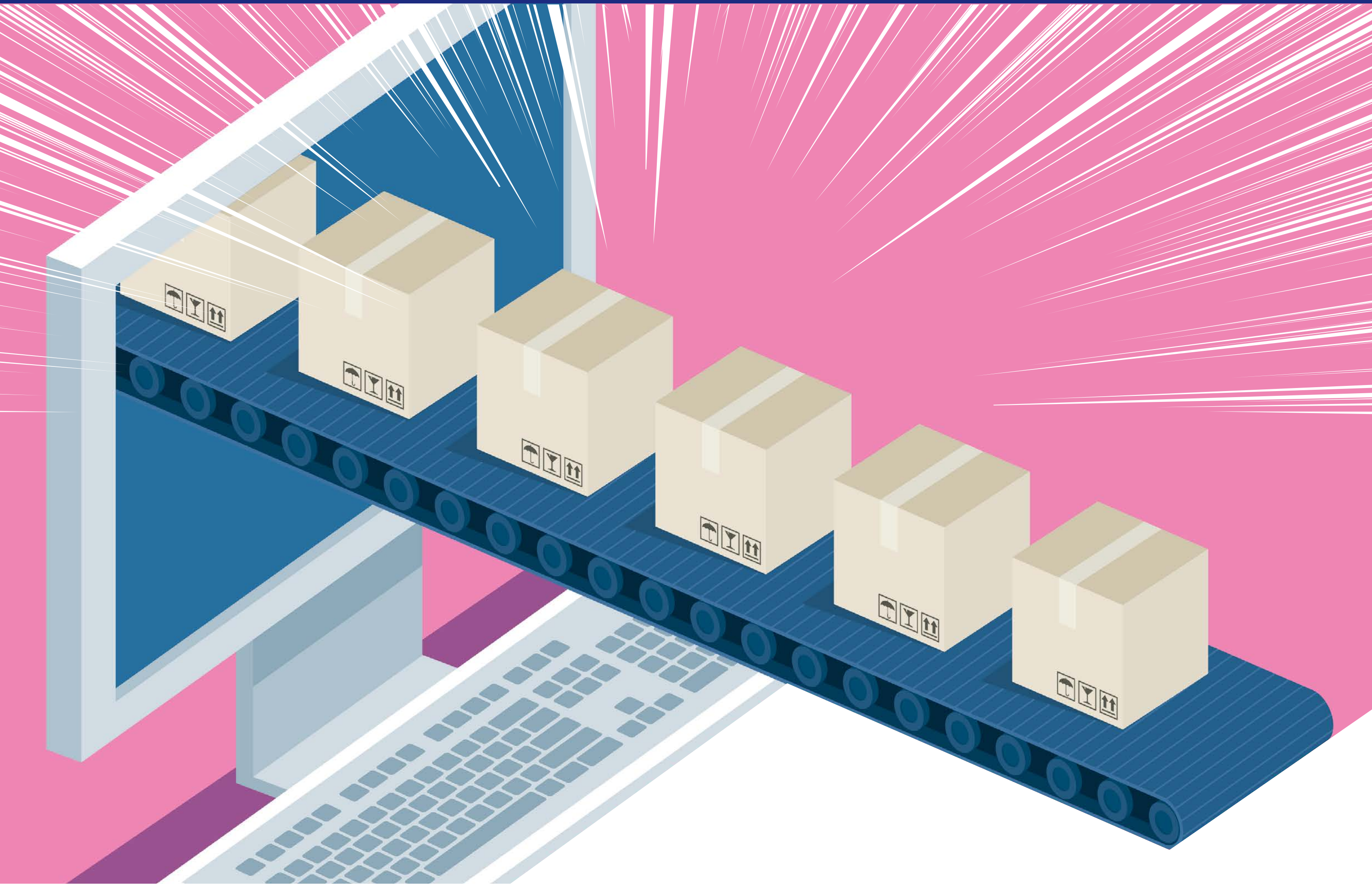
Checkmarx

The world runs on code. We secure it.

WHITEPAPER

Don’t Take Code from Strangers

An Introduction to Checkmarx Supply Chain Security



Abstract:

This white paper is designed to help organizations, management teams, security practitioners, and developers understand dependency integrities that exist within open source code packages and why they represent the weakest link within a software supply chain.

This white paper begins with a look at the relationship between the digital economy and open source software (OSS), with a focus on why open source code is a popular attack vector. It then introduces SLSA as a framework for supply chain integrity, discusses why traditional software composition analysis is insufficient when it comes to detecting code with malicious intent, and introduces a way forward to avoid taking malicious code from strangers.

After reading this white paper, readers will understand why an analysis of the **code repository, contributor reputation,** and **code behavior** is imperative for uncovering compromised code dependencies. Most important, readers will learn about the introduction of a new, innovative Checkmarx technology that blends best-of-breed software composition analysis with a visionary approach to detecting dependency issues.

Available today, this technology is designed to empower organizations to manage the risks associated with open source software and ensure software supply chain security.

