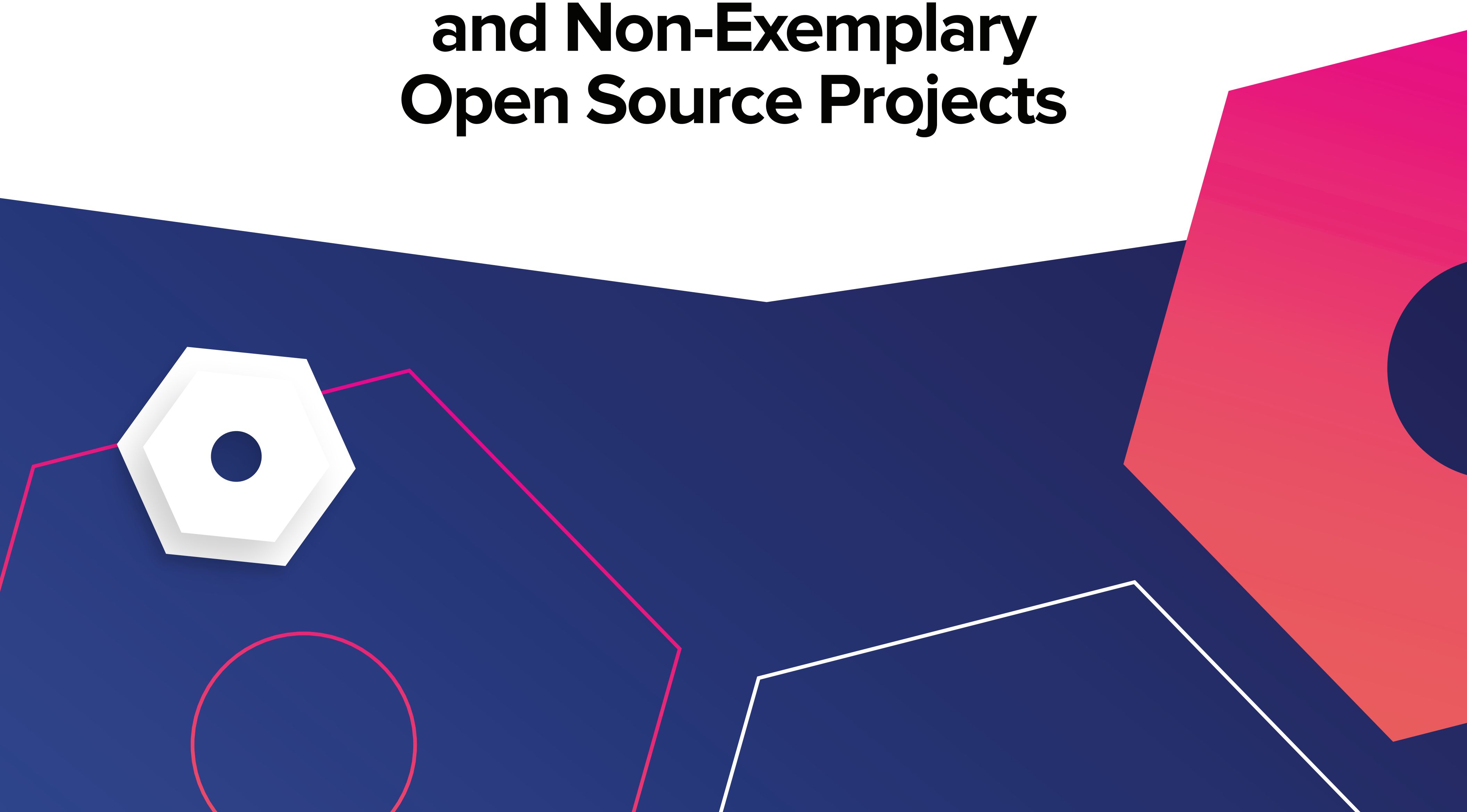
It didn't take long for the threat actors to follow up with a **$70 million ransom demand** to decrypt files for more than 1,500 victims. "This episode represents yet another incident in a long trend observed over many years: in order to scale exploitation of downstream victims, bad actors are increasingly targeting technology assets and providers that live upstream within the digital value stream. This includes open source libraries, IDEs, build servers, update servers, and, most recently in the case of Kaseya, Managed Service Providers (MSPs)," Sonatype's Matt Howard said in a **blog post** following the incident.

Although there are many tools designed to protect the perimeter of downstream technology assets, the truth of the matter is this: software itself is increasingly the soft underbelly of digital risk. If the past year is any indication, we expect that attackers will continue to target upstream software supply chain assets as a preferred path to exploiting downstream victims at scale.

This Kaseya incident quickly got the attention of US law enforcement authorities, including the **FBI and the Cybersecurity and Infrastructure Security Agency (CISA)**. It is a reminder for our industry and cyber defense teams to *shift security left* and focus on securing the upstream portion of the digital supply chain with the same energy and vigor that has long focused on the downstream portion.

CHAPTER 2

Understanding Exemplary and Non-Exemplary Open Source Projects



Given its prominence in modern software development, the quality of open source libraries used from third-party suppliers has a fundamental impact on digital innovation. But how should engineering leaders go about choosing the best open source projects? Which ones are optimal? Which ones are suboptimal? Further, how should engineering teams think about project hygiene, not only as it pertains to direct dependencies, but also transitive?

In this chapter we describe open source project quality metrics collected from the Maven Central ecosystem and compare them with recent quality metrics proposed by the Open Source Security Foundation (OpenSSF) and Libraries.io.

Our analysis reveals that MTTU, a measure of a project's dependency update velocity, is strongly associated with improved project security. We did not find OpenSSF Criticality or Libraries.io Sourcerank to be associated with improved project security.

Thus, in order to minimize risk associated with vulnerabilities in third-party open source libraries, **we recommend that software development teams adopt defined criteria for selecting open source projects. Further, we recommend that teams select projects that have low MTTU.**

Open Source Project Quality Metrics

Sonatype Mean Time to Update (MTTU)

MTTU is the average time required for a project to respond to new versions of its dependencies. Figure 2.1 shows how MTTU is calculated. Suppose we have a component A with dependencies B and C, both at version 1.2. Suppose B and C each release a new version (1.3) and some time later A releases a new version that bumps the version of B and C to 1.3. The time between the release of B version 1.3 and the release of A version 1.3 is the Time To Upgrade (TTU) for A's migration to B version 1.3 (and similarly for A's adoption of C version 1.3). The average of all these upgrade times is then the MTTU.

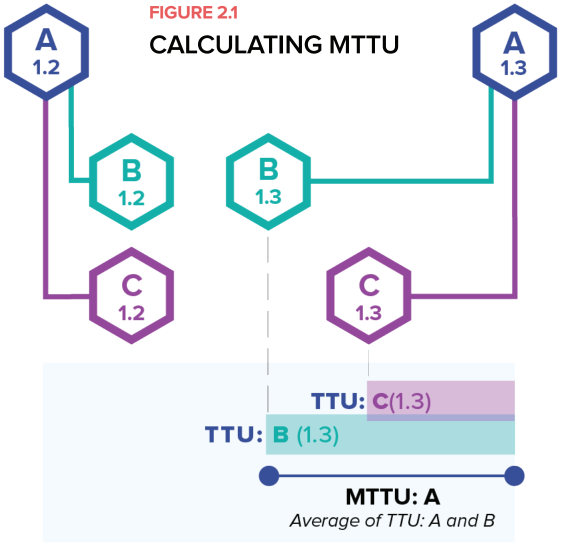


FIGURE 2.1

CALCULATING MTTU

MTTU provides visibility into an open source projects' dependency management practices — **Lower is better**. Projects that consistently react quickly to dependency upgrades in their downstream dependency chain will have low MTTU. Projects that either consistently react slowly or have high variance in their reaction time will have higher MTTU.

Figure 2.2 shows which percentage of components achieve various MTTU performance based on update data from 2020. Note that while the percentages climb quickly (26% upgrade within 20 days, 44% within 40, and 57% within 60), there is a long tail of slow-to-upgrade components, with 9% of components taking more than 180 days to upgrade.

