Backstabber's Knife Collection: A Review of Open Source Software Supply Chain Attacks

Marc Ohm[1(✉)], Henrik Plate[2], Arnold Sykosch[1,3], and Michael Meier[1,3]

[1] Institute for Computer Science 4, University of Bonn, Endenicher Allee 19A, 53115, Bonn, Germany
{ohm,sykosch,mm}@cs.uni-bonn.de
[2] SAP Security Research, SAP Labs France, 805 Av. Maurice Donat, 06250, Mougins, France
henrik.plate@sap.com
[3] Department for Cyber Security, Fraunhofer FKIE, Zanderstraße 5, 53177, Bonn, Germany

Abstract. A software supply chain attack is characterized by the injection of malicious code into a software package in order to compromise dependent systems further down the chain. Recent years saw a number of supply chain attacks that leverage the increasing use of open source during software development, which is facilitated by dependency managers that automatically resolve, download and install hundreds of open source packages throughout the software life cycle. Even though many approaches for detection and discovery of vulnerable packages exist, no prior work has focused on malicious packages. This paper presents a dataset as well as analysis of 174 malicious software packages that were used in real-world attacks on open source software supply chains and which were distributed via the popular package repositories npm, PyPI, and RubyGems. Those packages, dating from November 2015 to November 2019, were manually collected and analyzed. This work is meant to facilitate the future development of preventive and detective safeguards by open source and research communities.

Keywords: Application security · Malware · Software supply chain

1 Introduction

In general, software supply chain attacks aim to inject malicious code into a software product. Frequently, attackers tamper with the end product of a given vendor such that it carries a valid digital signature, as it is signed by the respective vendor, and may be obtained by end-users through trusted distribution channels, e.g. download or update sites.

A prominent example of such supply chain attacks is NotPetya, a ransomware concealed in a malicious update of a popular Ukrainian accounting software [8]. In 2017, NotPetya targeted Ukrainian companies but also hit global corporations, causing damage worth billions of dollars and is said to be one of the most

© Springer Nature Switzerland AG 2020
C. Maurice et al. (Eds.): DIMVA 2020, LNCS 12223, pp. 23–43, 2020.
https://doi.org/10.1007/978-3-030-52683-2_2

devastating cyberattacks known today [30]. In the same year, a malicious version of CCleaner, a popular maintenance tool for Microsoft Windows systems, was downloadable from the vendor's official website, and remained undetected for more than a month. During this period it was downloaded around 2.3 million times [27]. Another flavor of supply chain attacks aims at injecting the malicious code into a dependency of a software vendor's product. This attack vector was already predicted by Elias Levy in 2003 [29], and recent years saw a number of real-world attacks following that scheme. Such attacks become possible, because modern software projects commonly depend on multiple open source packages, which themselves introduce numerous transitive dependencies [2]. Such attacks abuse the developers' trust in the authenticity and integrity of packages hosted on commonly used servers and their adoption of automated build systems that encourage this practice [1].

A single open source package may be required by several thousands of open source software projects [23], which makes open source packages a very attractive target for software supply chain attacks. A recent attack on the npm package `event-stream` demonstrates the potential reach of such attacks: The alleged attacker was granted ownership of a prominent npm package simply by asking the original developer to take over its maintenance. At that time, `event-stream` was used by another 1,600 packages, and was on average downloaded 1.5 million times a week [22]. Open source software supply chain attacks are comparable to the problem of vulnerable open source packages which may pass their vulnerability to dependent software projects. This is known as one of the OWASP Top-10 application security risks [31]. However, in case of supply chain attacks, malicious code is deliberately injected and attackers employ obfuscation and evasion techniques to avoid detection by humans or program analysis tools.

The main contribution of this paper is the collection, categorization, and manual analysis of a dataset with malicious code from 174 packages that were used for real-world attacks on open source software supply chains between 2015 and 2019.

The remainder of the paper is structured as follows: Sect. 2 summarizes related work and Sect. 3 outlines the methodology used for the main contributions of this paper. Section 4 presents the necessary background on supply chain attacks, in particular two attack trees developed both on the basis of the dataset and by reviewing and investigating potential attacks and actual weaknesses of open source ecosystems. That is followed by Sect. 5, presenting the analysis and categorization of the actual code of 174 malicious packages observed in the wild. Section 6 summarizes and concludes the paper.

2 Related Work

Related work mostly covers *vulnerable* packages, which contain design flaws or code errors that are accidentally introduced, without bad intention but through negligence, and which may pose a potential security risk. In contrast to that, *malicious* packages contain design flaws or code errors that have been implemented selectively, with caution and the intention to be exploited or triggered

at later times in the software life cycle. Technically, malicious code and vulnerable code may look identical, the main difference lies in the intention of the developer (or lack thereof) and, in some cases, the use of evasion or obfuscation techniques to hinder the detection of such code.

Malicious and vulnerable packages reside in the same ecosystem and live through the same software life cycle. As such, related works that investigate package reuse in open source ecosystems in general, or the impact and spread of vulnerable packages in particular, also apply to malicious packages.

Decan, Mens, and Constantinou [13] leveraged security reports in order to examine how and when vulnerabilities in npm software packages are discovered and fixed. In order to assess the effect on other packages hosted on npm, a dependency graph was used. The key findings are that nearly half of the packages inherited vulnerabilities from other packages, and that version pinning to vulnerable and outdated packages are the main cause for such inherited vulnerabilities, even if fixes are available.

Zimmermann, Staicu, Tenny, and Pradel [40] were able to verify these findings and provide mitigation techniques. Highly popular packages and highly active developers were identified as single point of failures. Thus, the authors propose to raise developer awareness through training as well as automated code analysis.

Pfretzschner and Othmane [32] proposed a system to identify software supply chain attacks in npm packages by static code analysis. The tool is able to detect four kinds of attacks: Leakage of global variables, manipulation of global variables, local function manipulation, and dependency-tree manipulation. However, the authors failed to identify real-world examples of these attacks for evaluation.

Garrett, Ferreira, Jia, Sunshine, and Kästner [18] proposed anomaly detection through unsupervised learning in order to identify suspicious package updates. For that purpose they collected over 700,000 packages from npm and normal behavior was inferred from 1,500 randomly selected packages. The system reported 539 suspicious updates per week reducing manually inspection by 89%.

