International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 10 Issue: 12 | Dec 2023 www.irjet.net p-ISSN: 2395-0072

OSSCR: A framework for detecting Software Supply Chain "Risks" in Open-source Software Packages

Jai Balani†, and Ashish Bijlani††

†Computer Society, IEEE, Santa Clara Valley, CA, USA
††Research Scientist, Atlanta, Georgia, USA

Abstract - Open Source Software (OSS) has become the de-facto standard way of developing digital products and services. Modern OSS is distributed and consumed as ready-to-use *packages* hosted on Package Registries. However, bad actors evidently leverage techniques such as account hijacking, and social engineering to inject purposefully harmful code (malware) in benign packages and carry out *software supply chain attacks*. Yet, there is no robust way to measure potential supply chain cyber risks in OSS packages. Developers today rely on public metrics such as GitHub stars and number of downloads to infer the security and maturity of the software.

This work presents, OSSCR, the first-ever framework to evaluate OSS packages and measure potential supply chain risks. Our framework is based on the study of hundreds of previously documented malware samples. Specifically, we identify several code as well as metadata "risky" attributes that make OSS packages vulnerable to such risks. We believe that OSSCR framework will be very valuable to software developers as well as security teams at organizations that evaluate OSS code before shipping their apps/services.

Key Words: CyberSecurity, Supply Chain, Open Source Software, Malware, PyPI,

Introduction

Open-source Software (OSS) is increasingly being used to develop modern digital products and services for their benefits such as reduced development cost and time to market. According to a recent report from Synopsys, 91% of commercial applications (apps) contain OSS components [53]. Today, any independent developer or organization can "supply" OSS by publishing their code as ready-to-use *packages* on a centralized web store, called *Package Manager* (PM). Popular PMs include, Node Package Manager (NPM) that hosts millions of JavaScript packages, and Python Packaging Index (PyPI) that hosts hundreds of thousands of Python packages.

Problem. Unfortunately, OSS packages receive little to no security vetting by the PM administrators. As a result, malicious actors embed less secure packages in the supply chain with purposefully harmful OSS code (malware), carrying out *software supply chain attacks*. Compared to vulnerabilities that are accidental programming bugs introduced in benign OSS code and may (or may not) be exploitable, malware contains intentionally harmful and stealthy code that poses a direct cyber threat. Thousands of OSS packages containing malware have been reported across NPM, PyPI, and RubyGems, which have been downloaded millions of times [20, 47]. Such attacks are both difficult to detect and highly damaging. The same malware may be adopted by thousands of developers and find its way in several apps, potentially compromising the privacy of millions of users. Today, developers must thoroughly analyze OSS packages, and avoid *risky* packages that may expose them to high levels of supply chain risks. Unfortunately, there exists no robust framework to measure OSS supply chain risks. Current practices include sourcing only mature, stable, popular, and reputable OSS packages, where such attributes are inferred from publicly available metrics, such as GitHub stars, package downloads, and software development activity [50]. However, such practices are riddled with limitations and inefficiencies. Vanity metrics such as GitHub stars and package downloads do not reveal true information about the security posture of packages. More importantly, an attacker-controlled bot can easily manipulate such metrics. Manual analysis of code can be time-consuming and error-prone.

This work proposes a data-driven security framework, called OSSCR[1] to measure and control the level of supply chain risks when sourcing OSS packages. The design of OSSCR is guided by our study of 651 malware samples of documented OSS

supply chain attacks. Specifically, we have empirically identified a number of risky code and metadata attributes that make a package vulnerable to supply chain attacks. For instance, we tag inactive or unmaintained packages that no longer receive security fixes. Inspired by Android app runtime permissions [42], OSSCR uses a permission-based security model to offer control and code transparency to developers. Packages that invoke sensitive operating system functionality such as file accesses combined with remote network communication are flagged as risky as this functionality could leak sensitive data.

OSSCR has been developed with a goal to assist developers in identifying and reviewing potential supply chain risks in OSS packages. Since the degree of perceived security risk from an untrusted OSS package depends on the specific security requirements, OSSCR can be customized according to the threat model of the user. For instance, a package with no communal reputation or new maintainer, may be perceived to pose greater security risks to some organizations such as financial institutions due to heavy regulations, compared to others who may be more willing to use such packages for the functionality offered. Given the volatile nature of the problem, providing customized and granular risk measurement is one of the goals of our framework.