Table of Contents

The Digital Economy Runs on Open Source

Software is running the world – and it is everywhere. More and more elements of major businesses and industries are being run on software and delivered as online services. From entertainment, retail, financial services, and healthcare to automotive, transportation, agriculture, and national defense – industries across the board have experienced a fundamental software-based transformation that is so crucial for business longevity.

Nowhere is the link between digital transformation and business longevity so apparent than in light of the coronavirus pandemic. With rare exception, operating digitally was the only way to stay in business through mandated shutdowns and restricted activity. Thanks to 'Go Digital or Go Dark,' the pandemic accelerated digital transformation with businesses competing aggressively to be first to market with digital products and services. Now, as digital transformation continues to accelerate, pressure continues to mount on developers to write and deploy new applications and new features faster than ever.

But perhaps 'write' is not the right word.

Modern applications are more often assembled than they are written, with developers combining multiple open source packages, along with proprietary code, in a single application. Virtually all contemporary, proprietary software incorporates open source components. Items that impact everyday life, such as automobiles and phones, to cutting-edge artificial intelligence programs use open source software such as the Linux kernel operating system, Kubernetes (which powers cloud computing), and the Apache and Nginx web servers (which run over 60% of the world's websites). Recent industry research reveals that[1]:

- 90% of cloud servers, 82% of smartphones, and 62% of embedded systems run on open source operating systems
- More than 70% of 'Internet of Things' devices use open source software
- 90% of the Fortune Global 500 operate on open source software

The value of open source software is undisputed. But unlike proprietary software, which companies build internally, open source code is developed by typically unpaid developers, often as a part of a community-driven project in which ideas and contributions are shared. The software is made available to the community as what are referred to as projects or components – and are available to anyone for free. While this model allows for innovation to occur organically throughout the community, it's understood that any updates, patches, and new releases are also the responsibility of that volunteer community. However, the ultimate accountability falls on those who use open source.

This all begs the question, "How scared should we be that so much of the software on which the world depends is open source software?"

[1]Michigan Technological University, Tech Today, Open Source, October 2019.

Malicious code lurking within

The use of open source software is not new – and neither is the presence of code published in repositories for malicious purposes.

The prevalence of using open source software in corporate settings has increased drastically with the adoption of modern application development, and with it, the use of code from projects with less rigorous controls than were typical in the past. Because repositories often invite users to add updates and features, anyone – including threat actors – can publish their own code and contribute to an open source project.

Due to the volunteer nature of open source communities, once a developer has been accepted as a trusted member, his or her activity within the codebase may not be closely monitored. That means that an attacker could initially make valid, useful contributions, and then once trust and credibility has been established, add malicious code to the codebase unobserved. If you think this only occurs in the movies, think again. GitHub once reported that 20 percent of the bugs within code stored on its platform were ***planted by malicious actors.***[2]

Because open source projects are built on a foundation of community involvement and trust, open source libraries offer threat actors a large return on investment. The ease with which threat actors are able to exploit code dependencies to introduce malware and backdoors has made software supply chain attacks a popular attack vector. As a result, supply chain incidents stemming from malicious actors deliberately injecting hard-to-detect, weaponized code into open source packages are on the rise. Industry research bears this out. "By 2025, 45 percent of organizations worldwide will have experienced attacks on their software supply chains, a three-fold increase from 2021."[3]

It's time to ask a new question. Rather than asking how afraid should we be that the digital economy runs on open source, the question one should be asking is, "To what extent should we trust that an open source code package is free of code with malicious intent (i.e., Trojan Horses) – and how can we ensure we don't take code from strangers?"

Code Goes Rogue in Protest

With a user base of nearly 25 million downloads each week, Colors.js and Faker.js are two of the most popular NPM libraries. Supporting a number of open source projects, including Amazon's Cloud Development Kit, the last thing anyone wants is for them to stop working, but in January 2022, they did just that.

Marak Squires, author of the two JavaScript libraries, sabotaged his work (seemingly in protest against "Fortune 500" Companies benefitting from free open source code) with code that crashed tens of thousands of JavaScript programs in one strike.

The updates produced an infinite loop that caused dependent apps to spew gibberish, prefaced by the words 'Liberty Liberty Liberty.' The update sent developers scrambling as they attempted to fix their malfunctioning apps. While the damage was limited to the urgent need to fix numerous tools that became inoperable, the event demonstrates a more concerning problem. Just as easily, Squires could have introduced malicious code, which would be executed on hundreds of thousands of machines that were known to download the faulty version of the package.

[2]ZDNet, Almost one in five bugs are planted for malicious purposes, Liam Tung, December 2020.
[3]Gartner®, "How Software Engineering Leaders Can Mitigate Software Supply Chain Security Risks", Manjunath Bhat, Dale Gardner, Mark Horvath, 15 July 2021. GARTNER is a registered trademark and service mark of Gartner, Inc. and/or its affiliates in the U.S. and internationally and is used herein with permission. All rights reserved.