Jukka Ruohonen [33] examined vulnerable Python packages regarding their CVSS (Common Vulnerability Scoring System) score and the respective weakness according to CWE (Common Weakness Enumeration). An auto regressive model was used to calculate how likely a new release is vulnerable based on previous releases' vulnerability. It was found that the prediction of this event is difficult despite good statistical performance. However, the supply chain of a package was not taken into consideration.

While related work mostly focused on *vulnerable* packages and impact assessment with regard to *specific open source ecosystems*, especially Node.js (npm), this work considers *malicious* packages *across several ecosystems*.

3 Methodology

It is important to distinguish between vulnerable and malicious packages. As said, *vulnerable* packages may contain design flaws or code errors that are accidentally introduced, without bad intention but through negligence, and which

may pose a potential security risk. According to the Cambridge Dictionary *malicious* means "**intended** to cause damage to a computer system, or to steal private information from a computer system". Technically, malicious and vulnerable coding can be similar or even identical, thus, the main difference lies in the attacker's intention.

The main contribution of this paper is a dataset of malicious packages used in real-world attacks and their analysis. The analysis is detailed in Sect. 5 and comprises the subset of malicious packages used in real-world attacks for which the actual malicious code could be obtained (typically a compressed archive). Compilation took place between July 2nd and August 2nd, 2019 and was updated on 27th of January 2020. The dataset covers the programming languages JavaScript with its package repository npm, Java (Maven Central), Python (PyPI), PHP (Packagist) and Ruby (RubyGems), which are the most popular languages according to newly created GitHub repositories in 2018 [17].

During that time, the vulnerability database Snyk[1], security advisories[2,3,4], and research blogs (e.g. [3,4]) were reviewed to identify malicious packages and possible attack vectors. Only packages that are explicitly labeled as malicious are considered. Leaving out packages labeled as vulnerable might lead to missing some malicious packages. However, manually reviewing all vulnerable packages to find intention and hence prove maliciousness is infeasible. Likewise, the development of an automatized procedure is out of scope for this work but definitely desirable for future work.

Nonetheless, parts of the collection are automatized. This way no packages should be missed because of negligence or fatigue. A parser for the Snyk database is utilized to extract names, affected versions, and disclosure dates of packages listed as malicious. In the next step the publication of malicious versions of a package are dated according to Libraries.io[5], a service that monitors package releases across all major package repositories. Advisories and public incident reports are used to date the public disclosure of the malicious package.

Malicious packages are typically not available anymore on standard package repositories of the respective programming language, e.g. npm or PyPI. Thus, the script tries to download the affected version of a package from a PyPI[6], RubyGems[7], or npm[8] mirror. Failed attempts are manually checked for availability.

Collected packages are statically analyzed in a manual fashion. The package's metadata like name and publication/disclose date are analyzed to find out how it were injected into the ecosystem and how long it was available. The location of the suspicious code is found by manually looking through the package's code.

[1] https://snyk.io.
[2] https://www.npmjs.com/advisories.
[3] https://github.com/rubysec/rubygems-advisories.
[4] https://github.com/pypa/warehouse.
[5] https://libraries.io.
[6] https://nero-mirror.stanford.edu.
[7] https://mirror.auckland.ac.nz.
[8] https://registry.npm.taobao.org.

In-depth analysis is carried out to verify the maliciousness as well as to reveal the trigger and condition for malicious behavior, what its objective and targeted operating system (OS) is, and whether obfuscation was employed.

4 Background: Supply Chain Attacks

This background section starts with a high-level introduction of activities and systems related to open source software development projects in Sect. 4.1. Furthermore, different attack vectors for software supply chains will be presented with the help of two attack trees. In general, attack trees allow for a systematic description of attacks against any kind of system [34]. The root node of a given tree thereby corresponds to the attacker's top-level goal, and child nodes represent alternative ways to achieve it. The top-level goals of the attack trees presented in Sects. 4.2 and 4.3 are to inject malicious code into the software supply chain, thus, into a dependency of a development project, and to trigger that malicious code in different circumstances.

4.1 Open Source Development Projects

In a typical development environment as visualized in Fig. 1, *Maintainers* are members of a development project who administer the depicted systems, provide, review and approve contributions, or define and trigger build processes. Open source projects also receive code contributions from *contributors*, which may be reviewed and merged into the project's code base by maintainers. The *build process* ingests the source code and other resources of a project, and has the goal to produce software artifacts. These artifacts are subsequently published such that they become available to end-users and other development projects, either through to distribution platforms like app stores such that they may be consumed by end-users or to *package repositories* for other development projects.

The project resources reside in a *version control system* (VCS), e.g. Git, and are copied to the local file system of the *build system*. Among those resources is a declaration of direct dependencies, which is analyzed at the start of the build process by a *dependency manager* in order to establish the complete dependency tree with all direct and transitive dependencies. As all of them are required during the build, for instance, at compile time or during test execution, they are downloaded (pulled) from *package repositories* such as PyPI[9] for Python, npm[10] for Node.js, or Maven Central[11] for Java.

Such project environments are subject to numerous trust boundaries, and many threats target the respective data flows, data stores and processes. Managing those threats may be challenging even when considering only the environment of a single software project. When considering supply chains with dozens or hundreds of dependencies, it is important to notice that such an environment

[9] https://pypi.org.
[10] https://www.npmjs.com.
[11] https://search.maven.org/.

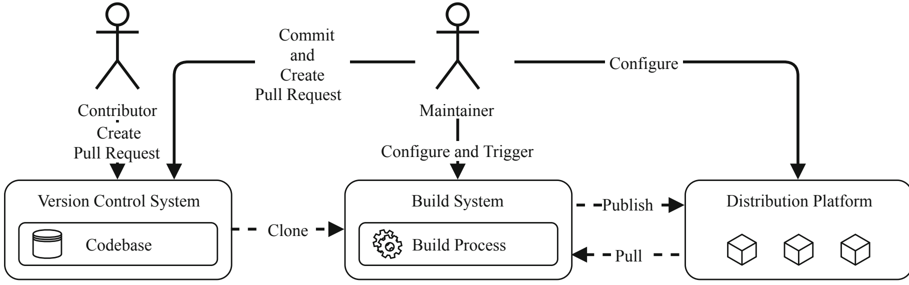

Fig. 1. High-level development and build activities.

exists for every single dependency, making it obvious that the combined attack surface of such projects is considerably larger than that of software entirely developed in-house.

Taking the perspective of attackers, malevolent actors have the intention to compromise the security of the build or runtime environment of software projects through the infection of one or more upstream open source packages, each one of which is developed in environments comparable to Fig. 1. How to reach this goal is described in the following sections by means of two attack trees that provide a structured overview about attack paths to inject a malicious code into dependency trees of downstream users and to trigger its execution at different times or under different conditions.