Open-Source Software Supply Chain Attacks

This section first provides a brief background on OSS supply chain attacks, introducing the key terminology as well as various stakeholders, and then presents a summary of prior attacks, along with our findings on various attacking techniques and malicious behaviors.

Fig. 1. Various stakeholders and their relationships in the modern OSS supply chain. Developers adopt ready to-use OSS packages for their web apps/services by installing them from Package Managers (a.k.a. Registries) into their own development environment. Package Maintainers, who may be different from the original package author, provide feature updates and security bug fixes.

2.1 Background and key terminology

Modern web apps (and services) are developed using high-level runtime programming languages (e.g., JavaScript, Python) for their rich ecosystem of hundreds of thousands of third-party OSS packages that enable quick prototyping and development. For instance, today developers can reuse open-source Python web frameworks such as Django [23] and Flask [31] to provide the boilerplate code for their apps by adopting ready-to-use packages from PMs. In contrast, OSS source files are stored in public repositories on code hosting platforms such as GitHub [28], but may require configuration or compilation for reuse.

Given the ease of reuse, PMs have become a vital part of the modern software development process. Every OSS ecosystem has its own PM (a.k.a *Registry*), which is maintained by the community authorities For instance, PyPI, NPM, and RubyGems are popular PMs that host hundreds of thousands of Python, JavaScript, and Ruby packages, respectively. NPM hosts over 2.2 million packages [36], and hundreds of new package versions are released everyday. Developers adopt packages by downloading them from PMs and installing into their own development environment All PMs provide command line tools for developers to easily search, publish, download, and install packages, which enables easy OSS reuse (i.e., without configuring or compiling from sources). For example, PyPI Python packages are installed using the pip command line tool by typing the name of the required package, and specific release version string as needed.

A package may further reuse, directly or transitively, other third-party OSS packages (or *dependencies*) for its functionality. For example, both “eslint” and “electron”, two highly popular NPM packages, reuse over hundred other packages. Therefore, installation of a package recursively downloads and installs all its dependencies from PMs. As a result, today’s OSS is highly distributed in nature, with deeply nested *supply chains* that dramatically increase software attack surface. Note that for the scope of this analysis, we only consider runtime dependencies of a package that are needed for its runtime functionality (e.g., for production), and ignore development dependencies that may be needed for development and testing.

Packages are maintained (e.g., providing feature updates or security bug fixes) by *maintainer(s)*, who may be different from the original package *author* that transferred the ownership for regular maintenance. *contributors*, on the other hand, are collaborators, who provide source code contributions (e.g., features). Figure 1 shows various stakeholders in the modern OSS supply chain.

Problem. The widespread use of PMs has also made them a gainful target for bad actors to exploit. Today, any independent developer or organization can publish their packages on PMs. However, unlike mobile app stores (e.g., Google Play Store) that analyze apps for potential security and privacy issues, OSS packages receive a little to no such security vetting by PM administrators [16, 25, 32]. As a result, bad actors not only accidentally exploit programmatic bugs (vulnerabilities) in benign third-party OSS code, but also inject malware in less secure packages in the supply chain to carry out *software supply chain attacks*. For example, NPM eslint-scope [24] and RubyGems rest-client [39] packages with millions of weekly downloads were trojanized to steal developer account credentials and leave a Remote-Code-Execution (RCE) backdoor on web servers, respectively.

In 2018 alone over 100 malicious OSS packages were found that had received 600 million cumulative downloads. By August 2019, the number grew to over 300 [21]. Over thousands of malicious OSS packages have been reported as of January, 2022 [17, 21, 37, 38, 40, 51], and a number of them went undetected for over a year, contrary to the commonly held "many eyes'' belief about the security and quality of OSS projects that can effectively be stated as: given enough eyeballs, all bugs are shallow. While "many eyes" may still be true for the Linux kernel, not every modern OSS project has as big and active community around it.

Compared to security vulnerabilities that are accidentally introduced in benign OSS code and may be exploitable, supply chain attacks pose a direct and purposefully harmful cyber threat. Additionally, security vulnerabilities can typically be fixed by patching the buggy code or upgrading the software to the next version that fixes the bug. Whereas, malware, being intentionally harmful and stealthy, is highly damaging and difficult to detect. For instance, the same malware may be adopted by thousands of developers and find its way in several apps, potentially compromising the privacy of millions of users. The victims of supply chain attacks are developers and organizations that adopt OSS to build their software apps/services, and end users that install such compromised apps. End user protection tools (e.g., anti-virus) fail to detect security/privacy risks posed by such malware as the malicious logic (e.g., steal credit cards or account credentials) is planted into supposedly benign and trusted apps.