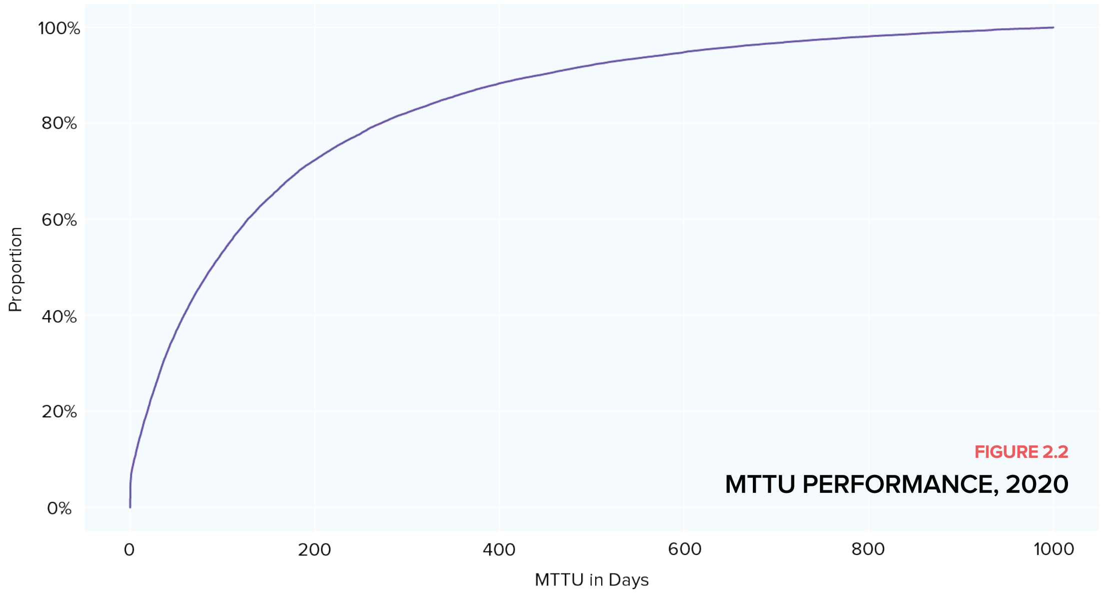


FIGURE 2.2

MTTU PERFORMANCE, 2020

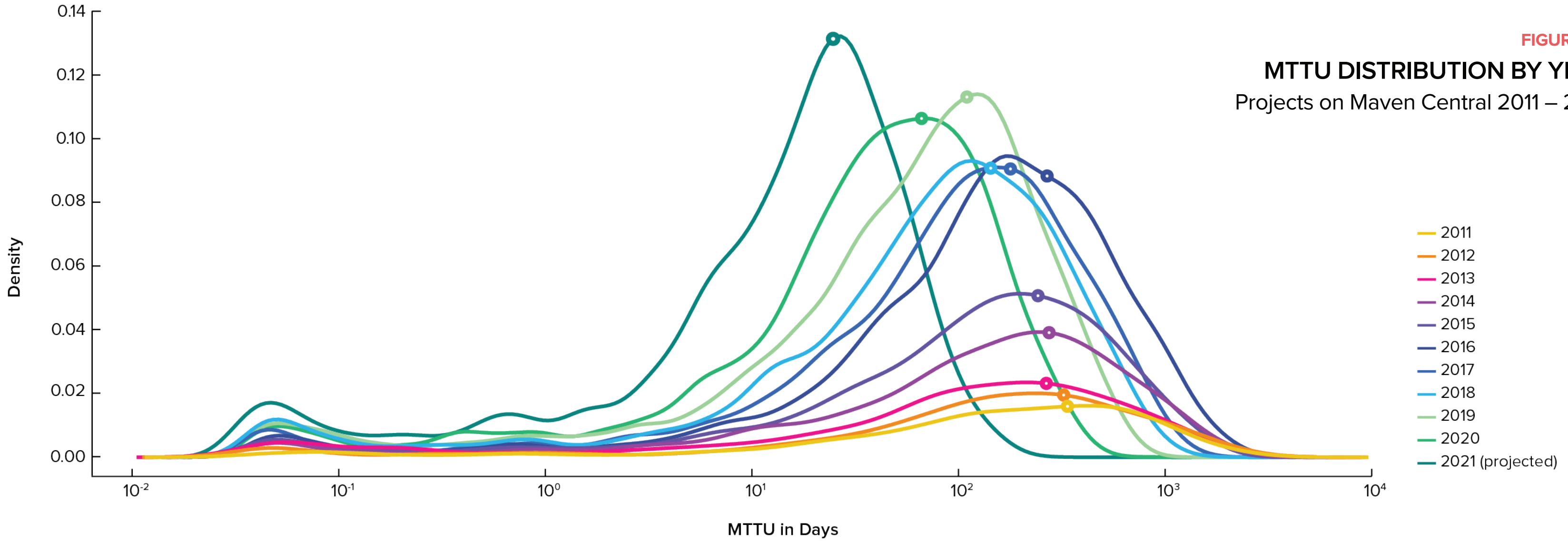


FIGURE 2.3

MTTU DISTRIBUTION BY YEAR

Projects on Maven Central 2011 – 2021

This slow update behavior has an even stronger impact as dependency chains grow. If a transitive dependency N levels deep releases an important security update, and each component on the dependency path takes D days to upgrade, then the top level application doesn't benefit from this fix for N × D days. Thus, even a chain of exemplary components that each upgrade within 20 days would result in a lag time of 100 days for a dependency five levels deep.

MTTU provides a measure of project quality that is based on how quickly the project moves to update dependencies. By this measure, quality has been increasing over the years.In Figure 2.3, we provide a graph of the distribution of project MTTU values by year for every year since 2011. We can see that in addition to the number of projects growing over the years, there has been a clear trend toward faster MTTUs as shown below.

- ▶ 2011 average MTTU = 371 days
- ▶ 2014 average MTTU = 302 days
- ▶ 2018 average MTTU = 158 days
- ▶ 2021 average MTTU (as of Aug 1) = 28 days[6]

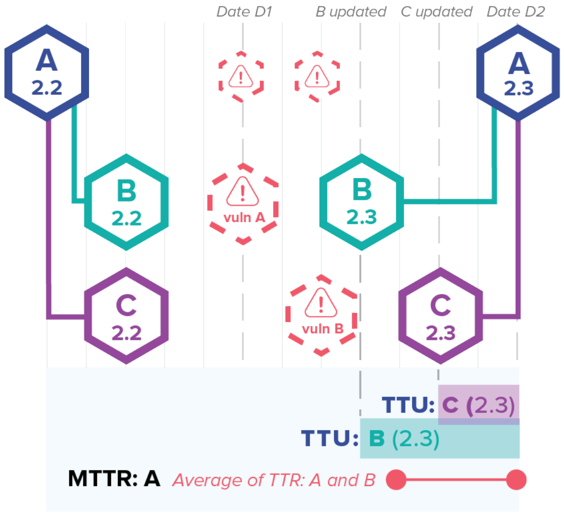
MTTU AND SECURITY

While MTTU does not directly measure responsiveness to security issues, our analysis in previous years has found that MTTU is correlated with mean time to remediate (MTTR), which is the time required to update dependencies that have published vulnerabilities, as shown in Figure 2.4. MTTR is defined just like MTTU, except that we take the average of those dependencies that were known to be vulnerable at the time of the update.

In our previous research[7], we found a significant correlation between MTTR and MTTU (Pearson correlation was 0.6 with N = 17,017). MTTR is generally only available for more popular and thus more scrutinized projects. Many projects fall below the level of usage required for security researchers to

FIGURE 2.4

CALCULATING MTTR



[6] Since 2021 has not yet ended, it is possible this number will change.

[7] 2019 State of the Software Supply Chain

perform vulnerability research and discover latent issues in the codebase. MTTU on the other hand is available for all projects, providing a common source of data for evaluating project quality. Thus, **we consider MTTU to be the best metric available to determine the impact a component will have on the security of projects that incorporate it**. Later in this chapter, we perform additional novel research to further confirm the value of MTTU.

Libraries.io Sourcerank

This metric aims to measure the quality of software, mostly focusing on project documentation, maturity, and community. It is computed by evaluating a number of yes / no responses to questions such as “Is the project more than six months old?” and a set of numerical questions, such as “How many ‘stars’ does the project have?” These are distilled into a single score, with yes / no questions adding or subtracting a fixed number of “points." Numerical values are then converted into points using a formula, e.g. “log(num_stars)/2." The current maximum number of points is approximately 30.

OpenSSF Criticality Score

OpenSSF has a set of analyses that combine into **a metric called “criticality."** Criticality measures a project's community, usage, and activity. This is distilled into a score that is intended to measure how crucial the project is in the open source ecosystem.

OpenSSF Scorecard

OpenSSF also has a more extensive evaluation of project quality called **the “Scorecard” project**. This project provides support for automatically monitoring development practices, tooling use, and other project quality and maturity attributes, then reporting which checks succeed and which fail. OpenSSF does not provide a mechanism for distilling this “Scorecard” into a single metric and so we did not include it in the quantitative analysis we describe below.

Quality Metric Comparison

Figure 2.5 summarizes the four proposed project quality frameworks by showing to what extent they incorporate information about five dimensions of quality: maturity, popularity, activity, dependency management, and development practices.

Popularity

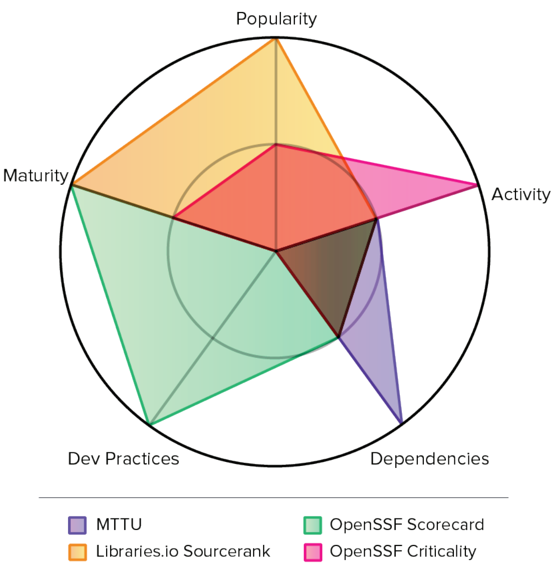
Libraries.IO includes project popularity metrics (stars, subscribers, and usage) as part of their Sourcerank metric. OpenSSF’s criticality metric includes usage (the number of projects that use the library) but not stars or subscribers. OpenSSF’s Scorecard system and MTTU do not include any factors that are related to popularity.

Activity

All four quality frameworks include some aspect of activity analysis. Sonatype’s MTTU metric is lightly correlated with activity because fast MTTU requires frequent releases. Libraries.IO Sourcerank tracks whether a project has been updated in the last six months, another weak correlation with activity. OpenSSF’s Scorecard metric includes a check (whether there has been a commit in the last 90 days) that is also weakly correlated with activity. The OpenSSF Criticality metric includes a robust set of activity measures such as commit frequency and release count.

FIGURE 2.5

OPEN SOURCE QUALITY METRIC COMPARISON



Dependencies

Sonatype’s MTTU provides the most robust measure of dependency update practices, as it measures how quickly a project updates its dependencies once new versions are released. Libraries.IO Sourcerank checks whether there were outdated dependencies at the time of scoring. OpenSSF Scorecard checks if automated dependency update tools are used. OpenSSF Criticality does not consider dependency management practices.

Dev Practices

The OpenSSF Scorecard is the only measure that considers development practices such as whether a code review is performed, and whether continuous integration and Static Analysis and Security Testing (SAST) tooling is used.

Maturity

Libraries.IO and OpenSSF metrics include measures of maturity. Libraries.IO Sourcerank includes semantic versioning checks and a number of documentation checks. OpenSSF Scorecard includes similar versioning and documentation checks, as well as checks regarding how the project is packaged and distributed. The OpenSSF Criticality metric includes checks that consider how quickly issues are resolved, responsiveness of contributors, and when the project was originally created.