4.2 Injection of Malicious Code

The attack tree illustrated by Fig. 2 is an extension and refinement of the graph presented by Pfretzschner and Othmane [32], and has as top-level goal to inject malicious code into the dependency tree of downstream packages. Thus, the goal is satisfied once a package with malicious code is available on a distribution platform, e.g. package repository, and it became a direct or transitive dependency of one or more other packages.

To inject a package into dependency trees an attacker may follow two possible strategies, he may either *infect an existing package* or submit a *new package*.

Obviously, developing and publishing a new rogue package using a name that is not used by anybody else avoids interference with other legitimate project maintainers. However, such a package has to be discovered and referenced by downstream users in order to end up in the dependency trees of victim packages. This may be achieved using a name similar to existing package names (*typosquatting*) [3,4,14,15,35,36], or by developing and promoting a *trojan horse* [12]. An attacker might also use the opportunity to reuse the identifier of an existing project, package, or user account withdrawn by its original and legitimate maintainer (*use after free*) [10].

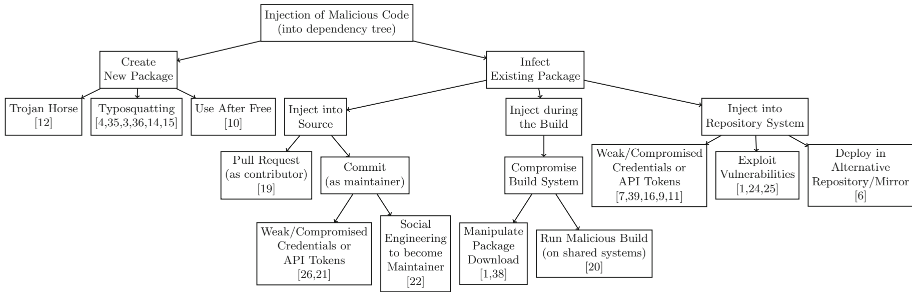

Fig. 2. Attack tree to inject malicious code into dependency trees.

The second strategy is to infect an *existing package* that already has users, contributors and maintainers. The attacker might choose packages for different reasons, e.g. a significant number or specific group of downstream users. Once the attacker chooses a package to infect, the malicious code may be injected *into the sources*, *during the build*, or *into the package repository*.

Open source projects live and strive through community contributions. Thus, attackers can mimic benign project contributors. For instance, an attacker may pretend to solve an existing issue by *creating a pull request* (PR) with a bug fix or a seemingly useful feature or dependency [19]. The latter could be used to create a dependency on an attacker-controller package created from scratch using the techniques described beforehand. In any case, this PR has to be approved and merged into the main code branch by a legitimate project maintainer. Alternatively, an attacker may *commit* malicious code into the project's code base all by himself by using *weak or compromised credentials* or security-sensitive API tokens [21,26]. Furthermore, attackers may become maintainer themselves through *social engineering* [22]. In all cases, no matter how the malicious code has been added to the sources, it will become part of an official package during the next release build—regardless where that build happens. Compared to attacks on build systems and package repositories, malicious code in VCS is more accessible to manual or automated reviews of commits or entire repositories.

The *compromise of build systems* typically entails tampering with resources used throughout the build process, e.g. compilers, build plugins or network services such as proxies or DNS servers. Such resources may be compromised if the build system, be it a developer's work station or a dedicated build server, is subject to vulnerabilities, or if insecure communication channels are such that attackers can *manipulate the package download* from repositories [1,38]. The release builds of the targeted package may also run on a shared build system and thus used by multiple projects [20]. Depending on the setup, such build processes may not run in isolation, hence resources such as package caches or build plugins are shared between the builds of different projects. In this case, an attacker may compromise shared resources during a *malicious build* of a project under his control such that the targeted project is compromised at a later point in time.

Even popular package repositories are still subject to simple but severe security vulnerabilities. While all the other attack vectors seek to inject malicious code into a single package, the *exploit of vulnerabilities* in package repositories themselves puts the entire repository with all its packages at risk [24,25]. Similar to injecting the code in the sources, the attacker may use *weak or compromised credentials* [7,9,11,16,39] or gain maintainer authorizations through *social engineering* [22] in order to publish malicious versions of legitimate packages.

Further, an attacker may upload malicious package versions to *alternative repositories or repository mirrors* [5,6] that are not provisioned by the original maintainers, and wait for victims pulling dependencies from there. Supposedly, such repositories and mirrors are less popular, and the attack is dependent on the victim's configuration, e.g. the order of repositories queried for dependencies or the use of mirrors.

4.3 Execution of Malicious Code

Once malicious code is present in a project's dependency tree, the attack tree illustrated by Fig. 3 has the top-level goal to trigger the malicious code under different conditions. Such conditions may be used to evade detection and/or target attacks towards specific users and systems.

Malicious code may trigger at different *life cycle phases* of the infected package and its downstream users (c.f. Sect. 5.3). If malicious code is contained in *test cases*, the attack primarily targets the contributors and maintainers of the infected package, which run such tests on their developer work stations and build systems. In many of the recorded attacks, malicious code is contained in *install scripts*, which are automatically executed during package installation by downstream users or their dependency managers. Such install scripts exist for Python and Node.js, and may be used to perform pre- or post-installation activities. Malicious code in install scripts puts the contributors and maintainers of downstream packages as well as their end users at risk. Malicious code may also be triggered at *runtime* of downstream packages, which requires that it is invoked as part of the regular control flow of the victim package. In Python, this may be achieved by including malicious code in `__init__.py`, which is invoked through *import* statements. In JavaScript, this may be achieved by monkey-patching (modifying) existing methods. The specifics of individual programming languages, package managers, etc. may easily be covered by refining this goal.