2.2 Malware analysis

To develop a comprehensive and effective risk measurement framework, it is important to build a thorough understanding of the attack vectors and advanced techniques that are employed to subvert the OSS supply chain. Therefore, we collected previously reported malware samples and carried out a systematic study of various (a) malicious code behaviors, (b) attack vectors, and (c) evasion techniques adopted by attackers. We obtained samples of malicious packages by requesting a dataset from researchers, who recently carried out similar studies [21]. Overall, we analyzed 651 malicious packages reported in PyPI, NPM, and RubyGems between January, 2018 and February, 2020. We briefly summarize various attacking techniques and malicious behaviors we found.

Malicious behaviors: We categorize various malicious behaviors into several dimensions of risks to the core security goals, namely the confidentiality (i.e., unauthorized access), integrity (i.e., unauthorized changes), and availability (e.g., inaccessibility for authorized use) of data. We briefly describe each of them below with examples.

(1) *Compromising data confidentiality.* We found a number of malicious packages that leak *or steal* data such as IP addresses, account credentials, and credit cards that may jeopardize the privacy of users. For example, eslint-scope [24], a NPM package with millions of weekly installs, was compromised to steal account credentials from developers,

(2) We also found packages that install *backdoor* or invoke attacker-controlled code. For example, a backdoor was injected into a popular PyPI package, called ssh-decorate, to exfiltrate users' SSH credentials to a remote server. Figure 2 shows the malicious code.

```
1 # attacker-controlled code
2 def log(data):
3         try:
4                 post = bytes(urlencode(data), 'utf-8')
5                 handler = urlopen("http://ssh-decorate.cf/index.php", post)
6                 res = handler.read().decode('utf-8')
7         except:
8                 pass
9
10 class Connection:
11         def connect(server, user, password, port, verbose, privateKeyFile):
12
13         ...
14
15         # backdoor installed
16         log({"server": server, "port": port, "pkey": pdata, "password": password, "user": user})
```

Fig. 2. A backdoor was injected into ssh-decorate to exfiltrate users' SSH credentials to a remote server.

(3) *Compromising data availability.* include packages that *sabotage* data (e.g., ransomware) [6], and even abuse compute resources for mining cryptocurrency [9], potentially causing Denial of Service.

Attack vectors: We found that attackers mainly propagate malware in the following three ways: typo-squatting, account hijacking, and social engineering. We briefly describe each one of them below with malware examples.

(1) *Account hijacking.* compromises the account of existing package maintainers through credential theft (e.g., weak or reused password) for injecting malware [19]. For instance, version 0.0.7 of strong_password, a popular Ruby package, was published with a RCE backdoor by hijacking the account of its maintainer. on RubyGems PM.

(2) Under *social engineering*, attackers exploit the collaborative nature of OSS projects and trick owners of inactive or unmaintained packages to transfer ownership with the intention of adding malware [14, 15, 35]. Sometime, after ownership transfer, attackers first publish a supposedly useful package, then modify it by adding malicious payload when their published package is adopted. Popular packages are typically targeted for adoption to maximize the potential reach of such attacks.

(3) Under *typo-squatting*, attackers publish new packages with names similar to existing popular packages, and exploit the inexperience and carelessness of developers (e.g., name typo) during package installation to "supply" malware [56]. For example, Python packages urllib and urlib3 (one lowercase 'L') impersonated popular urllib3 (two lowercase 'L') package to steal SSH keys [37].

Similarly, a Python package jeilyfish was reported in December, 2019 that impersonated a popular jellyfish (two lowercase 'L') to steal SSH keys, and went undetected for a year. We found typo-squatting to be the most common attack (64% of malware) as it does not require careful code injection in existing packages.

Injection techniques: Modern OSS packages enable install hooks, which can run custom code during the installation process (e.g., setup.py file in Python). Most malicious packages that we analyzed abuse such hooks to trigger the malicious behavior. Attackers also leverage dynamic code generation functionality (e.g., eval()) offered by modern runtime languages to download and execute malicious code. The latter is particularly used for programming languages that do not provide installation hooks (e.g., Ruby).

```
1 def _! begin yield rescue Exception end end _!{
2         Thread.new{ loop{ _!{
3                 sleep 900;
4         eval(open('https://pastebin.com/raw/5iNdELNX').read
```

```
5        )} }}
6 if Rails.env[0]=="p"}
```

Fig. 3. rest-client [39] malware abuses dynamic code generation to invoke malicious code, and uses multiple evasion techniques such as benign service, multi-stage payload, conditional logic, and target non latest version to hide.