As Figure 2.5 shows, no single metric provides robust coverage across all five attributes. However, by combining metrics, we can determine which projects score highly across popularity, activity, dependencies, dev practices, and maturity metrics. In the following section, we evaluate the individual metrics, as well as these combined metrics to determine which signals are most important in identifying high-quality projects that enable faster innovation with less risk.

Research Findings

When using open source components, there are a number of outcomes that developers would rather avoid. Components can contain vulnerabilities, which their applications then inherit. Worse, components can be vulnerable without having an available remediation path (e.g. “no path forward,” described later in this chapter), requiring significant effort to refactor or deprecate the component. Component upgrades can also break application builds, requiring even more development work. To determine which components exhibit higher and lower quality traits, we analyzed 100,000 applications. We looked at the individual quality metrics to identify which measures are most useful when choosing components.

All told, we obtained a collection of 39,164 open source components that were used across these 100,000 applications. We were able to obtain MTTU data for 52% of components, Sourcerank scores for 91% of components, and Criticality scores for 40% of these components.

The data set contained 233,569 component versions. For each version, we evaluated whether it was subject to any of the following conditions:

- **Vulnerable:** A component is *vulnerable* if it or any of its transitive dependencies contain a known security issue. Overall, 5,175 vulnerable component versions were found.
- **No Path Forward:** A component has *no path forward* (NPF) if the latest version of that component is vulnerable. This means the vulnerability cannot be remediated by upgrading the component. For direct dependencies, this occurs if a project is slow to remediate security issues in its codebase. For transitive dependencies, this occurs when a project is slow to update its own vulnerable dependencies (or these dependencies are slow to fix issues and update their dependencies, etc.). Overall, 788 component versions had no path forward at the time they were found to be vulnerable.
- **Breaking Changes:** A component has *breaking changes* if an update to that component changes public APIs in a manner that would cause either builds to break or runtime errors. Overall, 1,116 component versions had breaking changes, affecting 349 projects.

FIGURE 2.6

METRICS USED TO ASSESS RELATIVE QUALITY OF AN OSS PROJECT

	SONATYPE MTTU *Faster Updates*	OPENSSF CRITICALITY *Higher Score*	LIBRARIES.IO SOURCERANK *Higher Score*	POPULARITY *More Popular*
Vulnerable	**1.8x less likely**	–	**2.9x more likely**	**2.8x more likely**
No Path Forward	**∞ (no NPF in top 25%)**	6x less likely	–	**3x more likely**
Breaking Changes	**3.2x less likely**	**8x less likely**	**3.3x less likely**	**12x less likely**

XX component is less likely to contain defects XX component is more likely to contain defects

Exemplars

We consider exemplary projects to be those that are in the top 25% according to the metrics of interest. We then check to see whether exemplary projects are less likely to have the undesirable outcomes described above when compared to the bottom 25% of the population. In other words: if you select a top-rated project as compared to a bottom-rated project, what difference in outcomes would you expect? We also include a comparison with a selection method based purely on popularity, as measured by the number of times a component occurs in application scans.

Figure 2.6 summarizes, for each type of exemplar, and each outcome type, whether we found a statistically significant difference in the likelihood of negative outcomes, and the factor difference observed. Bolded entries are statistically significant at a level of $p < 0.005$, while non-bolded entries have $p < 0.05$. That is: our confidence that we’re seeing a true effect is higher for the bolded entries.

We color entries red when the effect is in the *negative direction*, for example if a high-quality component is **more** likely to contain a vulnerability.

In summary, Sonatype MTTU is the best metric for identifying open source projects that are less likely to contain vulnerabilities. Popularity and Sourcerank (which includes popularity measures) are poor metrics to consider if the goal is strictly to avoid vulnerable versions, as more popular projects tend to have more reported vulnerabilities, as discussed in Chapter 1.

SECURITY

As in previous studies, we see a strong correlation between better update hygiene, as measured by MTTU, and security, as measured by lower rates of "vulnerabilities" and "No Path Forward" conditions. Components with faster MTTU were 1.8 times less likely to have vulnerabilities when compared to the bottom 25% and much less likely to be stuck in a vulnerable state due to having no path forward.

MTTU AND TRANSITIVE DEPENDENCIES

Upgrade responsiveness is particularly important when it comes to resolving security issues with transitive dependencies. When incorporating open source components into an application, that program inherits not only the direct code quality and security practices of the component, but also its approach to dependency management.

New versions of dependencies typically bring bug fixes, security patches, new functionality, and performance improvements. Ideally, each library in the dependency tree for a component would swiftly upgrade their direct dependencies, thus ensuring that these new versions make their way swiftly throughout the dependency tree. MTTU measures exactly the extent to which this happens.

POPULARITY

In previous reports, we showed that Maven Central download popularity was a poor predictor of dependency management quality. This year, we examined popularity as measured by the number of times a component occurs in application scans. We again found the same results: popularity is not a good predictor of security. In fact, in this year's result, popularity was misleading, with more popular projects more likely to contain vulnerabilities and be stuck in no path forward states. Libraries.IO Sourcerank had a similar association with vulnerability, which is not surprising as Sourcerank includes multiple attributes focused on project popularity.

Guidance for Open Source Project Owners and Contributors

While there are plenty of projects that obtain good outcomes without following all the practices considered by the project quality metrics we discussed here, there is strong evidence that these practices lead directly to improved security and quality outcomes. We therefore recommend that project maintainers strive to adopt the best practices measured by Sonatype's MTTU and OpenSSF's Criticality metrics.

While the factors in the OpenSSF Scorecard may also improve project quality, we were not able to empirically evaluate this possibility because the OpenSSF Scorecard does not provide an associated quantitative metric.

Guidance for Enterprise Development Teams

Choosing high-quality open source projects is an important strategic decision for enterprise software development organizations. Components exhibit a wide variety of outcomes in terms of release velocity, security remediation behavior, and likelihood of breaking changes. Chapter 3 of this paper details the impact this can have on development efficiency of projects using these components. Therefore, we recommend choosing components with low MTTU values and high Criticality scores.

While Libraries.IO Sourcerank wasn't associated with higher performance in the outcomes we considered, it may well promote other desirable effects and does contain sensible practices. There is little reason not to prefer applications with higher Libraries.IO Sourcerank if they have high Criticality scores and low MTTU.

Just as traditional manufacturing supply chains intentionally select parts from approved suppliers and rely upon formalized procurement practices — enterprise development teams should adopt similar criteria for their selection of open source components. This practice ensures the highest quality parts are selected from the best and fewest suppliers — a practice W. Edwards Deming recommended for decades to manufacturers of physical goods. Implementing selection criteria and update practices will not only improve code quality, but can accelerate mean time to repair when suppliers discover new defects or vulnerabilities.

CHAPTER 3

Peer Practices Associated With Micro and Macro Dependency Management

In today's world, a wide variety of bits are flowing rapidly through our software supply chains. Because of this, an equally wide variety of decisions must be made by engineering team members across every phase of the DevSecOps value stream. What you will see in the following research is that software developers make suboptimal choices **69% of the time** with respect to updating third-party dependencies.

In particular, there are two types of decisions that have become increasingly critical to maintaining healthy software supply chains:

1. **Micro dependency decisions:** frequent tactical decisions that developers must make on a daily basis to determine whether or not to update existing dependencies when newer versions become available.
2. **Macro architectural decisions:** strategic decisions that software architects and engineering leaders must make when deciding which open source projects are optimal for their products and why.

> Software developers make suboptimal choices **69% of the time** with respect to updating third-party dependencies.

But how should these decisions be made at scale?

- ▶ Should companies expect software developers to intuitively know the right action to take?
- ▶ What are the benefits of making good decisions?
- ▶ What are the costs of making bad decisions?
- ▶ Do engineering leaders have a responsibility to equip developers with information designed to automate better decision making?

These are a few of the questions that we attempt to answer in Chapter 3.

To Update or Not: An Empirical View of Micro Dependency Management

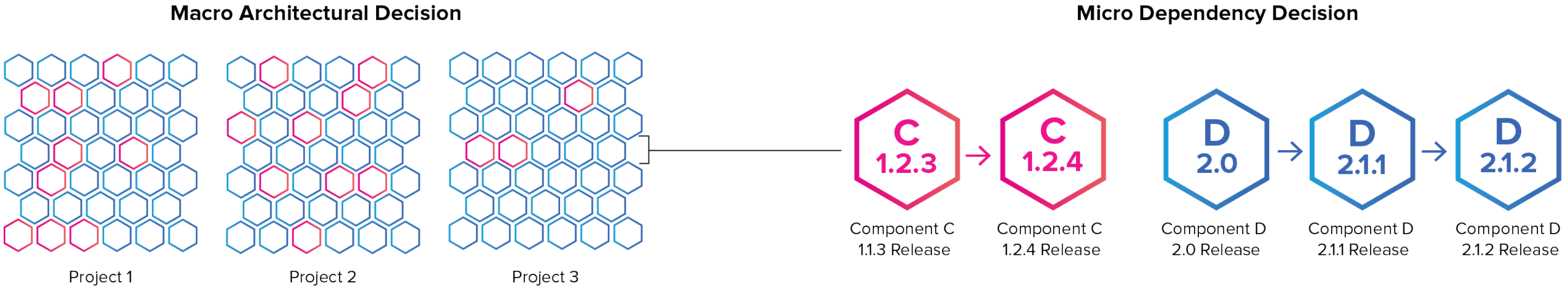
There are three reasons why dependency management is rapidly becoming an increasingly important practice for software engineering teams:

1. The enormous volume of open source dependencies present in production applications.
2. The incredible velocity at which new versions of dependencies are being released.
3. The fact that open source dependencies age like milk, and not like wine.