Independent of the life cycle phase, the execution of malicious behavior may always trigger (unconditioned) or only if certain conditions are met (*conditional execution*). As for any other malware, conditioned execution complicates the dynamic detection of malicious open source packages, since the respective conditions may not be known, understood or met in sandbox environments. Conditioning the execution on the *application state* is a common means to evade detection, e.g. in test environments or dedicated malware analysis sandboxes. Again, the specifics of individual build systems may be covered by respective sub goals, e.g. the presence of Jenkins environment variables indicates that malicious code is

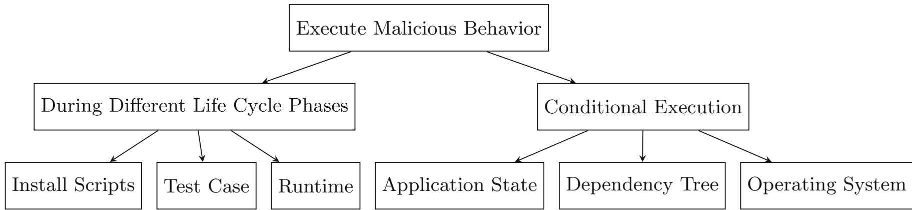

Fig. 3. Attack tree to execute malicious code.

triggered during a build rather than in a production environment. Moreover, conditions may be related to a specific victim package, e.g. check a specific application state such as the balance of a crypto wallet [22]. Heavy reuse of open source packages may lead to a malicious package ending up in the dependency tree of many downstream packages. If only certain packages are of interest to attackers, they may condition the code execution on the nodes of a given *dependency tree* at hand [22]. Furthermore, the *operating system* used may serve as condition.

5 Description of the Dataset

The dataset contains 174 packages and was compiled according to our methodology as described in Sect. 3. A total number of 469 malicious packages could be identified. Additionally, 59 packages were found that could be identified as proof of concept (published by researchers) and hence are excluded from further examination. Eventually, we were able to obtain at least one affected version for 174 packages. The rate of successful downloads of malicious packages for npm is 109/374 (29.14%), for PyPI 28/44 (63.64%), for RubyGems 37/41 (90.24%), and for Maven Central 0/10 (0.00%). All statements and statistics below refers to the set of downloaded packages as it is infeasible to infer characteristics from unobserved packages.

5.1 Composition and Structure

The dataset consists of 62.6% packages published on npm and hence are written for Node.js in JavaScript. The remaining packages were published via PyPI (16.1%, Python) and via RubyGems (21.3%, Ruby). Unfortunately, a malicious Java package targeting Android developers could not be downloaded. For PHP, we were not able to identify any malicious package at all.

The complete dataset is available for free on GitHub[12]. However, access will be granted on justified request only due to ethical reasons. The dataset is structured as follows: *package-manager/package-name/version/package.file*.

Malicious packages are grouped by their originating package manager on the first level. Further, multiple affected versions of one package are grouped

[12] https://dasfreak.github.io/Backstabbers-Knife-Collection.

under the respective package's name. As example for the affected version of the well-known case of event-stream it is: *npm/event-stream/3.3.6/event-stream-3.3.6.tgz*.

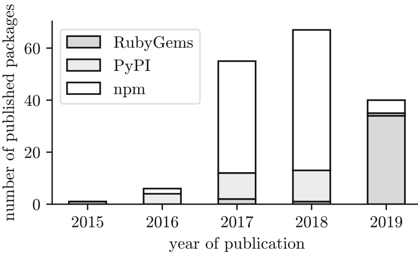

Fig. 4. Publication dates of collected packages.

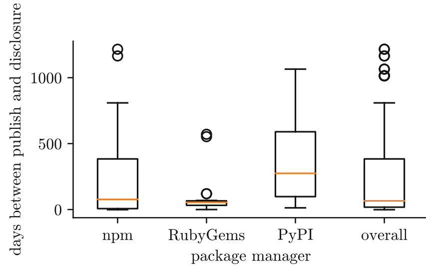

Fig. 5. Temporal distance between date of publication and disclosure.

5.2 Temporal Aspects

Figure 4 visualizes the publication dates of the collected packages which range from November 2015 to November 2019. The publication and disclosure dates are identified according to the upload time of the package and the publication date of the corresponding advisory identifying the respective version as malicious (cf. Sect. 3). A trend for an increasing number of published malicious packages is apparent. While malicious packages for PyPI are known to date back to 2015 and since then are increasing, npm gained a massive amount of malicious packages in 2017. Malicious packages on RubyGems experienced a boom in 2019.

Note that there are more incidents in total than Fig. 4 references, as it does not include reported malicious packages that we could not obtain. PyPI and npm show an ever-increasing trend as they can easily be used to spread malicious code due to their package managers' ability to execute arbitrary code on installation (c.f. Sect. 5.3). In contrast to that, RubyGems does not allow code execution on install but seems to be targeted more often in recent attacks. This might be due to PyPI's and npm's increasing efforts to hinder attackers from abusing their package repositories and managers, respectively.

Figure 5 shows that **on average a malicious package is available for 209 days** ($min = -1, max = 1,216, \sigma = 258, \tilde{x} = 67$) before being publicly reported. A minimum of -1 days was reached for *npm/eslint-config-airbnb-standard/2.1.1* which was affected by *npm/eslint-scope/3.7.2*. Even though the infection of *npm/eslint-scope/3.7.2* was known, the package was still in use due to the developers' re-packaging strategy, i.e. the infected version was hard copied into the source of *npm/eslint-config-airbnb-standard/2.1.1*. The maximum of 1,216 days was reached by *npm/rpc-websocket/0.7.7* which took over an abandoned package and went undetected for a long period.

In general this shows that packages tend to be available for a longer period. While PyPI has the highest average online time, that period varies the most for npm, and RubyGems tends to detect malicious packages more timely.

5.3 Trigger of Malicious Behavior

Malicious behavior of a package may be triggered at different points of interaction with the package. Most typically, a package may be installed, tested, or executed. A separation per package repository is visualized in Fig. 6.

It is apparent that **most malicious packages (56%) start their routines on installation**, which might be due to poor handling of arbitrary code during install. This can be triggered by the package repositories' install command, e.g. `npm install <package>`, which invokes code as defined in the package's definition, e.g. `package.json` and `setup.py`. This code might be arbitrary to do whatever is necessary to install the package, e.g. download additional files. It is by far the easiest way for attackers to effectively activate their malicious code and hence used frequently. This seems very common for malicious packages on PyPI. The difference for nmp and PyPI might stem from npm packages having more dependencies than a typical Python package [37] which might lead to more malicious packages targeting other dependent package on runtime like in the case of `event-stream` [22].

In contrast, Ruby does not implement such install logic and hence no packages for that case exist in Ruby. Consequentially, all found packages on RubyGems use runtime as trigger, often targeting Ruby on Rails, a server-side web application framework. Overall, 43% of the packages expose their malicious behavior during the program's runtime, i.e. when invoked from another function.

For 1% of the packages the test routines are used as trigger. Invoking the test routine of *npm/ladder-text-js/1.0.0* would execute `sudo rm -rf /*` which, if successful, deletes all the user's files. Note that this observation might not generalize due to the low number of found packages using this technique.