Evasion techniques: Our analysis reveals that attackers use multiple sophisticated anti-analysis techniques to defeat detection. Here we list a few commonly used techniques.

1. Use of benign services to hide malicious code, and circumvent detection. For example, rest-client malware abuses pastebin.com service to host second-stage payload as shown in Figure 3.

2. Malicious logic is typically guarding with conditional checks that are only triggered under specific (and often narrow) circumstances. For instance, rest-client payload is only triggered if analysis is performed in production mode ("p" in line 6 in Figure 3).

3. Code obfuscation to hide from both manual and automatic inspection. fast-requests [34] uses randomization and base64 encoding as obfuscation techniques.

OSSCR

This work introduces OSSCR, a configurable security framework for developers to mitigate risks of software supply chain attacks when adopting untrusted and potentially malicious third-party OSS packages. In the simplest form, given an OSS package, OSSCR provides a check-list of package attributes for developers to review and renders a final verdict on the level of supply chain risks.

Note that the degree of perceived security risk from untrusted OSS code depends on the specific security requirements. For instance, a package that sends harmless analytics data (e.g., IP addresses) to a third-party server may be perceived to pose greater security risks to some organizations such as financial institutions due to heavy regulations, compared to others who may be more willing to use such packages for the functionality offered. Given the volatile nature of the problem, OSSCR risk levels can be configured to fit a custom threat model.

In this work, we propose a set of risky package code and metadata attributes that must be reviewed as a part of OSSCR framework. These attributes are common across most documented malware samples, and are empirically derived from our study of 651 such samples §2.2. Figure 4 provides an overview of the workflow. OSSCR evaluates each package attribute separately according to custom security requirements to provide partial answers. All partial decisions are then integrated to render the final verdict. This modular design allows us to be extensible and adaptable; that is, we can easily add support for evaluating additional attributes in isolation, and combine decisions from each module.

In the remainder of this work, we discuss each of the proposed package attributes, and highlight its importance in accessing software supply chain risks.

Risky attribute # 1: use of sensitive system functionality. Malicious activities are typically performed by invoking Operating System (OS) functionality (or APIs). For example, file system calls (e.g., read(), write()) are typically used to access private data stored on disk (e.g., SSH keys [13]), replace OS binaries or install new files under critical dirs (e.g., /bin), and even infect other packages [27]. Virtual file systems such as /proc and /sys also reveal sensitive system information.

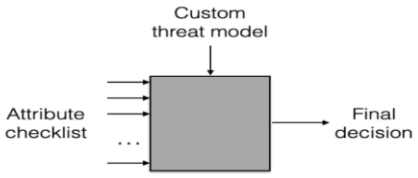

Fig. 4. Workflow of OSSCR. Given an OSS package, it provides a checklist of code and metadata attributes for developers to review, considers their custom security requirements, and renders a verdict on perceived software supply chain risk levels that the package may expose.

Permission	Description	Example: OS APIs	Example: Language APIs
FileRds	File reads	read()	JS: fs.readFileSync, Py: os.pread
FileWrts	File writes	write()	JS: fs.readFileSync, Py: os.pwrite
NetDwnlds	Network downloads	recv()	JS: https.get, Py: socket.recvfrom
NetUplds	Network uploads	send()	JS: https.get, Py: socket.send
ProcCreat	Process creation	fork()	JS: process.exec, Py: exec
CodeGen	Code generation	exec()	JS: vm.runInContext, Py: eval

Table 1. List of different types of sensitive functionality (APIs) we track. We analyze package code for Operating System (OS) as well as programming language APIs (e.g., Py: Python, Rb: Ruby, JS: JavaScript).

Similarly, network system calls (e.g., socket()) are used to communicate with a remote server for

downloading backdoor [39] at runtime or stealing data [13]. Process creation APIs (e.g., fork() on UNIX-based OSes) as well as code generation APIs provided by runtime languages (e.g., eval() in Python and Ruby) are abused to spawn hidden or background process [9] and load malicious payload at runtime [33, 39], respectively. As such, API usage profiles can reveal hidden malicious behavior.