Introducing SLSA:
An End-to-End Framework for Supply Chain Integrity

With open source code repositories proving to be a popular and reliable attack vector for threat actors, and attacks on software systems on the rise over the last two years, the software supply chain has come under close scrutiny.

Looking at a typical software delivery process from a high level, a developer sends code to a source control repository, which initiates a build process. The build system collects and compiles the source code. Binaries are signed and packaged. Lastly, the package is made available for use by end-users or downstream projects that incorporate the package into their software.

Because the software development and deployment processes are complex, threat actors can introduce malware into the development workflow using a variety of attack methods. To guard against the most serious supply chain concerns and preserve the integrity of software artifacts throughout the software supply chain, Google launched Supply-chain Levels for Software Artifacts ***(SLSA)*** as an end-to-end framework in collaboration with the Open Source Security Foundation ***(OpenSSF)***. The SLSA framework formalizes criteria around software supply chain integrity and assists the industry and open source community in securing the software development lifecycle.

SLSA is based on the fundamental principle that all software artifacts must meet the following two requirements:

> **Non-unilateral.** No one person can make changes to a software artifact anywhere in the software supply chain without the explicit evaluation and consent of at least one additional 'trusted person.'

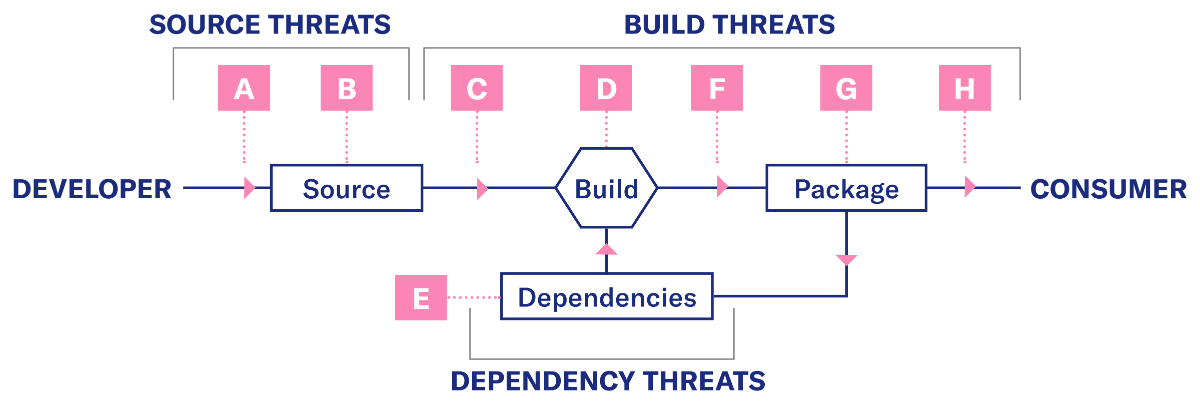
> **Auditable.** The software item can be traced back to its original, human-readable sources and connections in a secure and transparent manner.

The SLSA framework also establishes three trust boundaries to both encourage the right standards, attestation, and technical controls and to empower developers to harden a system from threats and risks. The three trust boundaries include:

> **Source integrity.** Source threats include bypassed code review and compromised source controls system.

> **Build integrity.** Build threats include modified code after source control, compromised build platform, bypassed CI/CD, compromised package repository, and use of a bad package.

> **Dependency integrity.** Dependency threats include use of a bad dependency, including transitive dependencies



SOURCE THREATS
- A Bypassed code review
- B Compromised source control system

BUILD THREATS
- C Modified code after source control
- D Compromised build platform
- F Bypassed CI/CD
- G Compromised package repo
- H Using a bad package

DEPENDENCY THREATS
- E Using a bad dependency

Figure 1. Trust Boundaries in the Software Development Life Cycle

Dependency integrity is the weakest link

No chain is stronger than its weakest link, and this rule carries into the software supply chain. Of the three trust boundaries, dependency integrity presents the weakest link in the software supply chain for several reasons:

> **Software Development Life Cycle (SDLC) controls.** In most organizations, SDLC is a well-defined process that integrates security controls. Organizations, for example, have control over their source code and a degree of control over the components and tools used during the build process. In addition, parts of the software supply chain have some sort of security or mitigation actions that can be applied, such as enforcing 2FA to GitHub, monitoring the build server, and protecting the internal repositories.

> **Lack of standards for external code.** Open source communities do not enforce standards for code published to repositories. Because there are no standards with external code packages, you can't assume open source contributors adhere to standards or certify their code.

> **Difficult-to-spot techniques.** Dependency confusion attacks make use of difficult-to-spot strategies such as vulnerabilities in package manager configurations and operations that facilitate repository jacking.

> **Transitive dependencies.** Transitive dependencies, where a package calls a package, which calls a package, which calls a package, and so on, may represent hidden risk. A piece of software, for example, can be hundreds of layers deep with each component having dependencies. If an attacker can compromise a downstream dependency, they will have achieved the critical step of initial access.

