Can the OpenSSF Scorecard be used to measure the security posture of npm and PyPI?

Nusrat Zahan, Parth Kanakiya, Brian Hambleton, Shohanuzzaman Shohan, Laurie Williams
North Carolina State University

***Abstract*—The OpenSSF Scorecard project is an automated tool to monitor the security health of open source software. We used the tool to understand the security practices and gaps in npm and PyPI ecosystems and to confirm the applicability of the Scorecard tool.**

Introduction

Recent high-profile supply chain attacks, such as Solarwinds and Codecov, made headlines and directed attention towards the importance of software supply chain security. Therefore, practitioners are increasingly concerned with whether their projects' open-source software (OSS) components are secure.

Though standards, such as the NIST Secure Software Development Framework (SSDF) [1] and OWASP Software Component Verification Standard (SCVS) [2], provide exhaustive lists of security practices, a lack of consensus is observed regarding the implementation, validation, and verification of these practices towards a unified and consistent baseline measurement. Research is being conducted towards the development of different security metrics. However, establishing a baseline measurement is not straightforward since it involves exploring various sources of information including source code repositories, vulnerability tracking systems, continuous integration/ continuous deployment (CI/CD) pipeline, license(s) validity, package release history, and other metrics to develop standards for adoption. Additional challenges arise during the assessment of the security of packages in a software supply chain, particularly when the packages come from different sources and then deciding whether or not packages meet the baseline standards based on evidence. Also, practitioners must continue to monitor the OSS package's "health" to identify and manage any future risks of the software supply chain breaking down. Therefore, practitioners are now more interested than ever in identifying healthy open-source components and determining the security practices compared to other components within the ecosystem. Towards this end, ***the goal of this study is to aid practitioners in producing more secure software products and make informed decisions on the security practices of candidate dependencies by depicting the current security practices and gaps across ecosystems via an empirical study of the OpenSSF Scorecard project.***

The OpenSSF Scorecard project [3] is an automated tool to monitor the security health of the OSS supply chain. The primary goal of this project is to auto-generate a "security score" for OSS projects, using a list of security metrics

Published by the IEEE Computer Society

to verify security standards and acquire valuable information on the trust, risk, and security posture of candidate dependencies. While projects like Scorecard exist to perform heuristic-based checks of a package's security practices to aid dependency selection, little research has been done to understand the viability of using Scorecard security metrics to identify existing security gaps and practices in an entire ecosystem rather than just a single package. Observing the pattern of these security measures across one or more ecosystems can assist practitioners in determining how they are performing within the ecosystem, what they can do to improve security standards, or can assist in the evaluation of whether a specific security check is functional with that ecosystem. In this work, we studied the Scorecard tool to analyze what security practice patterns are observed in both ecosystems.

OpenSSF Scorecard

The Open Source Security Foundation (OpenSSF) is a cross-industry collaboration aiming to improve OSS's security. Since the inception on August 3, 2020, OpenSSF has grown to include members from over 75 organizations, including GitHub, Google, IBM, Intel, Microsoft, Red Hat, the OWASP Foundation, and others.

OpenSSF launched the Scorecard project [3] in November 2020 to provide an automated security tool that gives a "security score" for OSS and reduces the manual effort required to analyze a package's security. These results are made available via the Scorecard API, a BigQuery public dataset, and via the Open Source Insights (OSI) site.

At the time of the study, Scorecard evaluates 18 heuristics ("checks") and assigns a score between 0 to 10 to each. Scorecard assigns one of four risk levels to each check: **"Critical"** risk-weight 10; **"High"** risk-weight 7.5; **"Medium"** risk-weight 5; and **"Low"** risk-weight 2.5. For each package, an aggregate confidence score is also returned, which is a weighted average of the individual check score weighted by risk. The 18 Scorecard checks ranked by associated risk are as follows:

- **Dangerous-Workflow** (Critical) determines if there are dangerous code patterns in the package's `GitHub workflows` due to misconfigured GitHub Actions. Different events with GitHub contexts can trigger workflows. A long list of event context data, such as GitHub issues or pull requests (PR) creation, can be controlled by users and, if exploited, may lead to malicious injection.
- **Vulnerabilities** (High) tracks the presence of unfixed vulnerabilities of a package in the Open Source Vulnerabilities (OSV) [4] database.
- **Binary-Artifacts** (High) verifies the presence of executable (binary) artifacts in the repository. Since binary artifacts cannot be reviewed, it is possible to maliciously subvert the executable.
- **Token-Permissions** (High) determines whether the package's automated workflow tokens are set to read-only. This is important because attackers might inject malicious code into the project using a compromised token with write access. If the permission's definitions in each workflow's `yaml` file are set as read-only at the top level, and the required write permissions are declared at the run-level, the project gets the highest score. One point is deducted if the top-level permissions are not defined.
- **Code-Review** (High) evaluates if the package conducts code reviews prior to merging a PR. Code-Reviews may be able to detect malicious code insertion attempts by an attacker. The first step of the check is to see if Branch-Protection is activated with at least one required reviewer. If this fails, the check looks to see if the last 30 commits are Prow, Gerrit, Github-approved reviews or if the merger differs from the committer.
- **Maintained** (High) evaluates if the package is actively maintained and obtains the score based on activities on commits and issues from collaborators, members, or project owners. For example, if a project has at least one commit per week for the preceding 90 days for the latest 30 commits and issues, it will receive the highest score. Inactive projects run the risk of having unpatched code and insecure dependencies.
- **Branch-Protection** (High) monitors whether

GitHub's branch protection settings have been applied to a package's branches. This check enables maintainers to set guidelines to enforce specific workflows, such as requiring reviews or passing particular status checks before acceptance into the main branch. The check is scored on a five-tiered scale. Each tier has multiple check and must be fully satisfied to gain points at the next tier. If one of these checks fails, the package's score will be decreased based on the tier scaling, and if all of them fail, the package's score will be zero.

- **Dependency-Update-Tool** (High) determines whether the repository has enabled dependabot or renovatebot dependency update tool to automate the process of updating outdated or insecure dependencies by opening a pull request. Out-of-date dependencies make a project vulnerable to known flaws and prone to attacks.
- **Signed-Releases** (High) determines whether the project signed the release artifacts in GitHub by looking for the following filenames in the project's last five releases: `*.minisig`, `*.asc (pgp)`, `*.sig`, `*.sign`. Signed-Releases attest to the provenance of the artifact.
- **Pinned-Dependencies** (Medium) looks for unpinned dependencies in `Dockerfiles`, `shell scripts`, and `GitHub workflows` to see if the project has locked its dependencies. Unpinned-Dependency practices allow auto-updating a dependency to a new version without reviewing the differences between the two versions, which may include an insecure component.
- **Security-Policy** (Medium) looks for a file entitled `SECURITY.md`(case-insensitive) in directories like the top-level or the `.github` of a repository to see if the package has published a security policy. Users can learn what constitutes a vulnerability and how to report it securely via a security policy.
- **Packaging** (Medium) detects language-specific GitHub Actions that upload the package to a related hub and determines if the package is published by GitHub packaging workflows. Packaging makes it easy for users to receive security patches as updates.
- **Fuzzing** (Medium) determines if the project uses fuzzing by checking the repository name in the OSS-Fuzz project list. Fuzzing is important to detect exploitable vulnerabilities.
- **Static Application Security Testing (SAST)** (Medium) determines if the project uses SAST. These tools can prevent bugs from being inadvertently introduced in the codebase. The checks look for known Github apps such as CodeQL, LGTM, and SonarCloud in the recent merged PRs, or the use of "`GitHub/codeql-action`" in a GitHub workflow.
- **License** (Low) verifies if the project has published a license by looking for any combination of the following names and extensions in the top-level directory: `LICENSE`, `LICENCE`, `COPYING`, `COPYRIGHT` and `.html`,`.txt`,`.md`. Scorecard can also detect these files in the `LICENSES` directory. The lack of a license will hinder any security review and creates legal risk for potential users.
- **CII-Best-Practices** (Low) verifies whether the package has a CII Best Practices Badge, which certifies that it follows a set of security-oriented best practices- vulnerability reporting policy, automatic process to rebuild the software, SAST and so on.
- **CI-Tests** (Low) determines if the project runs tests before PRs are merged by looking for a set of CI-system names in GitHub `CheckRuns` and `Statuses` in recent 30 commits. CI-Tests enable developers to identify problems early in the pipeline.
- **Contributors** (Low) determines if the project has contributors from multiple organizations by looking at the company field on the GitHub user profile to identify trusted code reviewers. The project must have had contributors from at least three organizations in the last 30 commits to receive the highest score.