To mitigate software supply chain risks, packages must be reviewed for code that directly access files, environment variables, or performs network operations. However, the same behavior could also be carried out indirectly by executing an external program (e.g., using system, fork()) or generating code at runtime (e.g., using eval in Python/Ruby) to isolate and hide malware [9]. Therefore, packages containing such indirect functionality must also be flagged.

Inspired by Android app runtime permissions [42], OSSCR uses a permission-based security model to offer code transparency to developers. Such permissions consolidate APIs according to their functionality type, and makes it easier to track the use of sensitive such programmatic attributes. Table 1 lists various characteristic permissions that we track and have empirically identified as "risky" based on our study of malware and benign samples.

Note that in this work, we do not focus on accessing sabotaging risks. However, packages could be analyzed for "dramatic" use of compute or storage resources for flagging sabotaging attempts. We leave it as a potential follow-up.

Risky attribute # 2: no public availability of the package source code. As mentioned in §2.1, Package Managers host ready-to-install packages of OSS projects, while their source code repositories are hosted on code hosting platforms such as GitHub [28]. As attackers target OSS packages to propagate malware, no traces of malicious activities are typically found in source repositories (e.g., on GitHub). This is because, unlike PMs, services like GitHub allow developers to explore source files, and any malicious code traces left in source repositories are likely to attract attention of developers and soon be removed. For instance, malicious version 0.0.7 of strong_password was published on RubyGems by hijacking maintainer account, but no such version sources were in its repository on GitHub [19].

Risky attribute # 3: no recent updates to the package. Packages that are unmaintained or abandoned packages no longer receive security fixes to known (n-day) vulnerabilities, and therefore, represent security weak links in the supply chain. Developers are also vulnerable to social engineering (e.g., ownership transfer) attacks against inactive and unmaintained packages [14, 15, 35]. Old and unmaintained packages are not automatically discarded by PMs if unused.

Risky attribute # 4: more packages with the same metadata from different authors. Attackers publish new packages with names similar to existing popular packages, and mount typo-squatting attacks [37]. Attackers also squat popular names across OSS ecosystems [3]. Typo-squatting is not a new technique; it has been leveraged to misdirect web users to malicious websites

PM admins do enforce package naming rules to combat typo-squatting [16, 25, 32]. For instance, RubyGems uses Levenshtein distance [?] to disallow new packages with names similar to popular packages [16]. PyPI replaces punctuation with hyphens when publishing packages, and handles all installation requests in a case-insensitive manner. NPM incorporates a typo-safe mechanism to allow similar package names, called scoped packages

Nevertheless, as these checks are enforced in real-time, they only cover a small subset of cases [43] to minimize performance overhead. They also result in high false positives [41]. Consequently, due to lack of sufficient checks during package submission, some organizations have adopted defensive typo-squatting to protect their users. They preemptively publish multiple *typo-guard* packages with similar names, and configure them to transparently serve intended package or alert users [8].

To detect typo-squatting packages, we must not only evaluate name similarity, but a set of metadata attributes based on the profile of typo-squatting packages identified in our study. The key observation is that these packages are published with the same project description, homepage, and software license to closely impersonate the target packages [1]. However, they differ from their targets in author information (e.g., email, username) and popularity (e.g., much fewer downloads and dependents). By relying on multiple attributes, we not only reduce errors, but also the likelihood of one

or more attacker-controlled attributes (e.g., popularity) gaming our system (e.g., using download bots).

Risky attribute # 5: packages with custom installation hooks. Modern OSS packages enable install hooks, which can run custom code during the installation process (e.g., setup.py file in Python). Most malicious packages that we analyzed abuse such hooks to trigger the malicious behavior. Attackers also leverage dynamic code generation functionality (e.g., eval()) offered by modern runtime languages to download and execute malicious code. The latter is particularly used for programming languages that do not provide installation hooks (e.g., Ruby).

Risky attribute # 6: packages maintainers with expired or incorrect email domains. Packages authored by developers with invalid domains, suggests lack of multi-factor authentication (e.g., 2FA). Packages from developers with expired email domains pose a serious supply chain threat. As email addresses of package developers are publicly displayed by all popular PMs, an attacker can easily track email addresses associated with expired domains. An expired domain can then be purchased and can purchase the domain by registering and modifying Domain Name Service (DNS) Mail Exchange (MX) entries. An attacker can hijack an expired email domain and take over the account associated with it. With such unauthorized access to the developer's email, the attacker can inject malicious code into the packages authored or maintained by the developer.