The average modern application contains 128 open source dependencies, and the average open source project releases 10 times per year[8]. This reality, combined with the fact that a few hyper-active projects release more than 8,000 times per year, creates a situation in which developers must constantly decide

FIGURE 3.1

MACRO ARCHITECTURAL VS. MICRO DEPENDENCY DECISIONS


Macro Architectural Decision
Project 1
Project 2
Project 3
Micro Dependency Decision
C 1.2.3
C 1.2.4
D 2.0
D 2.1.1
D 2.1.2
Component C 1.1.3 Release
Component C 1.2.4 Release
Component D 2.0 Release
Component D 2.1.1 Release
Component D 2.1.2 Release

Software architects and engineering leaders decide which open source projects to use.

Developers decide which dependencies to update, when, and what version to update to.

[8] 2019 State of the Software Supply Chain

when (and when not to) update third-party dependencies inside of their applications.

But, ask most developers about dependency management, and they'll tell you the same thing: keeping open source libraries fresh and optimally up to date is a good idea that requires terribly mundane work. In other words, in the eyes of most software engineers, dependency management is seen as a thankless maintenance task that's easy to get wrong, hard to get right, and generally detracts from time spent innovating. Developers know that it's important, but they frequently don't have the time or patience to make it a priority and lack the tooling to do it optimally. It's no wonder that many developers describe this situation as "dependency hell."

The result is that dependencies in applications can easily grow old and stale (vulnerable) despite the possibility that newer and fresher (more secure) versions are readily available.

In light of these circumstances, Sonatype researchers set out to answer the question: *are developers making efficient dependency management decisions?*

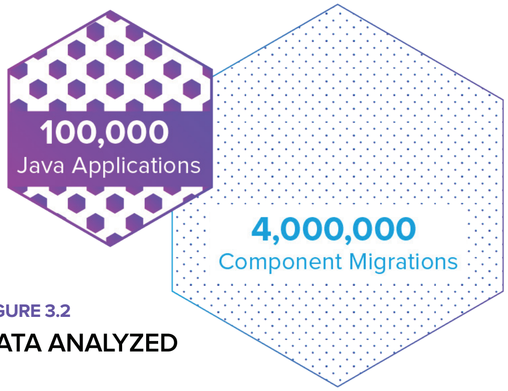
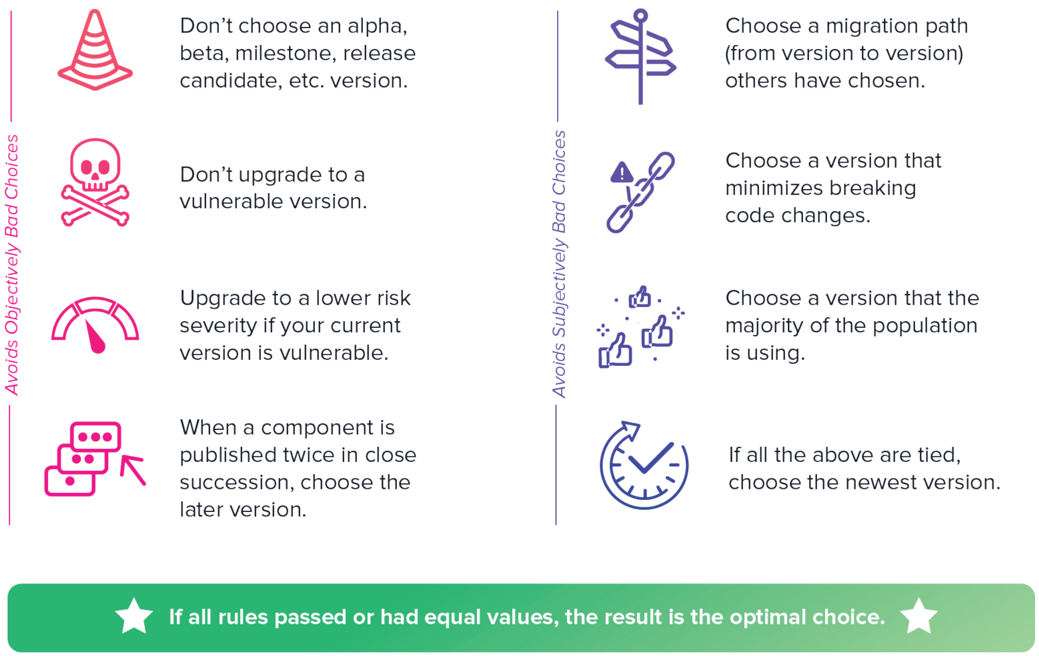


FIGURE 3.2

DATA ANALYZED

FIGURE 3.3

8 RULES FOR UPGRADING TO THE OPTIMAL VERSION


Avoids Objectively Bad Choices
Avoids Subjectively Bad Choices

Don't choose an alpha, beta, milestone, release candidate, etc. version.

Don't upgrade to a vulnerable version.

Upgrade to a lower risk severity if your current version is vulnerable.

When a component is published twice in close succession, choose the later version.

Choose a migration path (from version to version) others have chosen.

Choose a version that minimizes breaking code changes.

Choose a version that the majority of the population is using.

If all the above are tied, choose the newest version.

If all rules passed or had equal values, the result is the optimal choice.

To understand the relative quality of current dependency management decisions, Sonatype researchers spent the past year studying 100,000 Java applications[9] and analyzing more than four million component migrations (upgrades from version n to any number of potential newer versions).

In support of our research, we developed a scoring algorithm (Figure 3.3) designed to measure the relative quality of component migration decisions. The "component choice" algorithm is derived from eight common-sense rules distilled from the insights in the previous chapter.

Research results:

1. 10% of the projects in the Maven Central ecosystem are being used in production apps — and only 25% of those are actively being updated, which itself is a massive and complex effort.
2. Upgrade decisions are highly variable and frequently suboptimal, yet herd behavior doesn't lie.
3. Newer versions are generally better, but not always best.

[9] 100,000 anonymized, validated applications scanned by publicly available and commercial vulnerability analysis tools.

FINDING #1: Dependency management is a massive effort practiced on only 25% of libraries in production apps.

With 430,000 projects to choose from in Maven Central, it's remarkable that 100,000 Enterprise Java applications are leveraging only 40,000 (10%) of available projects. Furthermore, of the 40,000 projects being leveraged, only 10,000 (25%) are actively being managed and updated by the downstream software developers using these projects (Figure 3.4). Notwithstanding, it is staggering to see these 10,000 projects were updated more than 4 million times across 105,170 versions, with an estimated effort of one hour per update.

Needless to say, when it comes to dependency management, the level of developer effort and scale of developer decision making is massive, even though it only pertains to 25% of utilized dependencies.

Our analysis of 100,000 applications revealed that 75% of components in use were not upgraded in the last year. Why is that? Is it because engineers are indifferent? Is it because they are afraid of breaking builds? Or, is it simply because developers lack structured guidance at their fingertips?

Regardless of the reason, a majority of dependencies are simply not being updated in a regular manner. This is a missed opportunity for engineering teams to improve quality, minimize risk, reduce unplanned work, and save money by proactively managing 100% of their dependencies.

FINDING #2: Upgrades are variable and suboptimal, and herd behavior doesn't lie.

Upgrades are variable and suboptimal.

Our research revealed that organizations perform an average of 6,200 component migrations per year, and that 69% of the target migration choices made were suboptimal because they failed to identify the best version to upgrade to.

Migration decisions were divided into five groups, as outlined in Figure 3.5.

Whenever a developer updates a dependency, they have on average 21 available versions to choose from. Without intelligent automation to

FIGURE 3.4

ACTIVE PROJECTS IN THE MAVEN CENTRAL REPOSITORY

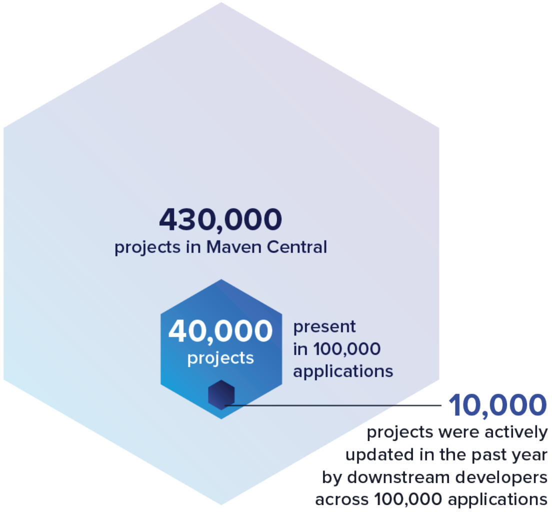


FIGURE 3.5

5 GROUPS OF MIGRATION DECISIONS

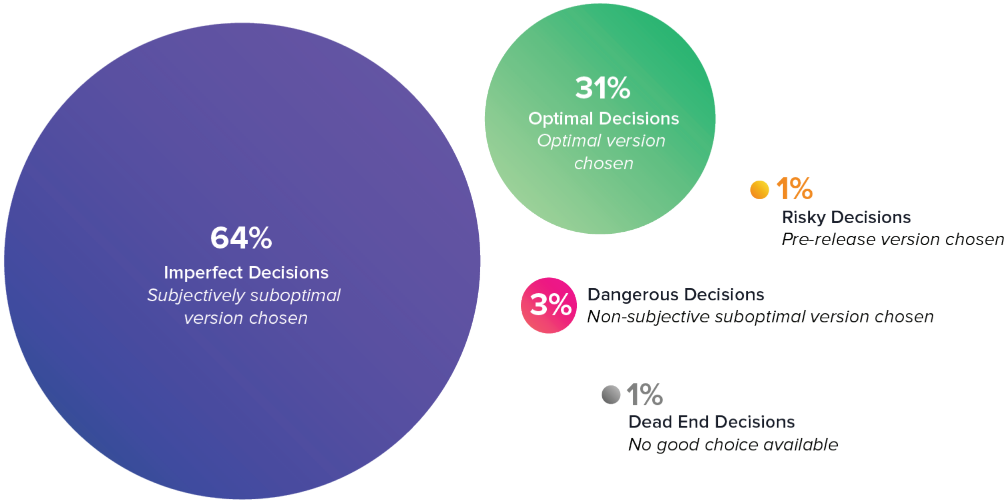
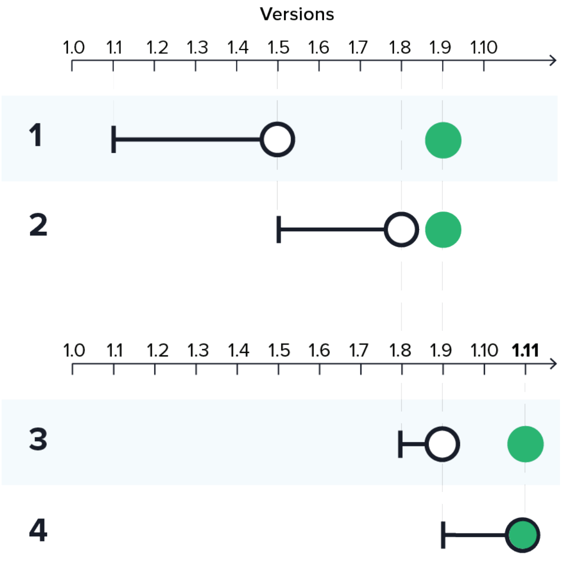


FIGURE 3.6

UNDERSTANDING IMPERFECT UPGRADE CHOICE



support them, and with so many versions to choose from, it is inevitable that developers will guess incorrectly which version is best.