> **Lack of visibility into flawless code.** Organizations have almost no control over or visibility into the external code packages chosen to be integrated into software systems. Hidden among the masses of Python modules, Node.js packages, and .NET libraries are an untold number of libraries that may be securely coded and elegantly implemented, but also very malicious at the same time.

- **Contributor rage.** Software systems that rely on open source code packages may be at the mercy of dependable code that suddenly goes rogue, becoming annoying at best and devastating at worst. In January 2022, for example, Marak Squires introduced changes that rendered his popular colors and faker NPM code packages useless.

When Good Software Goes Bad: Colorama's Evil Twin

Sneaking malicious software into existing codebases isn't new, but library names that contain a word that can have multiple spellings make it all too easy. Such was the case with colorama and colourama.

Colorama is a legitimate Python package that translates ANSI color commands to the Windows terminal. It's a fairly popular library, and with well over two thousand stars on GitHub, it has a good reputation. Colourama, on the other hand, is a form of typosquatting that was deliberately made to trick British-English users looking for colorama.

Financially motivated, it copied the original code and added malware that hijacked infected users' Windows operating system clipboard, where it would scan every 500ms for a Bitcoin address. When found, it would replace it with attacker's own Bitcoin address to redirect Bitcoin payments/transfers made by an infected user. Because the VBscript created persistence through a registry entry, special attention was needed to completely uninstall the colourama package and VBscript.

Traditional Code Analysis Systems Fall Short

Open source code accounts for the majority of code in today's modern applications. The unfortunate reality with open source dependencies is that along with the benefits come increased, undetectable risks. To avoid becoming a victim (or unsuspecting accomplice) of a software supply chain attack, detecting and defending against dependency-based attacks is vital. That, however, is easier said than done as traditional code analysis solutions to detect malicious code dependencies fall short.

Software composition analysis (SCA) Traditional SCA products analyze applications, generally during the development process, to detect embedded open source software and, sometimes, other third-party components. Software composition analysis tools can be relied upon to identify known vulnerabilities, such as out-of-date libraries that have available security patches, and to determine the license used to distribute a software package in order to aid in assessing any legal risks.

When it comes to identifying malicious code dependencies, traditional SCA is insufficient because it takes a reactive approach versus a proactive approach to identifying risk. This is because SCA relies on someone else to find the software vulnerability, publish it, and issue an alert to upgrade or patch your system. SCA tools operate by determining whether there is a known vulnerability or CVE and whether there is a more recent version of the code package. If no one finds the problem, it results in a long mean time to detection (MTTD), with a worst-case scenario where attacks or a vulnerability could lie dormant for months.

One recent example of a classic open source weakness was the discovery of the Log4j vulnerability in early December 2021. Log4j was found to have a zero-day vulnerability that had existed since 2013 and which allows attackers to take control of a system, steal data, upload malware, and even mine cryptocurrency.

Static application security testing (SAST) Static application security testing solutions analyze an application's source, bytecode, or binary code for security vulnerabilities, typically at the programming and/or testing phases of the software life cycle.

Traditional SAST is an effective way to detect bugs in code, but is an ineffective way to detect perfect, albeit malicious, code that an attacker has injected into a code package. The code will appear to be legitimate on the surface – only when you crack the code open do you see what the code is really doing. Another drawback with SAST is the fact that SAST is not run against external code. Organizations only run SAST on their own code because it's not the organizations responsibility to fix bugs in dependencies.

Dynamic application security testing (DAST) Dynamic application security testing solutions analyze applications in their dynamic, running state during testing or operational phases. DAST simulates attacks against an application (typically web-enabled applications and services) and analyzes the application's runtime reactions to determine whether it is vulnerable.

The drawback with DAST is that it is intended to determine whether a developer made a coding mistake that introduced a vulnerability. It was never built to recognize and evaluate the reputation of a developer contributing code or reputation factors of open source code. Correctly coded malicious code will be recognized as a legitimate process. In addition, similar to SAST, organizations only run DAST to find vulnerabilities in their own internal code.