Discussion

Limitations

Through this study, we increase awareness and visibility in detecting risky packages to enhance supply chain security. We proposed and studied several risk attributes. However, malware detection in OSS packages, like the nature of most security problems, is going to be an arms race. That is, for every new tool or technique from security researchers, a more sophisticated technique will be used by attackers to evade detection. As such, there will always be room for growth and innovation to identify additional risky attributes. We have designed OSSCR as a modular security framework, and thus will be able to extend and incorporate additional inputs. We believe that our findings will help security researchers as well as PM admins in accessing package vulnerability and reducing supply chain risks.

Related Work

The following section compares prior work in the area of software supply chain attacks and the study of package managers to ours.

- **Study of package managers**. Much of the research in this area has encompassed measurement studies that report insights into various kinds of software supply chain attacks. They do not study package code or attributes and provide risk insights. Tschacher's experimental typo-squatting packages received 45,000 downloads from over 17,000 domains (including .gov), signifying implications of such attacks [56]. Zimmermann et al. [60] interdependent structure of packages and their trends in the NPM environment and showed similar results. This work proposes a framework to measure supply chain risks, and evaluates packages in all three popular PMs, namely NPM, PyPI, and RubyGems.

- **Software supply chain attacks**. The earliest software supply chain attack is the Thompson hack in 1983, in which he left a backdoor in the compiler, and could compromise a program even if its source code is benign. Following that, similar attacks [18, 45, 46, 58, 59] are launched, targeting various supply chain components such as infrastructure, operating systems, update channels, compilers and cryptographic algorithms. Recent years have witnessed an increasing trend of supply chain attacks targeting package managers [2, 12, 14, 15, 24, 39, 44, 54, 56], which host prebuilt packages for benefits such as code sharing.

- **Detection and mitigation**. Bertus [10] detects crypto-miners. [20, 47] provide a vetting pipeline and heuristics to detect attacks, respectively. [48] relies only on dynamic monitoring and static code analysis of package installation code for detection of attacks. In this work, we carried out a large scale measurement study using only static analysis of package code. In-toto framework [55] ensures code integrity by signing OSS packages, but cannot detect typo-squatting and social-engineering attacks (e.g., taking over package ownership), which we consider in OSSCR framework.

[29] analyzes OSS source code to identify malicious Git commits. Whereas, our work analyzes packaged OSS code available through Package Managers (e.g., NPM, PyPi), which were reportedly the targets of most attacks. Similarly, [7] focus on securing the build infrastructure, and not on attacks on PMs.

SLSA framework from Google [30] provides a security framework for developers for protecting against supply chain attacks. However, it focuses on securing the build infrastructure and artifact integrity.

OSSPolice [22] identified license violations and security issues with Open source software used in mobile apps. Similarly, LibScout [5] studied security issues with Java third-party libraries. Both the tools detect libraries and correlate them with existing vulnerability data to identify vulnerable ones. In contrast, this study focuses only on the study of "risky" and potentially malicious attributes of packages.

REFERENCES

[1] SK-CSIRT Advisory. 2017. Ten Malicious Libraries Found on PyPI - Python Package Index. http://www.nbu.gov.sk/ skcsirt-sa-20170909-pypi/

[2] Özkan Mustafa Akkuş. 2019. Defcon: Webmin 1.920 Unauthenticated Remote Command Execution. https://www. pentest.com.tr/exploits/DEFCON-Webmin-1920-Unauthenticated-Remote-Command-Execution.html

[3] Aladdin Almubayed. 2019. Practical Approach to Automate the Discovery and Eradication of Open-Source Software Vulnerabilities at Scale.

[4] Python Packaging Authority. 2022. Analyzing PyPI package downloads. https://packaging.python.org/en/latest/ guides/analyzing-pypi-package-downloads/

[5] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. ACM, Vienna, Austria.

[6] Adam Baldwin. 2019. The package destroyer-of-worlds contained malicious code. https://www.npmjs.com/advisories/ 890

[7] Len Bass, Ralph Holz, Paul Rimba, An Binh Tran, and Liming Zhu. 2015. Securing a Deployment Pipeline. In *Proceedings of the Third International Workshop on Release Engineering (RELENG '15)*. IEEE Press, 4âĂŞ7.