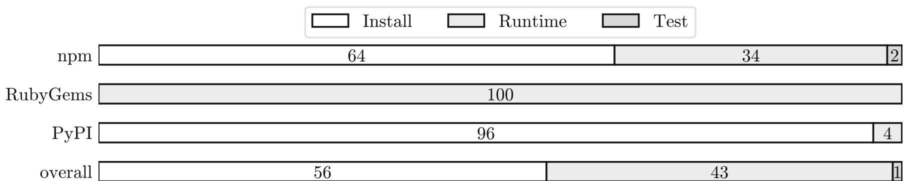

Fig. 6. Trigger of malicious behavior separated per package repository and overall.

5.4 Conditional Execution

As seen in Fig. 7, **41% of the packages check for a condition before triggering further execution**. This may depend on the application's state, e.g. check whether the main application is in production

Fig. 7. Ratio of conditional and unconditional execution per package repository and overall.

mode (e.g. *RubyGems/paranoid2/1.1.6*), resolvability of a domain name (e.g. *npm/logsymbles/2.2.0*), or the amount contained in a crypto wallet (e.g. *npm/flatmap-stream/0.1.1*). This may be done to find profitable targets and evade sandboxing (dynamic analysis).

Other techniques are to check whether another package is present in the dependency tree (e.g. *npm/load-from-cwd-or-npm/3.0.2*) or whether the package is executed on a certain OS (e.g. *PyPI/libpeshka/0.6*). This is done to either target another package or because the malicious functions rely on OS characteristics and functions.

The majority of packages published on PyPI and RubyGems execute unconditionally. For npm the ratio of conditional and unconditional execution is nearly equal. However, packages from PyPI seem not to use Application State as condition which might be due to Python not being used on server-side – unlike npm and Ruby (on Rails) – very often.

5.5 Injection of Malicious Package

In Fig. 8 it is apparent that **most (61%) malicious packages mimic existing packages' names via typosquatting**. A deeper analysis of that phenomenon revealed that the Levenshtein distance of an average typosquatting package to its target is 2.3 ($min = 0, max = 11, \sigma = 2.05, \tilde{x} = 1.0$). In some cases the typosquatting target is available from another package repository, e.g. the Linux package repository `apt` under the exact same name. This is for instance the case for *python-sqlite*. The maximum distance of 11 is reached in the case of *pythonkafka* which targeted *kafka-python*. Common techniques are adding or removing hyphens, leaving out single letters, or exchange of letters that are often mistyped. A word that is targeted exceptionally often is "color" or, respectively, its British English counterpart: "colour". Typosquatting is already proven to be a highly effective technique to infect large numbers of victims in short time [36].

The second most common injection method was the infection of an existing package. This may often be achieved with *compromised credentials* for the repository system (e.g. *npm/eslint-scope/3.7.2*). In most cases, the exact infection technique could not be determined in retrospect. This is because the related source is often removed from the version control system or no further details

about the injection are made public. Hence, these packages are listed as *infect existing package*. This technique requires more work from the attackers point of view as he has to take over a developer's/publisher's account first. Once that is accomplished, an update containing malicious content can easily reach numerous users as they are already using that package and depend on its functionality. It is especially dangerous if no version pinning is used and dependencies are updated automatically.

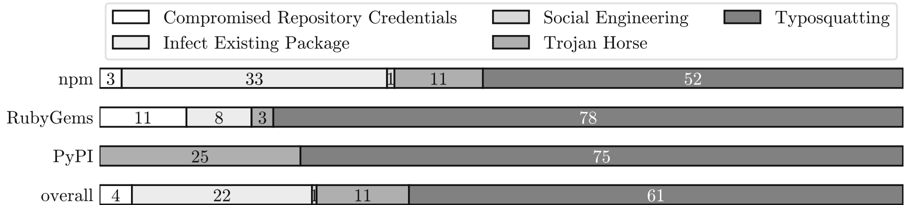

Fig. 8. Injection technique used to introduce the malicious package into a package per package repository and overall.

Another injection technique is to create a new package which consist of nothing but the malicious package to which we refer to as *trojan horse*. No meaningful typo-squatting targets were found for these packages. These packages might have been around as preparation for further attacks to be used in conjunction with an infected existing package or standalone.

5.6 Primary Objective

As shown in Fig. 9, **most packages aim at data exfiltration**. Commonly, the data of interest is the content of `/etc/passwd`, `~/.ssh/*`, `~/.npmrc`, or `~/.bash_history`. Furthermore, malicious packages try to exfiltrate environment variables (which might contain access tokens) and general system information. Another popular target (7 reported packages, 3 of them available in our dataset) is the token for the voice and text chat application Discord. A Discord user's account may be linked to credit card information and thus be used for financial fraudulence. Exfiltrated data – especially access tokens – may be used for further attacks and spreading of the malicious code [28]. Credit card information may be used for financial fraud.

Moreover, 34% of the packages function as dropper to download second stage payload. Another 5% open a backdoor, i.e. reverse shell, to a remote server and await further instructions. This category will turn victims into zombies that can be controlled by the attacker, e.g. for DDoS attacks. 3% aim to cause a denial of service by exhausting resources through fork bombs and file deletion (e.g. *npm/destroyer-of-worlds/1.0.0*) or breaking functionality of other packages (e.g. *npm/load-from-cwd-or-npm/3.0.2*). This only yields gain for an attacker if

a competing party is attacked. Only 3% have financial gain as primary objective by for instance running a cryptominer in the background (e.g. *npm/hooka-tools/1.0.0*) or stealing cryptocurrency directly (e.g. *pip/colourama/0.1.6*). In addition, combinations of the above mentioned objectives might occur.

5.7 Targeted Operating System

In order to identify the targeted OS, the source code was manually analyzed for hints which may be as explicit as an if–then construct like `if platform.system() is 'Windows'` as used in e.g. *PyPI/openvc/1.0.0* or implicit by relying on resources only available on certain OS. These resources may be for instance files containing sensible information like `.bashrc` etc. (cf. Sect. 5.6, *npm/font-scrubber/1.2.2*) or executables like `/bin/sh` (e.g. *npm/rpc-websocket/0.7.11*).