Understanding "imperfect choices". To understand the definition of an imperfect upgrade choice, consider the following scenario:

Component *foo* was upgraded four times over the course of a year. At the time of each upgrade, the developer should reasonably consider all available versions and take into account multiple dimensions of data including: security, quality, popularity, and licensing, as set out in Chapter 2.

In Figure 3.6, upgrades 1 and 2 occurred when the optimal version was 1.9 according to the "component choice" rules, as described on page 23.

Upgrades 3 and 4 occurred after a new component version became available which made 1.11 the new optimal version.

In the aggregate, upgrades 1 and 3 were suboptimal, resulting in unnecessary upgrades 2 and 4.

The cost of performing suboptimal upgrades to a single component, for a single team, for a single application is small. However, when considering the fact that only 31% of upgrade decisions examined in this study were optimal, it is easy to see how much time and effort developers could save by consistently making better upgrade decisions. Specifically, in our sampling of 100,000 applications and four million update decisions, we discovered that 69% of such decisions were suboptimal.

Equipped with intelligent automation, a medium sized enterprise with 20 application development teams would save a total of 160 developer days (1,280 hours) and $192,000 per year at a fully loaded cost of $150 per hour. This would give each development team almost two weeks of extra productivity time — each year.

FIGURE 3.7

TIME SAVED WITH INTELLIGENT AUTOMATION

By making optimal upgrade decisions, organizations could save

8

days per development team per year.

A medium sized enterprise with **20 application development teams**

would **save 160 developer days** (1,280 hours)

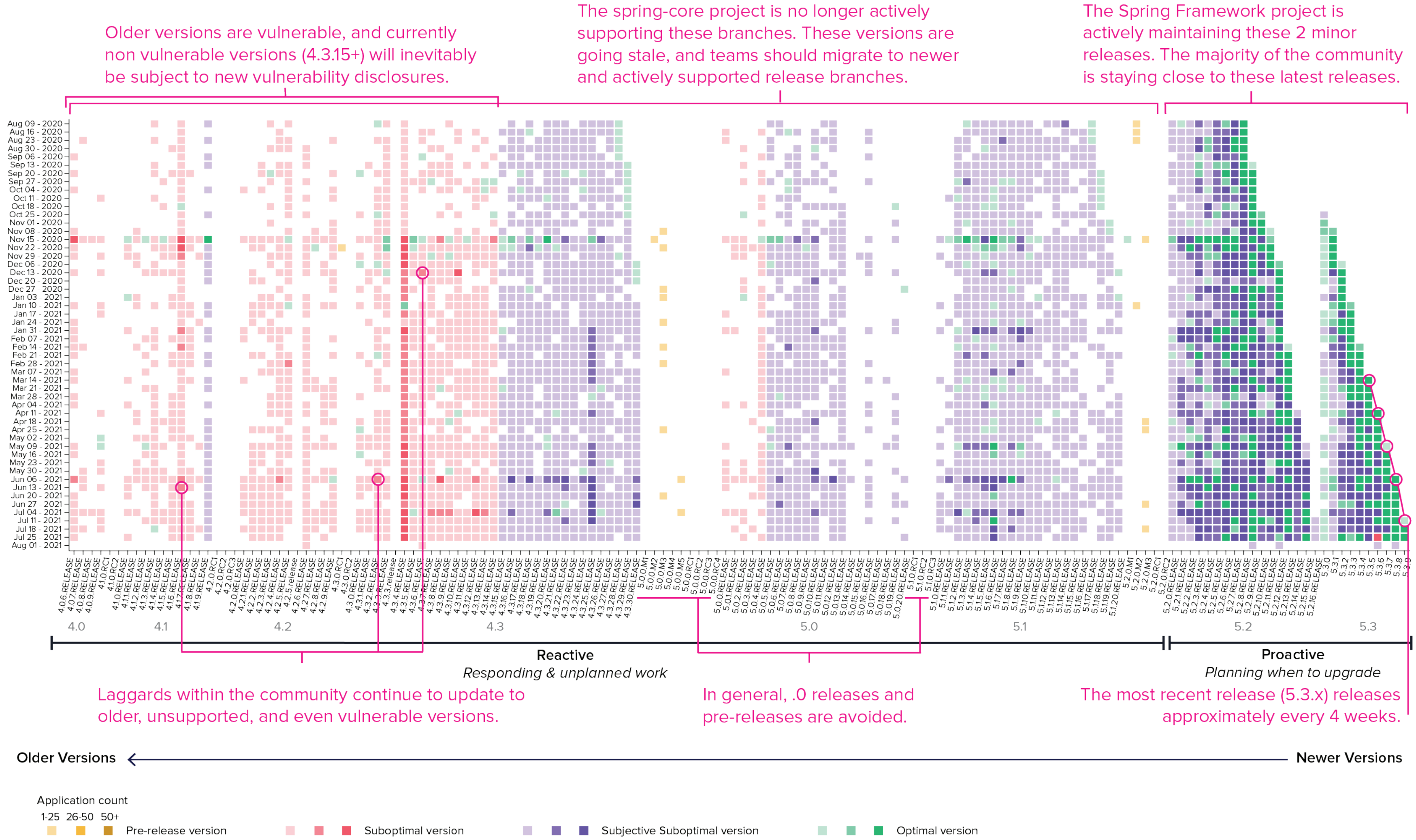
and **$192,000** per year

at a fully loaded cost of $150 per hour.

FIGURE 3.8

HERD MIGRATION BEHAVIOR OF ORG.SPRINGFRAMEWORK:SPRING-CORE

August 9, 2020–August 1, 2021



Older versions are vulnerable, and currently non vulnerable versions (4.3.15+) will inevitably be subject to new vulnerability disclosures.

The spring-core project is no longer actively supporting these branches. These versions are going stale, and teams should migrate to newer and actively supported release branches.

The Spring Framework project is actively maintaining these 2 minor releases. The majority of the community is staying close to these latest releases.

Laggards within the community continue to update to older, unsupported, and even vulnerable versions.

In general, .0 releases and pre-releases are avoided.

The most recent release (5.3.x) releases approximately every 4 weeks.

Herd behavior doesn't lie.
By examining 100,000 applications and four million migration decisions made by the developer herd, it is easy to visualize patterns and practices associated with both efficient and inefficient dependency management behaviors.

On the previous page, Figure 3.8 provides a visual summary of herd migration behavior over the past year associated with spring-core, a single component within the highly popular spring-framework. The y-axis shows the past 52 weeks of upgrade activity, with the top row representing herd migration decisions made one year ago, and the bottom row representing herd migration decisions made during the most recent week. The x-axis represents the 150 most recent versions with older versions to the left, and newer versions to the right.

FINDING #3:
New versions are not necessarily better.

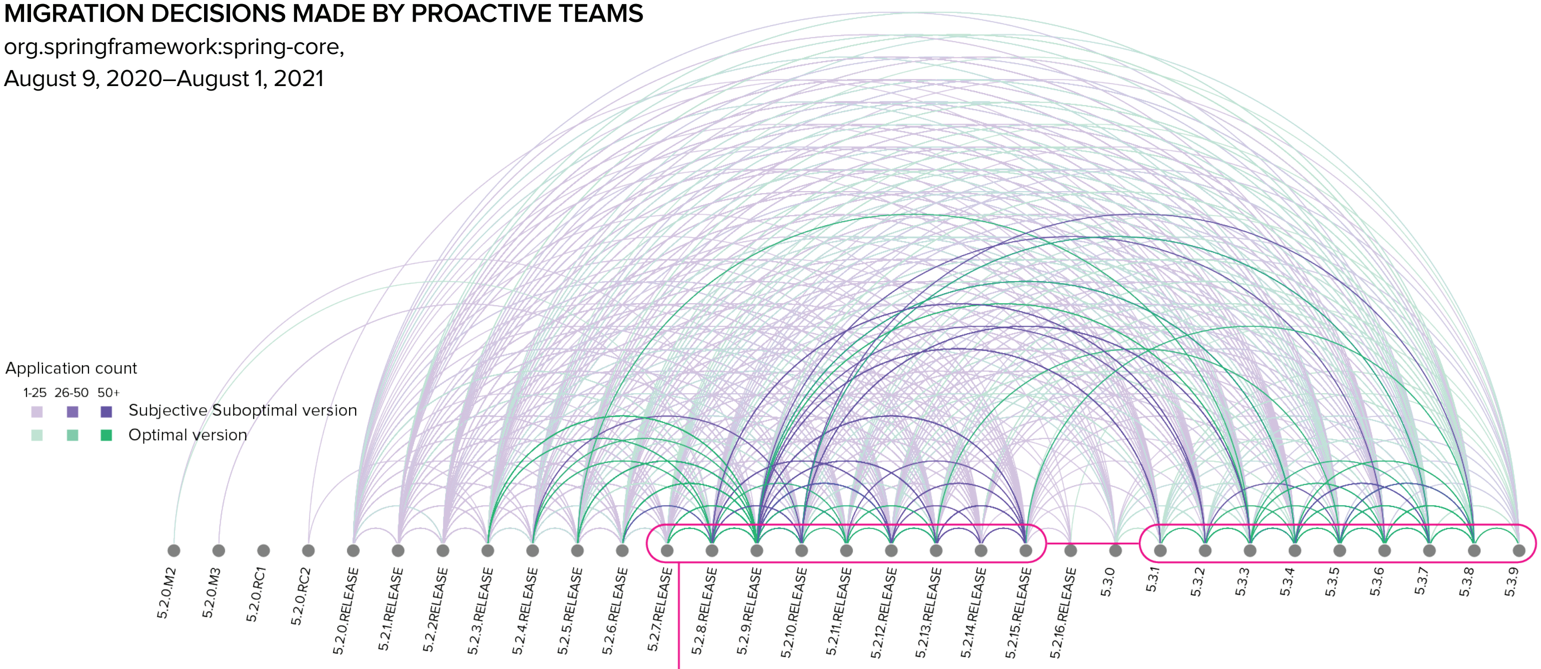
In an attempt to help automate dependency management decision making, some package managers provide for an open-ended version range that pulls the latest version as soon as it becomes available. In the spirit of keeping things fresh, many tools myopically submit pull requests for every new release. While such updates happen automatically, they can also have unintended consequences like the introduction of unplanned work and unnecessary security risk – e.g. malware injection and namespace confusion (Dependabot, Renabot, etc.). This type of naive dependency update strategy can lead to frustration and distraction for project maintainers, as described by **Dan Abranov's recent blog**.