[8] William Bengtson. 2020. Python Typosquatting for Fun not Profit. https://medium.com/@williambengtson/python typosquatting-for-fun-not-profit-99869579c35d.

[9] Bertus. 2018. Cryptocurrency Clipboard Hijacker Discovered in PyPI Repository. https://medium.com/@bertusk/ cryptocurrency-clipboard-hijacker-discovered-in-pypi-repository-b66b8a534a8

[10] Bertus. 2018. Detecting Cyber Attacks in the Python Package Index (PyPI). https://medium.com/@bertusk/detecting cyber-attacks-in-the-python-package-index-pypi-61ab2b585c67

[11] Germán Méndez Bravo. 2018. ECMAScript parsing infrastructure for multipurpose analysis. https://github.com/ Kronuz/esprima-python

[12] Catalin Cimpanu. 2018. 17 Backdoored Docker Images Removed From Docker Hub. https://www.bleepingcomputer. com/news/security/17-backdoored-docker-images-removed-from-docker-hub/

[13] Catalin Cimpanu. 2018. Backdoored Python Library Caught Stealing SSH Credentials. https://www.bleepingcomputer. com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/

[14] Catalin Cimpanu. 2018. Hacker backdoors popular JavaScript library to steal Bitcoin funds. https://www.zdnet.com/ article/hacker-backdoors-popular-javascript-library-to-steal-bitcoin-funds/

[15] Catalin Cimpanu. 2018. Malware Found in Arch Linux AUR Package Repository. https://www.bleepingcomputer. com/news/security/malware-found-in-arch-linux-aur-package-repository/

[16] Jonathan Claudius. 2018. Establish a fixture to prevent gem typo attacks on Rubygems.org.

https://github.com/rubygems/rubygems.org/issues/1776.

[17] CNCF. 2022. Catalog of Supply Chain Compromises. https://github.com/cncf/tag-security/tree/main/supply-chain security/compromises.

[18] Jonathan Corbet. 2003. An attempt to backdoor the kernel. https://lwn.net/Articles/57135/

[19] Tute Costa. 2019. strong_password v0.0.7 rubygem hijacked. https://withatwist.dev/strong-password-rubygem hijacked.html

[20] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2020. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. *arXiv preprint arXiv:2002.01139* (2020).

[21] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. 2021. Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages. (Feb. 2021).

[22] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, Dallas, Texas, 2169–2185.

[23] Je" Forcier, Paul Bissex, and Wesley J Chun. 2008. *Python web development with Django*. Addison-Wesley Professional.

[24] JS Foundation and other contributors. 2018. Postmortem for Malicious Packages Published on July 12th, 2018. https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes

[25] Python Software Foundation. 2015. PEP 503 – Simple Repository API. https://www.python.org/dev/peps/pep-0503/.

[26] Python Software Foundation. 2019. The ast module helps Python applications to process trees of the Python abstract syntax grammar. https://docs.python.org/3/library/ast.html

[27] Harry Garrood. 2019. Malicious code in the purescript npm installer. https://harry.garrood.me/blog/malicious-code in-purescript-npm-installer/

[28] GitHub 2020. GitHub: The world's leading development platform. https://github.com

[29] Danielle Gonzalez, Thomas Zimmermann, Patrice Godefroid, and Max Schaefer. 2021. Anomalicious: Automated Detection of Anomalous and Potentially Malicious Commits on GitHub. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 258–267. https://doi.org/10.1109/ICSE-SEIP52600. 2021.00035

[30] Google. 2019. Safeguarding artifact integrity across any software supply chain. https://slsa.dev.

[31] Miguel Grinberg. 2018. *Flask web development: developing web applications with python*. " O'Reilly Media, Inc.".

[32] NPM Inc. 2017. New Package Moniker rules. https://blog.npmjs.org/post/168978377570/new-package-moniker-rules.