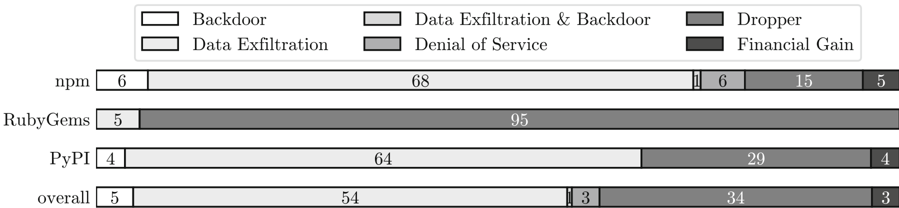

Fig. 9. Primary objective of the malicious package per package repository and overall.

The analysis of the packages for their targeted OS as shown in Fig. 10 revealed that **most packages (53%) are agnostic, i.e. do not rely on OS-specific functions**. The analysis was done on the initial visible code of the package and thus the targeted OS of the second stage payload remains unknown. However, Unix-like systems seem to be targeted more often than Windows and macOS. This might be due to Unix-like systems being used as build environments and hence more valuable data like access tokens (c.f. Sect. 5.6) may be accessible.

There is only one known case of macOS being the target in which the package *npm/angluar-cli/0.0.1* performs a denial of service attack on the McAfee virus scanner for macOS by deleting and modifying its files.

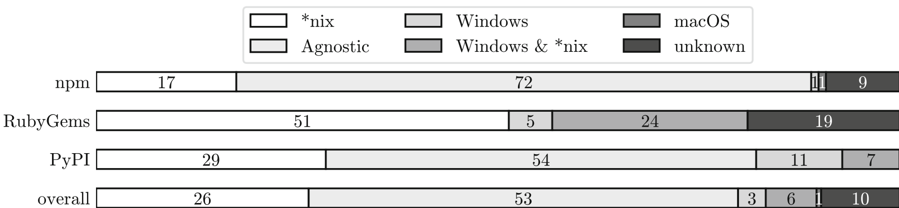

Fig. 10. Targeted operating system per package repository and overall.

5.8 Obfuscation

Malicious actors often try to disguise the presence of their code, i.e. hindering its detection by sight. Noticeable in Fig. 11 is that **nearly the half of the packages (49%) employ some kind of obfuscation**. Most often a different encoding (Base64 or Hex) is used to disguise the presence of malicious functions or suspicious variables such as domain names. This is an easy and effective way to go since most languages have these capabilities on-board without external dependencies.

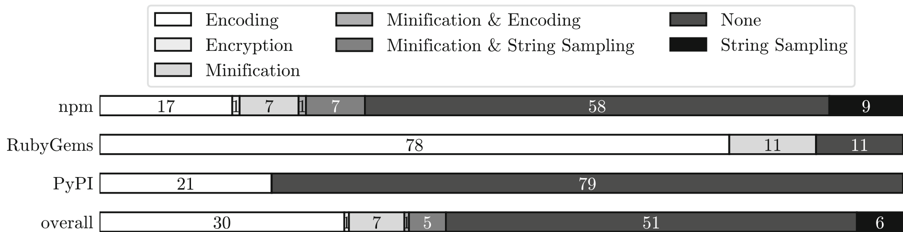

Fig. 11. Employed obfuscation technique per package repository and overall.

A technique often used by benign packages to compress source code and thus save bandwidth is minification. However, this is a welcome opportunity for malicious actors to sneak in extra code which is unreadable for humans (e.g. *npm/tensorplow/1.0.0*). Another way to hide variables is to use string sampling. This requires a seemingly random string which is used to rebuild meaningful strings by picking letter by letter (e.g. *npm/ember-power-timepicker/1.0.8*).

In one case the malicious functions are hidden by encryption. The package *npm/flatmap-stream/0.1.1* leverages AES256 with the short description of the targeted package as decryption key. That way, the malicious behavior is solely exposed when used by the targeted package. Furthermore, combinations of the above mentioned techniques exist.

5.9 Clusters

In order to infer on the presence of attack campaigns, all packages were analyzed for reuse of malicious code or dependency relationships. The malicious code snippets that were manually identified were compared visually for similarity. This way, **it was possible to identify 21 clusters** for which at least two packages either have similar malicious code in common, or an attacker-controlled package depends on another one with the actual malicious code. In total, 157 of the 174 packages (90%) belong to a cluster. On average a cluster comprises 7.28 packages ($min = 2, max = 36, \sigma = 8.96, \tilde{x} = 3$).

A cross comparison of publications dates of packages within one cluster revealed that the average temporal distance between publications is 42 days,

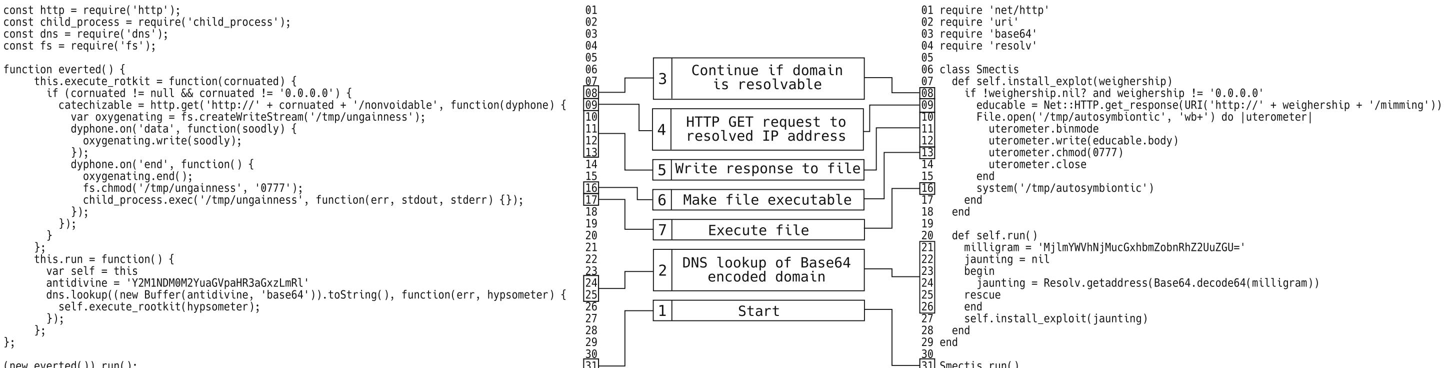

Fig. 12. Comparison and explanation of two malicious packages: *npm/jqeury/3.3.1* on the left; *RubyGems/active-support/5.2.0* on the right. The code was found in `ext/trellislike/unflaming/waffling/extconf.rb` and `package/src/data/var/everted.js`, respectively. Names of classes, variables, and function are randomly picked from an English dictionary (different for each package of the cluster). After instantiation (1), a DNS lookup (2) for an obfuscated (Base64) domain is started. Both snippet contain code for conditional execution. If the lookup is either empty or the non-routable meta-address `0.0.0.0` (3), the execution of malicious code is skipped. This may be seen as evasion technique as it may be the case for sandboxed environments as used for dynamic malware analysis. However, if the domain can be resolved, a HTTP GET request is send (4). The response is written to new a file located in `/tmp` (5, 6) and executed (7).