To contextually automate dependency management, more intelligent tools are emerging that minimize both upgrade risk and upgrade events, thereby maximizing overall efficiency. Using the upgrade rules defined in Figure 3.3, we find there is a correlation between optimal choice and the latest version. Score 9 is assigned to the optimal

FIGURE 3.9

MIGRATION DECISIONS MADE BY PROACTIVE TEAMS

org.springframework:spring-core,
August 9, 2020–August 1, 2021



Many teams are living "on the edge" or "near the edge" and regularly updating when new versions become available.

FIGURE 3.10

DEFINING X FOR LIVING CLOSE TO THE EDGE

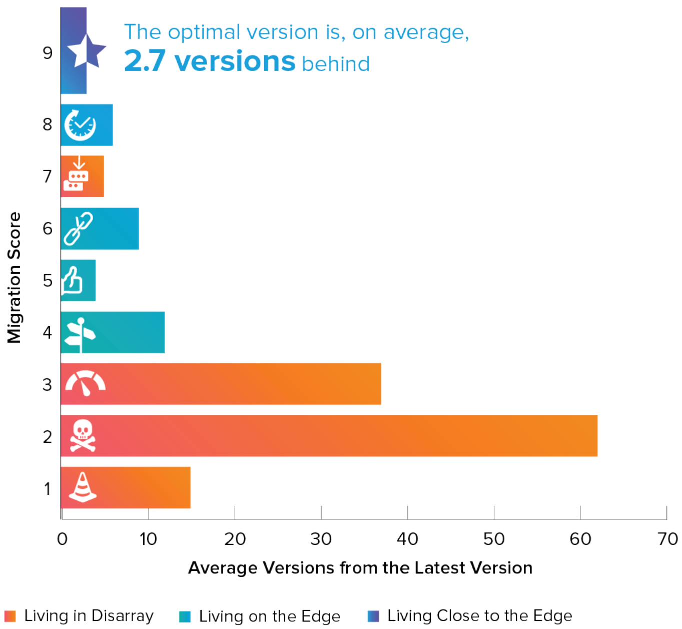
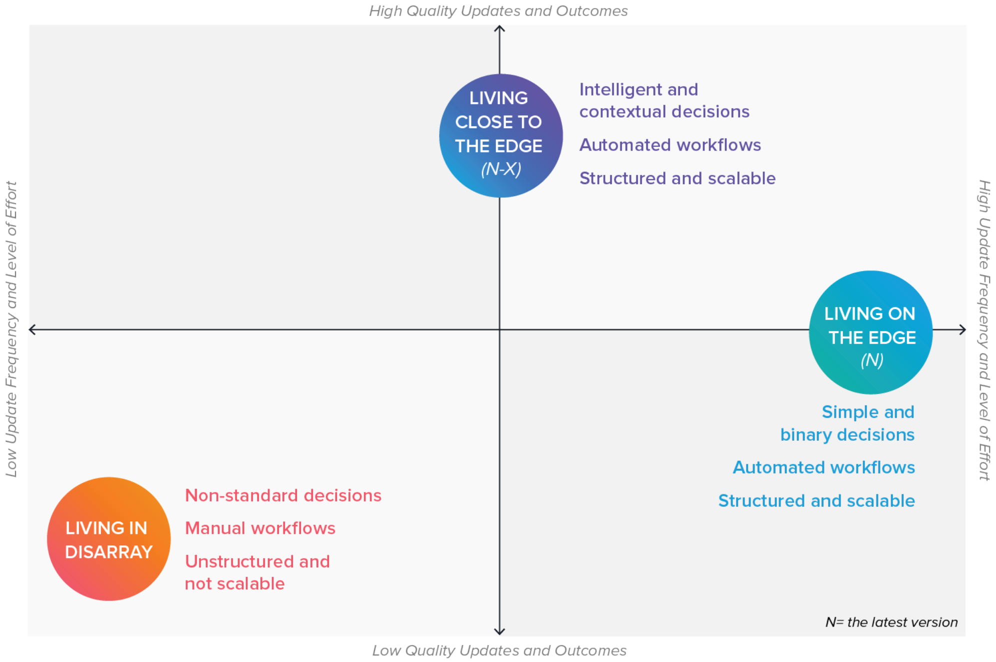


FIGURE 3.11

STRATEGIES FOR DEPENDENCY MANAGEMENT


High Quality Updates and Outcomes
Low Quality Updates and Outcomes
Low Update Frequency and Level of Effort
High Update Frequency and Level of Effort
LIVING CLOSE TO THE EDGE (N-X)
Intelligent and contextual decisions
Automated workflows
Structured and scalable
LIVING ON THE EDGE (N)
Simple and binary decisions
Automated workflows
Structured and scalable
LIVING IN DISARRAY
Non-standard decisions
Manual workflows
Unstructured and not scalable
N= the latest version

choice. On average, the most optimal target is 2.7 versions from the LATEST version, demonstrating that the most recent release is not the best choice, and highlighting why simplistic update policies are insufficient.

FINDING #4: Three distinct patterns of dependency management behavior.

Our analysis reveals that development teams exhibit three distinct patterns of dependency management behavior:

Teams Living in Disarray

Developers working on these teams lack automated guidance. They update dependencies infrequently. When they do update, they utilize gut instincts and commonly make suboptimal decisions. This approach to dependency management is highly reactive, wasteful, not scalable, and leads to stale software with elevated technical debt and increased security risk.

Teams Living on the Edge

Developers working on these teams benefit from simplistic but non-contextual automation. Dependencies are automatically updated to the latest version, whether optimal or not. Such automation helps to keep software fresh, but it can inadvertently lead to increased security risks and higher costs associated with unnecessary updates and broken builds. This approach is proactive and scalable, but not optimal in terms of expense or outcomes.

Teams Living Close to the Edge

Developers working on these teams have the advantage of intelligent and contextual automation. Dependencies are automatically recommended for updating, but only when optimal. This type of intelligent automation keeps software fresh without inadvertently introducing wasted effort or increased security risk. This approach is proactive, scalable, and optimal in terms of cost efficiency and quality outcomes.

Selecting the Best Projects: Reflections on Macro Architectural Decisions

When developing applications in the context of a modern software supply chain, it is critical that engineering leaders define clear policies. These help their developers make sound architectural decisions as to which open source projects are acceptable (optimal exemplars), and which ones are unacceptable (suboptimal non-exemplars).

Establishing and enforcing intelligent architectural policy is important for several reasons:

1. Save time and money by standardizing which projects are best for you, and eliminating inconsistent diligence efforts associated with component selection.
2. Improve application security and quality by standardizing on projects that are most likely to provide reliable access to new versions of non-vulnerable dependencies. This aids with micro dependency decisions and helps with never getting stuck with “no path forward” traps.
3. Reduce technology bloat associated with non-standard decision making.

If developers made dependency update decisions based on a structured system of guidance, we would expect to see a correlation between optimal update decisions and exemplary projects, as well as suboptimal update decisions and non-exemplary projects. The fact that these correlations DO NOT EXIST, reveals a clear opportunity for engineering leaders to benefit by standardizing open source architectural guidance at scale.

Conclusions and Practical Recommendations

- ▶ Based on the research, it’s clear that material inefficiencies exist along with significant avoidable risk. Software engineering teams have considerable room for improvement with respect to dependency management practices.
- ▶ To improve efficiencies, save money, and optimize dependency management at scale, engineering leaders should embrace intelligent automation. Chosen tools should remove the current error-prone **micro decision** making from day-to-day developer workflows.
- ▶ Engineering leaders should also embrace tools to guide **macro decisions** made by architects and developers with respect to initial technology selection.

CHAPTER 4

Software Supply Chain Maturity

For this year's report, we surveyed 702 engineering professionals about their software supply chain management practices, including approaches and philosophies to utilizing open source components, organizational design, governance, approval processes, and tooling. The survey also inquired about engineering outcomes including deployment frequency, security, engineering productivity, and job satisfaction. The responses came from IT professionals representing a variety of roles and industries.

The objective of the survey was twofold:

1. Determine if certain software supply chain practices correlate to successful engineering outcomes.
2. Develop a benchmark and maturity model so organizations can evaluate themselves in comparison to their peers.

The survey itself consisted of 41 questions:

- Ten questions were focused on understanding the relative quality of software outcomes (dependent variables).
- 24 questions were focused on understanding patterns and practices embraced by engineering teams (independent variables).
- Seven questions were focused on understanding job satisfaction.