[33] NPM Inc. 2018. Reported malicious module: getcookies. https://blog.npmjs.org/post/173526807575/reported-malicious module-getcookies

[34] NPM Inc. 2019. All versions of fast-requests contain obfuscated malware that uploads Discord user tokens to a remote server. https://www.npmjs.com/advisories/1086

[35] NPM Inc. 2019. Plot to steal cryptocurrency foiled by the npm security team. https://blog.npmjs.org/post/185397814280/ plot-to-steal-cryptocurrency-foiled-by-the-npm

[36] NPM Inc. 2019. Tech talk: 1 million packages, plus learn how npm's security team saved the day! https://medium.com/npm-inc/npm-weekly-200-dont-miss-today-s-tech-talk-1-million-packages-plus-learn how-npm-s-security-f75f9882aebe

[37] Snyk Inc. 2017. Malicious Package | Affecting urllib package. https://snyk.io/vuln/SNYK-PYTHON-URLLIB-40671

[38] Snyk Inc. 2019. Malicious packages found to be typo-squatting in Python Package Index. https://snyk.io/blog/malicious packages-found-to-be-typo-squatting-in-pypi/.

[39] Jussi Koljonen. 2019. Warning! is rest-client 1.6.13 hijacked? https://github.com/rest-client/rest-client/issues/713

[40] IQT Labs. 2022. Software Supply Chain Compromises - A Living Dataset. https://github.com/IQTLabs/software supply-chain-compromises

[41] Peter Lejeck. 2019. Consider relaxing levenshtein distance rules. https://github.com/rubygems/rubygems.org/issues/2058.

[42] Google LLC. 2022. Permissions on Android. https://developer.android.com/guide/topics/permissions/overview

[43] Reed Loden. 2018. Malware in 'active-support' gem. https://hackerone.com/reports/392311

[44] Logix. 2018. Malware Found In The Ubuntu Snap Store. https://www.linuxuprising.com/2018/05/malware-found-in ubuntu-snap-store.html

[45] Lily Hay Newman. 2018. Inside the Unnerving Supply Chain Attack That Corrupted CCleaner. https://www.wired. com/story/inside-the-unnerving-supply-chain-attack-that-corrupted-ccleaner/

[46] Lily Hay Newman. 2019. Hack Brief: How to Check Your Computer for Asus Update Malware. https://www.wired. com/story/asus-software-update-hack/

[47] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. 2020. Backstabber's knife collection: A review of open source software supply chain attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 23–43.

[48] Marc Ohm, Arnold Sykosch, and Michael Meier. 2020. Towards Detection of Software Supply Chain Attacks by Forensic Artifacts. In *Proceedings of the 15th International Conference on Availability, Reliability and Security (ARES '20)*. Association for Computing Machinery, New York, NY, USA, Article 65, 6 pages. https://doi.org/10.1145/3407023.3409183

[49] Nikita Popov. 2019. A PHP parser written in PHP. https://github.com/nikic/PHP-Parser

[50] Huilian Sophie Qiu, Yucen Lily Li, Susmita Padala, Anita Sarma, and Bogdan Vasilescu. 2019. The signals that potential contributors look for when choosing open-source projects. *Proceedings of the ACM on Human-Computer Interaction* 3, CSCW (2019), 1–29.

[51] roscoe. 2018. PyPI Malware. https://github.com/rsc-dev/pypi_malware

[52] RubyGems 2019. Make 2FA mandatory for everyone who wants to publish gems to rubygems.org. https://github.com/rubygems/rubygems.org/issues/2104

[53] Inc. Synopsys. 2021. Synopsys Study Shows 91% of Commercial Applications Contain Outdated or Abandoned Open Source Components. https://www.securitymagazine.com/articles/92368-synopsys-study-shows-91-of-commercial applications-contain-outdated-or-abandoned-open-source-components

[54] Liran Tal. 2019. Malicious remote code execution backdoor discovered in the popular bootstrap-sass Ruby gem. https://snyk.io/blog/malicious-remote-code-execution-backdoor-discovered-in-the-popular-bootstrap-sass-ruby-gem/

[55] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. 2019. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1393–1410. https://www.usenix.org/conference/usenixsecurity19/presentation/torres arias

[56] Nikolai Philipp Tschacher. 2016. Typosquatting in programming language package managers. [57] whitequark. 2019. Parser is a production-ready Ruby parser written in pure Ruby. https://github.com/whitequark/parser

[58] Claud Xiao. 2015. Novel Malware XcodeGhost Modifies Xcode, Infects Apple iOS Apps and Hits App Store. https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps and-hits-app-store/

[59] Kim Zetter. 2015. Researchers Solve Juniper Backdoor Mystery; Signs Point to NSA. https://www.wired.com/2015/12/ researchers-solve-the-juniper-mystery-and-they-say-its-partially-the-nsas-fault/

[60] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *Proceedings of the 29th USENIX Security Symposium (Security)*. Austin, TX.