6:50:18 (min = 1:29:40, max = 353 days, 11:17:02, σ = 78 days, 0:43:10, $\tilde{x}$ = 7 days, 15:24:51). The biggest cluster was formed around the `crossenv` case [35] counting 36 packages published with an average temporal distance of 5.98 days. It was published in two waves, 11 packages within 15 min on 19th of July 2017 and another 25 packages within 30 min on 1st of August 2017.

The cluster having publication dates that are 353 days apart consists of the two packages *PyPI/jeilyfish/0.7.0* and *PyPI/python3-dateutil/2.9.1*. The first was published on 12/11/18 12:26 AM and contained code that download a script to steal SSH and GPG Keys from Windows machines. It went undetected for a long time until the second package was published on 11/29/19 11:43 AM which did not contain malicious code itself but referenced the first package. The cluster was reported and deleted on 12/12/19 05:53 PM.

While most clusters solely contain packages from one package repository, it was possible to find a cluster that mainly contained packages from npm but also *RubyGems/active-support/5.2.0* from RubyGems. This means that attack campaigns exist or at least techniques flow across multiple package repositories.

5.10 Code Review of Two Malicious Packages

For vivid illustration, *npm/jqeury/3.3.1* (left) and *RubyGems/active-support/5.2.0* (right) will be discussed in Fig. 12. They both belong to the same cluster according to our manual assessment of code similarity, even though they were published on different repositories.

6 Discussion and Conclusions

From an attacker's point of view, package repositories represent a reliable and scalable malware distribution channel. We were able to create the first manually curated dataset of malicious open source packages that have been used in real-world attacks. It consists of 174 malicious packages (62.6% npm, 16.1% PyPI, 21.3% RubyGems) ranging from November 2015 to November 2019. Manual analysis revealed that most packages (56%) trigger their malicious behavior on installation, and 41% use further conditions to determine whether to run or not. More than half of the packages (61%) leverage typosquatting to inject themselves into the ecosystem, and data exfiltration is the most common goal (55%). The packages typically are agnostic to operating systems (53%), and often employ obfuscations (49%) to hide themselves. Finally, we were able to detect multiple clusters of malicious packages through reused code even across different programming languages. The dataset provides insight and is available for free to facilitate research in the area of prevention, detection, and mitigation of software supply chain attacks.

However, there are some limitations. Our dataset is highly biased towards malicious packages that are written in JavaScript for Node.js and published on npm which is due to npm's enormous size and popularity. Unfortunately, we were not able to obtain malicious packages for Java (Maven Central) and PHP

(Packagist). Furthermore, roughly 34% of the malicious packages are droppers with the goal to download a second stage payload, which might not be available anymore. One might notice that we listed the deployment in alternative repository or mirrors as injection method but downloaded most of the packages from such sources. While it is possible that these packages have been altered to be malicious, the package's presence in our dataset is still valid as the package would be malicious is both cases. Furthermore, the "intended" maliciousness according to the advisories was verified through manual analysis. Leaving out packages labeled as vulnerable might lead to missing some malicious packages. However, automated detection of maliciousness is out of scope of this work but up for future work. One possible approach for applying the lessons learned from our manual code review could be to identify common control or data flow patterns in malicious code, e.g., silenced exceptions, and search for their presence in other packages.

Our analysis shows that it is important to make use of already available security means. To tackle the most prominent trigger – arbitrary code execution during installation – package managers need to be reworked. Python, for instance, already offers Python Wheels,[13] which avoids code execution during installation. We offer two recommendations for dealing with existing infected packages. For maintainers, multi-factor authentication and strong passwords should be mandatory. Developers should use version pinning. However, the version needs to be chosen absolute, i.e. no automated security patches or bug fixes (minor updates) which again may be counterproductive when it comes to vulnerabilities. Typosquatting packages are already being frequently purged by common package repositories but nonetheless make it through often. General awareness of developers and more stringent rules from the package repositories may help against that type of attack.

However, now that a dataset exists it is possible to use proven malicious packages as seeds in order to find more related cases (c.f. Sect. 5.9). In this context, the manually curated and labeled dataset allows for supervised learning approaches that support the automated and repository-wide search for malicious packages. Moreover, with regard to existing and new mitigation strategies, the presented dataset may pose as a benchmark. Last, acknowledging the importance of a comprehensive and up-to-date dataset, it will be necessary to continue its curation – contributions are welcome.

Acknowlegements. This work is funded under the SPARTA project, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 830892.

[13] https://pythonwheels.com/.