Responses to all 41 questions were assessed against the following eight elements of software supply chain management practices:

1. **Application inventory** (Inventory) – Do you know all the applications your organization has in development/production, and who the stakeholders/owners are? Do you know the details about them, including how they are built, and the Software Bill of Materials (SBOM) for the OSS components they include?
2. **Supplier hygiene** (Suppliers) – Do you know if your OSS components come from a trusted, quality supplier?
3. **Build & release** – Do you understand how your software "parts" and processes come together to build and release applications into production?
4. **Project consumption** (Consumption) – Do you govern OSS component selection?
5. **Giving back** (Contribution) – Do you contribute to the OSS community?
6. **Policy control** (Risk Management) – What is your tolerance for risk? Do you have automated policy enforcement?
7. **Digital transformation** (Execution Plan) – What plans, resources, and training do you have to help institutionalize new processes and tools?

FIGURE 4.1

FIVE STAGES OF SOFTWARE SUPPLY CHAIN MANAGEMENT MATURITY

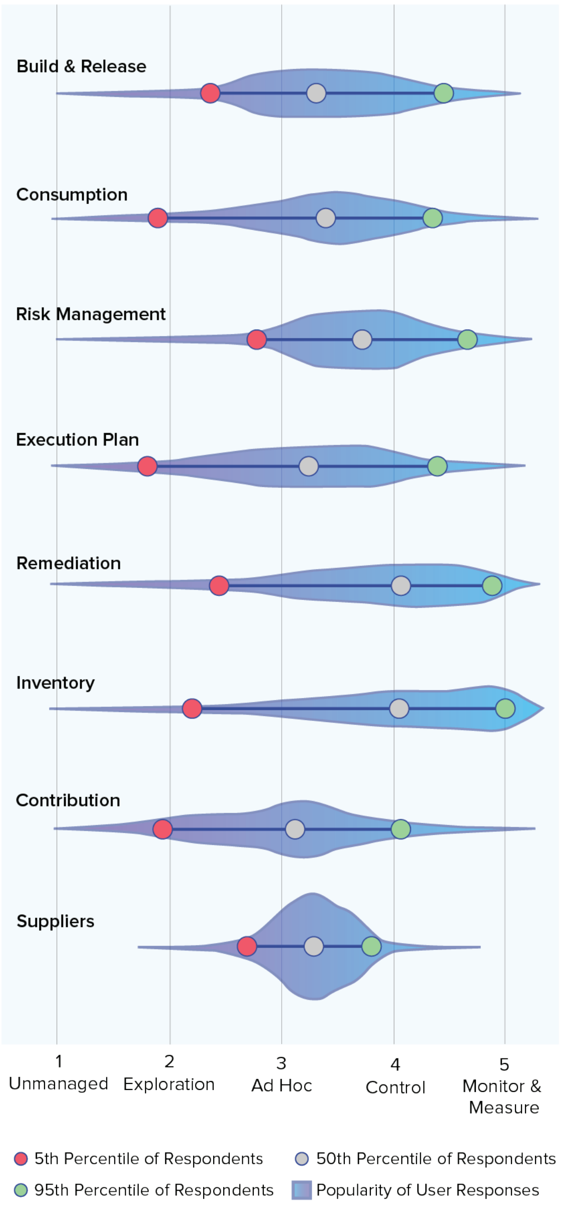
Less Mature ← → *More Mature*

UNMANAGED	EXPLORATION	AD HOC	CONTROL	MONITOR & MEASURE
This first stage is referred to as the **Unmanaged** stage because organizations are often operating with an "anything goes" mindset, are often reactive, and have minimal process/oversight related to the themes.	A realization of some sort is usually the impetus for thrusting an organization into the **Exploration** stage. This is often triggered by an "event" that causes an "all hands on deck" reaction to uncover necessary information/solutions, or a champion of some sort leading an improvement effort. This stage is often focused on identifying the perceived problem/inefficiency, learning about current implementations, and starting to explore potential solutions.	In the midst of starting to define processes and implement tooling to improve the identified problem, **Ad Hoc** solutions reign as the teams work toward institutionalization and socialization of new tooling and processes.	In the **Control** stage, ad hoc solutions give way to more formalized governance processes across the enterprise. Socialization and institutionalization of these processes and tools is ongoing, but for the most part, stakeholders are bought in to the need for improvement measures and are working towards compliance.	The **Monitor and Measure** stage occurs once new processes and tools have been institutionalized, and organizations have reached a phase of being able to proactively address OSS component risk. In addition, a healthy amount of ROI is realized, and measurements to demonstrate success are available.

FIGURE 4.2

SOFTWARE SUPPLY CHAIN MATURITY SCORE BY THEME

5th, 50th, and 95th Percentile



8. **Remediation** – How do you implement fixes to address identified OSS component risk?

Aggregate responses were then scored and mapped into five different stages of software supply chain management maturity, as defined in Figure 4.1.

How Mature are Today's Software Supply Chains?

Based on the survey results, it's a bit of a mixed bag. Let us explain.

In Figure 4.2, we've plotted the 702 responses against the five different stages of maturity. We see that, across the various themes, the majority of respondents were graded less than the "Control" level of maturity. Further, based on the definitions above, we can assert that the "Control" level of maturity is the point at which an organization transitions from "figuring it out" to a minimal level of maturity that will enable high-quality outcomes. The three levels of maturity (Unmanaged, Exploration, Ad Hoc) prior to the "Control" level of maturity are suboptimal; this is where most of survey responses were scored.

Reality vs. Perception on Software Supply Chain Maturity

The majority of respondents demonstrate an "Ad Hoc" approach to software supply chain management for all themes except two: Remediation and Inventory. Respondents indicate they are remediating risky components and they understand where the risk resides. This is true even though they have an "Ad Hoc" approach to Build & Release and Risk Management processes.

The survey suggests that respondents have talked themselves into believing that they're doing a good job, leading at the least to **a false sense of security** and at worst to huge **inefficiencies in the engineering process.**

We also compared the objective analysis done in chapters 2 and 3, which analyzed 100,000 applications, to the subjective survey responses. The data shows a clear disconnect between what is actually happening, and what people think is happening: 70% of remediations are suboptimal, which aligns with the "Ad Hoc" maturity rating for both Risk Management and Execution practices.

In summary, the survey suggests that respondents have talked themselves into believing that they're doing a good job, leading at the least to a false sense of security and at worst to huge inefficiencies in the engineering process. Objectively, however, the data from Chapters 2 and 3 indicates that development teams are not following structured guidance, and do not have intelligent tooling to ensure quality outcomes. Reconciling this perception with reality will help organizations in achieving the promised efficiency gains in dependency management.

CHAPTER 5

Emergence of Software Supply Chain Regulation and Standards

What's Happening in the United States?

Following the multitude of attacks in 2020 aimed at software supply chains, the United States Federal Government took notice and began to take action.

February

Starting in **February 2021 —** President Biden issued an **Executive Order** (EO) laying out changes to secure all supply chains, including software. The order called upon the Secretaries of Commerce and Homeland Security to coordinate with heads of appropriate agencies to report on the security and integrity of critical information and communications technology software supply chains.

Also in **February**, the CIO for the Department of Defense (DoD CIO) rolled out a new reference architecture called **Department of Defense (DOD), Zero Trust Reference Architecture** (ZTA). Within the ZTA the DoD CIO outlined various Zero Trust Pillars and Capabilities including a section centered on protecting applications and software supply chains.

The reference architecture specifically calls for protection of Applications and Workloads, defined as:

> *Applications and workloads include tasks on systems or services on-premises, as well as applications or services running in a cloud environment. Zero Trust workloads span the complete application stack from application layer to hypervisor. Securing and properly managing the application layer, as well as compute containers and virtual machines is central to Zero Trust adoption. Application delivery methods such as proxy technologies, enable additional protections to include Zero Trust decision and enforcement points. Developed Source Code and common libraries are vetted through DevSecOps development practices to secure applications from inception.*

The Executive Order "on Improving the Nation's Cybersecurity" is a milestone for the U.S. government.

The document further defines software supply chain protection as:

> *The ability to validate the security on a binary, library, or source code used to build an application.*

April

In **April 2021**, the United States saw the formalization of software supply chain standards begin to take shape when the CISA and National Institute of Standards and Technology (NIST) released their paper "**Defending Against Software Supply Chain Attacks**"

In it, the two agencies highlighted that software compromised in supply chain attacks could have "widespread consequences for government, critical infrastructure, and private sector software customers." They also noted how these types of attacks can easily allow bad actors to get around other cyber defenses to carry out cyber espionage.

The document provides in-depth guidance for both governments and companies to implement reasonable safeguards to secure their software supply chains.

Suggestions include:

- **Developing a written program** to address software supply chain risk.
- **Inventorying organizational reliance** on external software and code across all operational departments.
- **Assessing risk** from these vendors and adopting appropriate contractual and other safeguards.
- **Coordinating efforts** across management, IT, personnel, compliance, product development and operational departments.
- **Monitoring the threats and vulnerabilities** to the software supply chain, including through technical measures and threat analysis, on an ongoing basis.

May

In **May 2021**, Biden signed a second **Executive Order** for software supply chains, this time, as part of a critical look at the nation's cybersecurity posture. The EO "on Improving the Nation's Cybersecurity" is a milestone for the U.S. government.

The EO prescribed a number of technologies, including multi-factor encryption and endpoint detection as critical to protecting the nation's cyber assets. Further, the EO established a detailed plan for taking steps to secure the federal software supply chain. The order called for NIST to publish guidelines for establishing best practices to detect vulnerabilities, and requirements that all critical software delivered to government customers including an SBOM. It also included milestones that agencies must meet to demonstrate progress toward the goals.