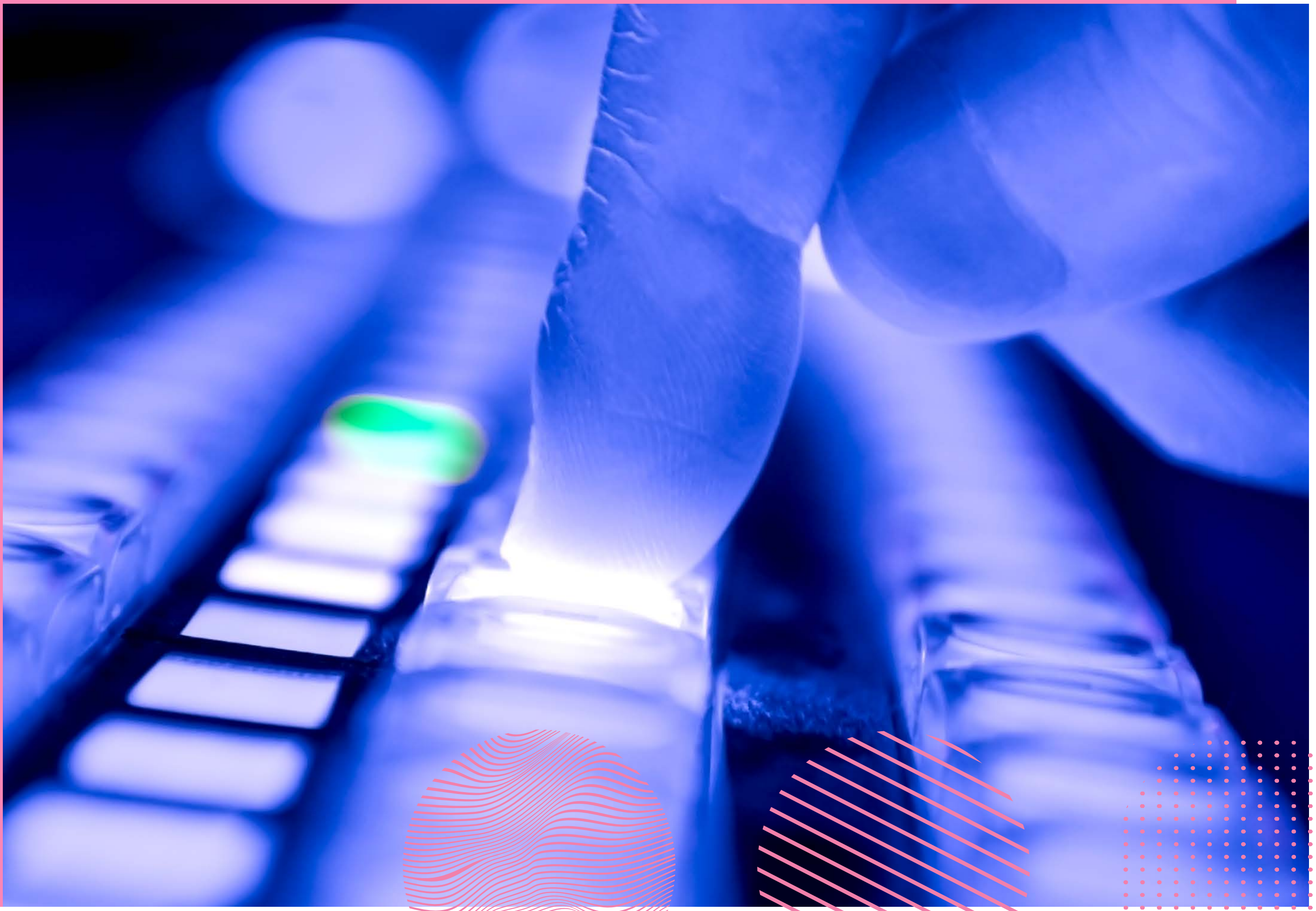
Popular NPM Package Hijacked to Publish Crypto-Mining Malware

In October 2021, a threat actor gained access to the NPM user account of one of the owners of the popular package ***UAParser.js.*** The attack group published new versions of the package (0.7.29, 0.8.0, 1.0.0), which included a few malicious files, and an additional 'preinstall' script in the 'scripts' section in the package.json file. This new 'preinstall' script was intended to trigger the execution of the malicious files upon package installation. The attacker's final goal was to infect package users with both a crypto miner and credential stealer malware.

This attack was reported by a vigilant user on the project's repository on GitHub, preventing what could have been the infection of millions of users.

Using the same techniques, and much of the same code, a ***similar attack*** occurred in early November on two other highly popular packages: 'coa' and 'rc.' Both packages were infected in a similar manner to the previous UAParser.js incident by an account takeover of the packages' owners. In this case, a bug in the attacker's code that prompted an error upon installation facilitated early detection and mitigation.

In both cases, the suspicious activity was noticed by chance, or at least not by any dedicated mechanism. Without a way to detect the activity, the next potential account takeover and subsequent infection has a good chance of going unnoticed for a relatively long period of time, and possibly causing severe damages.



The Way Forward to Trust in Open Source Code Packages

One of the biggest challenges for developers is the need to make informed choices about the open source software they use in their own software systems. Determining whether external code is malicious can be difficult because developers have virtually no visibility into the risks associated with open source code packages. Faced with having to decide whether to 'take it or leave it,' developers need a way to identify malicious dependencies in code packages so that they can make wise, informed choices.

Detecting supply chain attacks in code packages

To avoid taking malicious code from strangers, organizations need a proactive way to vet the open source code for malicious dependencies. Gaining trust in open source code requires analysis of the health and wellness of the community, reputation of the contributor, and behavior of the code package.