References

1. Chess, B., Lee, F.D.Q., West, J.: Attacking the build through cross-build injection: how your build process can open the gates to a trojan horse. https://www.fortify.com/downloads2/public/fortify_attacking_the_build.pdf (2007). Accessed 06 Mar 2019
2. Baker, G.: Keep your dependencies secure and up-to-date with GitHub and Dependabot (2019). https://github.blog/2019-01-31-keep-your-dependencies-secure-and-up-to-date-with-github-and-dependabot/. Accessed 08 Oct 2019
3. Bertus: Cryptocurrency clipboard hijacker discovered in PyPi repository (2018). https://medium.com/@bertusk/cryptocurrency-clipboard-hijacker-discovered-in-pypi-repository-b66b8a534a8. Accessed 09 Mar 2019
4. Bertus: Discord token stealer discovered in PyPi repository (2019). https://medium.com/@bertusk/discord-token-stealer-discovered-in-pypi-repository-e65ed9c3de06. Accessed 02 July 2019
5. Bintray: Malicious packages reported in JCenter (2017). https://status.bintray.com/incidents/w4dfr0rpznkt. Accessed 14 Mar 2019
6. Braun, M.: A confusing dependency (2018). https://blog.autsoft.hu/a-confusing-dependency/. Accessed 14 Mar 2019
7. ChALkeR: Gathering weak npm credentials (2017). https://github.com/ChALkeR/notes/blob/master/Gathering-weak-npm-credentials.md. Accessed 10 Mar 2019
8. Cimpanu, C.: Petya ransomware outbreak originated in Ukraine via tainted accounting software (2017). https://www.bleepingcomputer.com/news/security/petya-ransomware-outbreak-originated-in-ukraine-via-tainted-accounting-software/. Accessed 24 Feb 2019
9. Cimpanu, C.: Backdoored python library caught stealing SSH credentials (2018). https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/. Accessed 10 Mar 2019
10. Claburn, T.: You can resurrect any deleted GitHub account name (2018). https://www.theregister.co.uk/2018/02/10/github_account_name_reuse/. Accessed 06 Mar 2019
11. Coe, B.E.: A core contributor to the conventional-changelog ecosystem had their npm credentials compromised (2018). https://github.com/conventional-changelog/conventional-changelog/issues/282#issuecomment-365367804. Accessed 17 Feb 2020
12. Constantin, L.: Npm attackers sneak a backdoor into node.js deployments through dependencies (2018). https://thenewstack.io/npm-attackers-sneak-a-backdoor-into-node-js-deployments-through-dependencies/. Accessed 06 Mar 2019
13. Decan, A., Mens, T., Constantinou, E.: On the impact of security vulnerabilities in the npm package dependency network. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp. 181–191. IEEE (2018)
14. Denbraver, H.: Malicious packages found to be typo-squatting in python package index (2019). https://snyk.io/blog/malicious-packages-found-to-be-typo-squatting-in-pypi/. Accessed 17 Feb 2020
15. Dunn, J.E.: Pypi python repository hit by typosquatting sneak attack (2017). https://nakedsecurity.sophos.com/2017/09/19/pypi-python-repository-hit-by-typosquatting-sneak-attack/. Accessed 17 Feb 2020
16. Edge, J.: A backdoor in a popular ruby gem (2019). https://lwn.net/Articles/785386/. Accessed 17 Feb 2020

17. Elliott, T.: The state of the octoverse: top programming languages of 2018, November 2018. https://github.blog/2018-11-15-state-of-the-octoverse-top-programming-languages/. Accessed 30 Sept 2019
18. Garrett, K., Ferreira, G., Jia, L., Sunshine, J., Kästner, C.: Detecting suspicious package updates. In: Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, pp. 13–16. IEEE Press (2019)
19. Gilbertson, D.: I'm harvesting credit card numbers and passwords from your site. Here's how (2018). https://hackernoon.com/im-harvesting-credit-card-numbers-and-passwords-from-your-site-here-s-how-9a8cb347c5b5. Accessed 09 Nov 2018
20. Gruhn, V., Hannebauer, C., John, C.: Security of public continuous integration services. In: Proceedings of the 9th International Symposium on Open Collaboration, WikiSym 2013, pp. 15:1–15:10. ACM, New York (2013)
21. Holmes, E.: How I gained commit access to homebrew in 30 minutes (2018). https://medium.com/@vesirin/how-i-gained-commit-access-to-homebrew-in-30-minutes-2ae314df03ab. Accessed 06 Mar 2019
22. II, T.H.: Compromised npm package: event-stream (2018). https://medium.com/intrinsic/compromised-npm-package-event-stream-d47d08605502. Accessed 06 Mar 2019
23. Janaszek, M.: State of package.json dependencies (2018). https://medium.com/warsawjs/state-of-package-json-dependencies-de99828b6c3f. Accessed 08 Oct 2019
24. Justicz, M.: Remote code execution on rubygems.org (2017). https://justi.cz/security/2017/10/07/rubygems-org-rce.html. Accessed 06 Mar 2019
25. Justicz, M.: Remote code execution on packagist.org (2018). https://justi.cz/security/2018/08/28/packagist-org-rce.html. Accessed 7 Oct 2019
26. Khandelwal, S.: Password-guessing was used to hack Gentoo Linux Github account (2019). https://thehackernews.com/2018/07/github-hacking-gentoo-linux.html. Accessed 07 Oct 2019
27. Khandelwal, S.: CCleaner attack timeline - here's how hackers infected 2.3 million PCs (2018). https://thehackernews.com/2018/04/ccleaner-malware-attack.html. Accessed 24 Feb 2019
28. Kuizinas, G.: State of package.json dependencies (2017). https://medium.com/@gajus/distributing-a-self-replicating-malicious-code-using-npm-cf2bf3209293. Accessed 14 Apr 2020
29. Levy, E.: Poisoning the software supply chain. IEEE Secur. Priv. **1**(3), 70–73 (2003)
30. Ng, A.: Us: Russia's NotPetya the most destructive cyberattack ever (2018). https://www.cnet.com/news/uk-said-russia-is-behind-destructive-2017-cyberattack-in-ukraine/. Accessed 25 Feb 2019
31. OWASP: Owasp top 10: the ten most critical web application security risks (2017). https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf. Accessed 06 Mar 2019
32. Pfretzschner, B., ben Othmane, L.: Identification of dependency-based attacks on node.js. In: Proceedings of the 12th International Conference on Availability, Reliability and Security, p. 68. ACM (2017)
33. Ruohonen, J.: An empirical analysis of vulnerabilities in python packages for web applications. In: 2018 9th International Workshop on Empirical Software Engineering in Practice (IWESEP), pp. 25–30. IEEE (2018)
34. Schneier, B.: Attack trees, December 1999. https://www.schneier.com/academic/archives/1999/12/attack_trees.html. Accessed 14 Apr 2020
35. Spring, T.: Attackers use typo-squatting to steal npm credentials (2017). https://threatpost.com/attackers-use-typo-squatting-to-steal-npm-credentials/127235/. Accessed 06 Mar 2019

36. Tschacher, N.P.: Typosquatting in programming language package managers. Master's thesis, Universität Hamburg, Fachbereich Informatik (2016)
37. Vaidya, R.K., De Carli, L., Davidson, D., Rastogi, V.: Security issues in language-based sofware ecosystems. arXiv preprint arXiv:1903.02613 (2019)
38. Veytsman, M.: How to take over the computer of any Java (or Clojure or Scala) developer (2014). http://blog.ontoillogical.com/blog/2014/07/28/how-to-take-over-any-java-developer/. Accessed 14 Mar 2019
39. Zhu, H., Mills, B.: Postmortem for malicious packages published on july 12th, 2018 (2018). https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes. Accessed 07 Oct 2019
40. Zimmermann, M., Staicu, C.A., Tenny, C., Pradel, M.: Small world with high risks: a study of security threats in the npm ecosystem. arXiv preprint arXiv:1902.09217 (2019)