June

In **June 2021**, as directed by the EO, NIST released their **definition of "critical software"** defining it as:

> *[...] any software that has, or has **direct software dependencies** upon, one or more components with at least one of these attributes:*
>
> - *is designed to run with elevated privilege or manage privileges;*
> - *has direct or privileged access to networking or computing resources;*
> - *is designed to control access to data or operational technology;*
> - *performs a function **critical to trust**; or,*
> - *operates outside of normal trust boundaries with privileged access.*

The definition applies to software of all forms (e.g., standalone software, software integral to specific devices or hardware components, cloud-based software) purchased for, or deployed in, production systems and used for operational purposes.

July

In July 2021, NIST **published guidance** for outlining security measures for critical software use and minimum standards for vendors' testing of their software source code.

Also in **July**, the National Telecommunications and Information Administration (NTIA) released a minimum definition of an SBOM. This was a critical step toward improving transparency for software supply chains for both technology vendors and government customers.

The NTIA describes an SBOM as "effectively a nested inventory, a list of ingredients that make up software components, and provides those who produce, purchase, and operate software with information that enhances their understanding of the supply chain. SBOMs are a formal, machine-readable inventory of software components and dependencies. SBOMs contain information about those components, and their hierarchical relationships. SBOMs may include open source or proprietary software and can be widely available or access-restricted."

Further, NTIA defined the **minimum elements for a SBOM** as three broad, interrelated areas:

1. **Data Fields:** Documenting baseline information about each component that should be tracked.
2. **Automation Support:** Allowing for scaling across the software ecosystem through automatic generation and machine-readability.
3. **Practices and Processes:** Defining the operations of SBOM requests, generation, and use.

Furthermore in **July**, both the House of Representatives and the Senate began drafting legislation in two separate committees.

The House's Homeland Security Committee introduced seven bipartisan bills, five of which focused strictly on strengthening cybersecurity, including a "Pipeline Security Act," and "Cybersecurity Vulnerability Remediation Act."

The Senate's Homeland Security and Governmental Affairs Committee introduced **The Supply Chain Security Training Act**, calling it, "bipartisan legislation to help protect against cybersecurity threats and other technological supply chain security vulnerabilities that arise when the federal government purchases services, equipment or products.

"As supply chains become interconnected, vulnerabilities in suppliers' products and services ... become more attractive targets for attackers."

— U.K. government's request for advice on defending against digital supply chain attacks

What's Happening in the United Kingdom?

In **May 2021,** the U.K. government announced that it was seeking advice on **defending against digital supply chain attacks** from organizations that either consume IT services, or MSPs that provide software and services.

The request noted:

> *"As supply chains become interconnected, vulnerabilities in suppliers' products and services correspondingly become more attractive targets for attackers who want to gain access to the organisations. Recent high-profile cyber incidents where attackers have used MSPs as a means to attack companies are a stark reminder that cyber threat actors are more than capable of exploiting vulnerabilities in supply chain security, and seemingly small players in an organisation's supply chain can introduce disproportionately high levels of cyber risk."*

Also in **May**, the Department for Digital, Culture, Media, and Sport (DCMS) **opened up a survey** that closed in early July, and invited comments from industry experts and tech organizations on stepping up supply chain security across the UK.

The initiative is a part of the U.K.'s nationwide "**cyber resilience**" efforts set out in its National Cyber Security Strategy to safeguard businesses and organizations that increasingly rely on technology from cyber-attacks, and to strengthen overall digital supply chain security.

While the feedback has not been released to the public yet, the U.K. government has noted that it will result in the re-evaluation of supply chain risks, reviewing policies, and likely implementing new guidelines and frameworks to strengthen specific areas of digital supply chain security. It could also mean the introduction of new, country-wide legislation for software firms and IT service providers.

What's Happening in Germany?

In **May 2021,** Germany passed the **Information Technology Security Act 2.0** as an update to the First Act to "increase cyber and information security against the backdrop of increasingly frequent and complex cyber-attacks and the continued digitalisation of everyday life." While this Act influences many areas of the IT industry in Germany, it specifically states that suppliers, i.e. manufacturers of critical components, will also be subject to certain obligations to safeguard the entire supply chain.

Critical components are defined as IT products:

1. that are used in critical infrastructures;
2. for which disruptions to availability, integrity, authenticity, and confidentiality may lead to a failure or a significant impairment of the functionality of critical infrastructures or to threats to public safety; and
3. that on the basis of a law regarding this provision are designated as a critical component, or realize a function designated as critical on the basis of a law.

What's Happening in the European Union?

In **July 2021**, the ENISA issued a report titled "**Understanding the increase in Supply Chain Security Attacks**" that reviewed 24 different software supply chain attacks and how they came to fruition.

It found that:

- "In order to compromise the targeted customers, attackers focused on the suppliers' code in about 66% of the reported incidents."
- "For 58% of the supply chain incidents analysed, the customer assets targeted were predominantly customer data, including Personally Identifiable Information (PII) data and intellectual property."
- "For 66% of the supply chain attacks analysed, suppliers did not know, or failed to report on how they were compromised. However, less than 9% of the customers compromised through supply chain attacks did not know how the attacks occurred."

"Attackers focused on the suppliers' code in about 66% of the reported incidents."

— ENOSA report, "Understanding the Increase in Supply Chain Security Attacks"

More importantly, the report shared recommendations that organizations should put in place. While more guidance than regulation, it does foreshadow what could come down the road.

Suggestions include:

- identifying and documenting suppliers and service providers;
- defining risk criteria for different types of suppliers and services such as supplier and customer dependencies, critical software dependencies, single points of failure;

- monitoring of supply chain risks and threats;
- managing suppliers over the whole lifecycle of a product or service, including procedures to handle end-of-life products or components;
- classifying of assets and information shared with or accessible to suppliers, and defining relevant procedures for accessing and handling them.

The report also suggests possible actions to assure that the development of products and services comply with security practices. Suppliers are advised to implement better policies for vulnerability and patch management.

Recommendations for suppliers include:

- ensuring that the infrastructure used to design, develop, manufacture, and deliver products, components and services follows cybersecurity practices;
- implementing a product development, maintenance, and support process that is consistent with commonly accepted product development processes;
- monitoring of security vulnerabilities reported by internal and external sources that includes used third-party components;
- maintaining an inventory of assets that includes patch-relevant information.

What's Happening Globally?

May

In **May 2021,** the United Nations released a report two years in the making from "the Group of Governmental Experts on Advancing responsible State behaviour in cyberspace in the context of international security."

Similar to actions at the national and regional levels, the report touches on several areas of cybersecurity, and provides guidance on the "reasonable steps States should take to ensure the integrity of the supply chain so that end users can have confidence in the security of information and communication technology (ICT) products."

The report notes:

> *Ensuring the integrity of the ICT supply chain and the security of ICT products, and preventing the proliferation of malicious ICT tools and techniques and the use of harmful hidden functions are increasingly critical in that regard, as well as to international security, and digital and broader economic development.*
>
> *Global ICT supply chains are extensive, increasingly complex and interdependent, and involve many different parties. Reasonable steps to promote openness and ensure the integrity, stability and security of the supply chain can include:*
>
> *(a) Putting in place at the national level comprehensive, transparent, objective and impartial frameworks and mechanisms for supply chain risk management, consistent with a State's international obligations.*
>
> *(b) Establishing policies and programmes to objectively promote the adoption of good practices by suppliers and vendors of ICT equipment and systems in order to build international confidence in the integrity and security of ICT products and services, enhance quality and promote choice.*
>
> *(c) Increased attention in national policy and in dialogue with States and relevant actors at the United Nations and other fora on how to ensure all States can compete and innovate on an equal footing, so as to enable the full realization of ICTs to increase global social and economic development and contribute to the maintenance of international peace and security, while also safeguarding national security and the public interest.*

You can read the **full list of guidance** provided by the United Nations.

June

In **June 2021**, the United States and the European Union formed a Trade and Technology Council (TTC). This was in part developed to work together on the fight to secure critical technology and software supply chains. According to the White House, the TTC "will be composed of working groups focused on advancing cooperation on tech standards on artificial intelligence, the internet of things and other emerging technologies, ICT security, data governance, investment screening and semiconductors."

About the Analysis

The authors have taken great care to present statistically significant sample sizes with regard to component versions, downloads, vulnerability counts, and other data surfaced in this year's report. While Sonatype has direct access to primary data for Java, JavaScript, Python, .NET and other component formats, we also reference third-party data sources as documented. Further, Sonatype's research analyzed scan data from 100,000 anonymized, validated applications.

Acknowledgments

Each year, the State of the Software Supply Chain report is a labor of love. It is produced to shed light on the patterns and practices associated with OSS development and the evolution of software supply chain management practices.

The report is made possible thanks to a tremendous effort put forth by many team members at Sonatype, including: Bruce Mayhew, Dr. Stephen Magill, Matt Howard, Ax Sharma, Sal Kimmich, Elissa Walters, Alli VanKanegan, Juan Morales, Moncef Ben-Soula, Cody Nash, Andrew Yorra, Brian Fox, Mike Hansen, Joel Orlina, Melissa Schmidt, Ember DeBoer, Ilkka Turunen, Luke Mcbride.

We would also like to offer thanks for contributions big and small and for sharing perspective to our many colleagues across the DevOps and open source development community.

A very special thanks goes out to Alli VanKanegan who created the incredible design for this year's report.

sonatype

Sonatype is the leader in developer-friendly, full-spectrum software supply chain management providing organizations total control of their cloud-native development life cycles, including third-party open source code, first-party source code, infrastructure as code, and containerized code. The company supports 70% of the Fortune 100 and its commercial and open source tools are trusted by 15 million developers around the world. With a vision to transform the way the world innovates, Sonatype helps organizations of all sizes build higher quality software that's more aligned with business needs, more maintainable, and more secure. For more information, please visit **Sonatype.com**, or connect with us on **Facebook**, **Twitter**, or **LinkedIn**.

Headquarters
8161 Maple Lawn Blvd, Suite 250
Fulton, MD 20759
USA • 1.877.866.2836

European Office
199 Bishopsgate
London EC2M 3TY
United Kingdom

APAC Office
5 Martin Place, Level 14
Sydney 2000, NSW
Australia

Sonatype Inc.
www.sonatype.com