- **Community health and wellness.** Analyzing the health and wellness of a code repository provides insight as to how trustworthy the code packages are and how often an open source package is updated and maintained. Elements to be considered include:
  - How vibrant is the community?
  - Does the community have a lot of members?
  - How active is the community?
  - Do they actively commit code?
  - Who can commit code?
  - Can anyone commit code? Are outside contributions allowed?
  - What are the safety features of the repositories?
  - Who checks the code and how many reviewers are involved?
  - How responsive is the community to issues and do they have processes in place to resolve issues? What is the mean time to resolve issues, and do they publish metrics?
- **Contributor reputation.** Looking at who contributes the code, other packages they may have created, and their overall online presence can provide clues as to the potential intent of their coding activities. Similar to a credit score used by institutions to determine credit-worthiness of individuals, insight into a contributor's activities and reputation provides a trust score that can be used in determining whether to use a contributor's code package. Elements to be considered include:
  - Who is the person committing the code?
  - Is this person known?
  - Has this person been seen before?
  - Have they previously committed to any open source projects?
  - Is this the first time the person is committing to a project?
- **Package ecosystem.** Beyond scrutinizing the health and wellness of the community and the reputation of the contributor, it's important to consider the ecosystem in which the package operates. Evaluating what a piece of code does, what processes it creates, what ports it opens, and what connections it tries to make are all critical indicators of a package's intent. Elements to consider include:
  - Is the package name similar to another popular package?
  - Is the version number unusually high?
  - Does the code try to execute anything?
  - Is the code running shell commands?

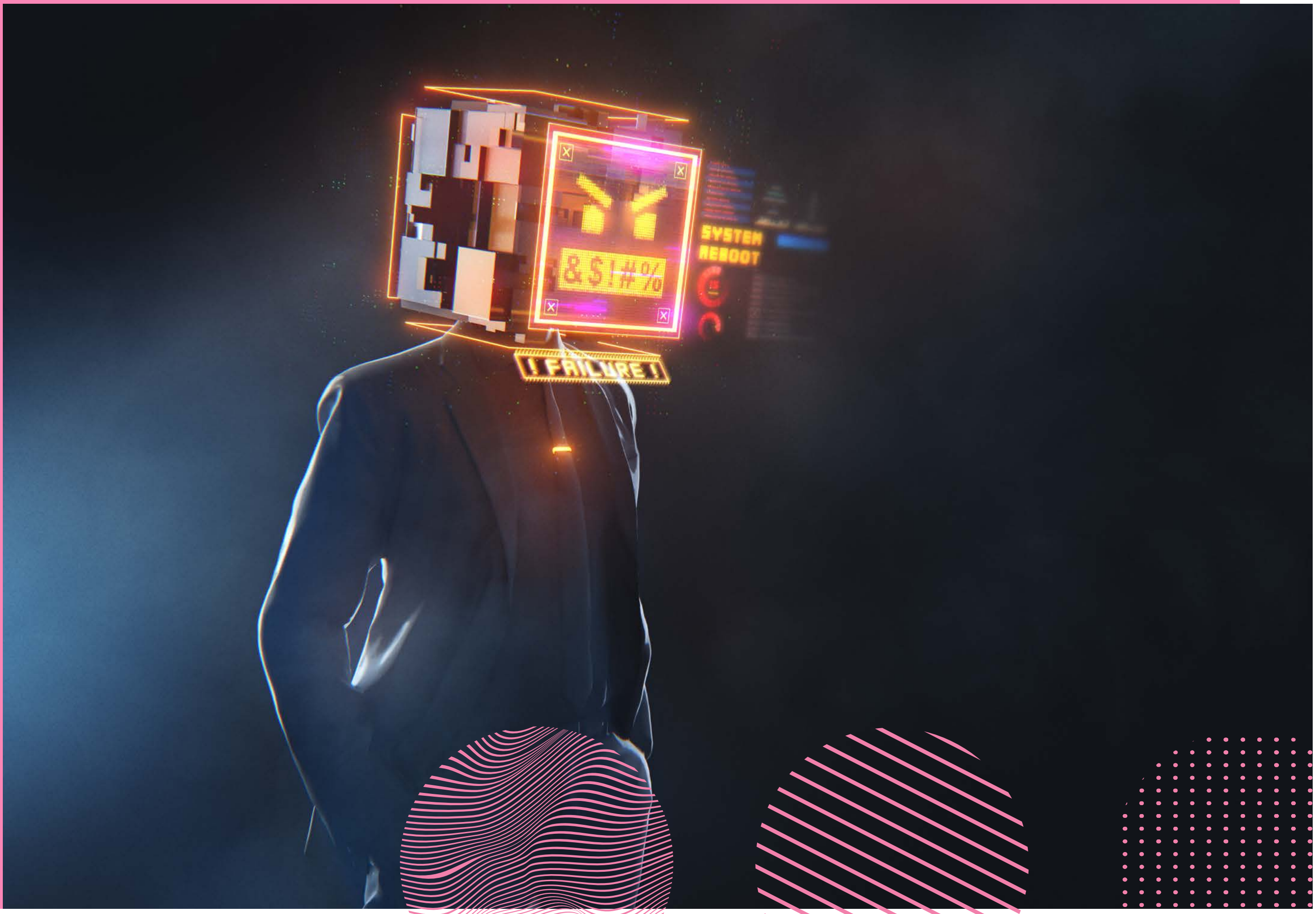
Is the code trying to extract anything from the system within which is it running?

Just 11 Lines of Deleted Code Nearly Broke the Internet

Did you know that in March 2016, the internet almost came crashing down? It began with a dispute between developer Azer Koçulu and messaging company Kik over a module Koçulu was working on, also called kik.

The company wanted him to change the name of his module so they could use the name kik for their own product. When Koçulu declined, NPM became involved in the argument. Instead of siding with Koçulu, NPM agreed with the company, rationalizing that allowing Kik the company to use the package name kik would make more sense. Deeply angered by the decision, Koçulu deleted all 273 modules he'd registered on NPM. Because all the focus was on kik, no one considered the ramifications of deleting the left-pad module.

Koçulu's simple, 11-line-long 'left-pad' module was heavily relied upon by the programming community, including companies such as Facebook, Netflix, and Airbnb. Thanks to caching, the vast majority of internet users didn't experience any downtime and wouldn't have noticed anything out of the ordinary. But for web developers, it was a temporary nightmare. Faced with thousands of builds failing each second, NPM took unprecedented action and re-published the original 'left-pad' module from a back-up.



Checkmarx Pushes the Boundaries of Secure Software Supply Chain Innovation

Open source code presents myriad benefits; however, the potential for dependency integrity to be compromised is an ever-present threat. Developer teams must therefore operate with the proactive assumption that all code may have been maliciously manipulated and apply a zero-trust security mindset to all external code packages being integrated into modern applications.

When an open source dependency is discovered to have a known vulnerability or have been deliberately compromised, it needs to be found and fixed immediately. Development teams could try to manage this risk by manually poring over vulnerability databases and matching alerts with dependencies in use, but doing this for the hundreds of code packages in your supply chain is unsustainable – and creating your own risk analysis of the thousands of open source contributors is impractical.

In today's rapid software development lifecycle, development teams can't afford to have security testing slow them down and security teams can't afford to have vulnerable software in production. As organizations employ modern application development approaches like Agile and DevOps to ensure ever-more aggressive release cycles, the ability to deliver insight and results into the hands of the people who need it, in the manner in which it is most helpful to them without impeding their productivity, becomes a fundamental development requirement.

Checkmarx elevates the standard for Software Composition Analysis

Checkmarx's approach to software composition analysis addresses these issues by providing accurate, relevant, and actionable open source risk insight, backed by a dedicated open source security research team, and seamlessly integrated throughout the SDLC. Checkmarx SCA™, which comes as a standalone solution, and is a component of the Checkmarx Application Security Testing Platform, allows developers to build software with confidence using a mix of custom and open source code. Checkmarx SCA and developer-centric Application Security Testing (AST) solutions combined do more than just tell you that you may have a security problem. They help you understand the exact nature of the problem, assign it a priority level, and determine the most efficient method for remediating it.

Checkmarx goes beyond traditional vulnerability analysis

Checkmarx simplifies the process of uncovering compromised dependencies by extending Checkmarx SCA with unique, innovative technology specifically designed to identify supply chain attacks. By integrating machine-learning-driven behavioral analysis and contributor-reputation indicators alongside SCA's curated threat feed, independent security research, and market-leading capabilities like Exploitable Path, Checkmarx provides a unified view into the risk, reputation, and behavior of open source packages and delivers a holistic, unified, and effective approach for managing the risks associated with open source code packages.

Uncover compromised dependencies with Supply Chain Security

Checkmarx SCA with Supply Chain Security (SCS) offers a more comprehensive approach to preventing supply chain attacks and securing open source usage by enabling developers to perform vulnerability, behavioral, and reputational analysis from a single, integrated platform. By natively integrating advanced behavioral analysis into SCA, Checkmarx provides developers with a streamlined, frictionless user experience to enhance their organization's supply chain security.

Checkmarx Supply Chain Security enables organizations to accelerate modern application development using open source software safely and securely through a full suite of critical capabilities:

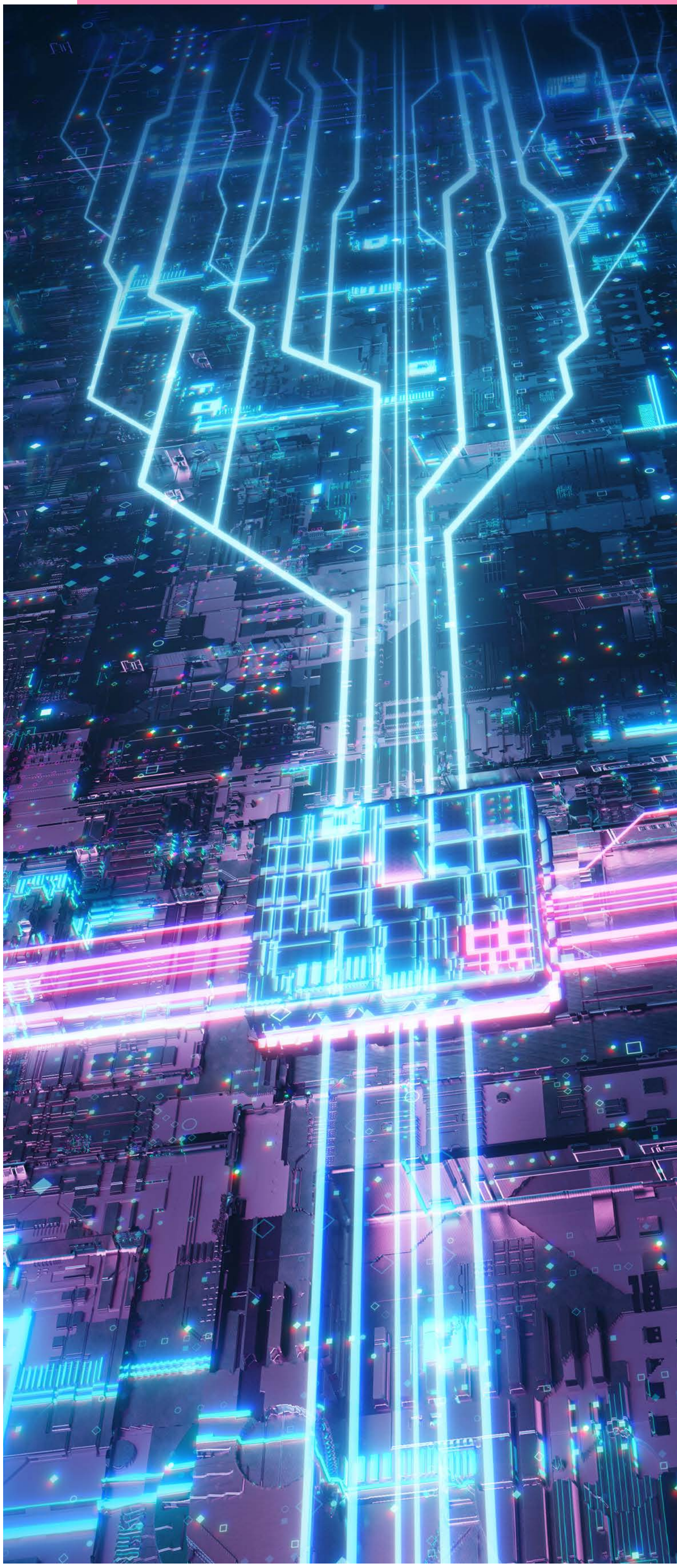
> **Health and Wellness and Software Bill of Materials (SBOM):** Provides knowledge of the open source package and community, combined with SBOM creation.

> **Malicious Package Detection:** Detects dependency confusion, typosquatting, chainjacking and other malicious activities and packages.

> **Contributor Reputation:** Restores trust in the provenance of open source packages by eliminating the need to manually analyze contributor activity across all projects that could impact an organization.

> **Behavior Analysis:** Incorporates static and dynamic analysis to observe how the code runs. Our detonation chamber provides deep analysis of code packages and removes ambiguity to defend against stealthy threats.

> **Continuous Results Processing:** Delivers constant updates on our research and threat hunting, maintaining a reputation and vulnerability database for customer usage.



Final Thoughts

Open source software has facilitated the acceleration of application development and shortened development cycles. As with any new advancement in technology, there can be risks associated with open source components, which organizations must identify, prioritize, and address. Of the three trust boundaries established by the SLSA framework, dependencies in open source code packages are by far the weakest link in the software supply chain. When an open source dependency integrated into your application is discovered to have a known vulnerability, you could try to manage the risk by using vulnerability databases and matching dependencies against alerts. But the better solution is to avoid incorporating compromised dependencies from the start.

When it comes to selecting open source code packages, don't take code from strangers. Developers cannot check everything manually. By using an automated, multi-phase analysis to gain visibility into the health of a code package, developers can select open source packages more wisely and code at speed.

Checkmarx SCA with Supply Chain Security sets a new standard for software composition analysis solutions. Without the innovative approach spearheaded by the Checkmarx SCS team, organizations have little if any visibility into the overall safety and potential risk of their open source supply chains.

To learn more about Checkmarx SCA with Supply Chain Security, please request a demo ***here***.

About Checkmarx

Checkmarx is constantly pushing the boundaries of Application Security Testing to make security seamless and simple for the world's developers while giving CISOs the confidence and control they need. As the AppSec testing leader, we provide the industry's most comprehensive solutions, giving development and security teams unparalleled accuracy, coverage, visibility, and guidance to reduce risk across all components of modern software – including proprietary code, open source, APIs, and Infrastructure as code. Over 1,675 customers, including 45% of the Fortune 50, trust our security technology, expert research, and global services to securely optimize development at speed and scale. For more information, visit our website, check out our blog, or follow us on LinkedIn.

Checkmarx at a Glance

1,675+
Customers in 70 countries

750
Employees in 25 countries

45%
of the Fortune 50 are customers

30+
Languages & frameworks

500k+
KICS downloads in 2021

The world runs on code. We secure it.