Other Community Efforts

Guidelines and Standards: In response to Section 4 of the President's Executive Order (EO) on "Improving the Nation's Cybersecurity (14028)" [5], the U.S. National Institute of Standards and Technology (NIST) updated the Secure Software Development Framework (SSDF) [1] to integrate the framework into each SDLC imple-

mentation. The framework focuses on how an organization can transition from its current open source software development processes to the SSDF practices, describing high-level practices and tasks based on established secure software development practices. There are four groups of practices, containing 41 tasks to fulfill the goal: i) Prepare the Organization, ii) Protect the Software, iii) Produce Well-Secured Software, iv) Respond to Vulnerabilities. The framework mentioned that automatability of these practices are an important factor to consider, especially for implementing practices at scale. The OpenSSF Scorecard tool [3] aims to complement the Executive Order (EO) and this framework by mass-scale automating these practices and tasks. Out of the 18 Scorecard security metrics, 13 can be mapped to the SSDF framework's tasks as part of secure SDLC practicing for organizations.

The OWASP Software Component Verification Standard (SCVS) [2] is a framework to develop a common set of activities, controls, and best practices that can help in identifying and reducing risk in a software supply chain. There are 6 control families that contain 87 controls for different aspects of security verification or processes. The SCVS has three verification levels, where higher levels include additional controls.

Tools: Open Source Insights (OSI) [6] is a Google-developed and hosted tool that aims to help practitioners to grab information about the source code location, package metadata, licenses, releases, and vulnerabilities of open-source products. OSI scans millions of open-source packages from different ecosystems, constructs dependency graphs, and annotates the metadata in a dashboard. Apart from the package's dependencies, the OSI dashboard also shows a package's direct or transitive dependent stats. On the OSI website, users can view the vulnerability mapping of a package as well as the vulnerability mapping with associated dependencies. Apart from these package metadata, OSI has also integrated the Scorecard security metrics to assist in apprehending the package security practices.

The CHAOSS project [7] is used to develop a set of metrics and tools to support consistency in the individual assessments of a project's health and sustainability for the people who build, maintain, contribute to, and consume OSS [8]. However, they do not have any public data to analyze a project's health. The OSSMETER project [9] is a monitoring and analysis platform to assess a project's source code quality and unmaintained products. Their last release was in 2014, and the last source code was committed in 2017. The OSS Review Toolkit, Dependency-track, and Anvaka, are built to track dependencies in the software supply chain. On the contrary, scancode-toolkit: ScanCode detects licenses, copyrights, package manifests, and dependencies. These efforts are valuable as an ad-hoc basis solution. However, none of these enable the measuring of different security metrics in one place. Here, OpenSSF Scorecard [3] contributes by continuously identifying relevant security metrics and integrating them into a user-friendly command-line interface.

Methods

This section discusses the data sourcing and generating process of this study. We compiled a package list and relevant metadata from npm and PyPI ecosystems to collect the security score for those packages from Scorecard tool.

Ecosystem Package Metadata

First, we collected the package list and dependents information for npm and PyPI ecosystems.

Package Name: To begin, we collected a list of all package names available in both ecosystems. We sourced the list of npm packages name (1,494,105) from Zahan et al. [10] study and the list of PyPI package names (3,65,450) was collected using PyPI API [11] in April 2022.

Dependents data: The number of dependents reflects the importance of a project by quantifying how many other projects use it. We collected dependent information from the OSI API [6]. In this work, we collected dependent information to prioritize the packages list for manual review.

OpenSSF Scorecard Score

The Scorecard tool only runs on source code hosted by GitHub. Hence, to obtain the Scorecard scores for a given package, the first step was to map the package with its respective source code location. To retrieve the source code location for both ecosystems, we use the OSI API [6]. We collected 767,389 npm unique GitHub repositories

and 191,158 PyPI unique GitHub repositories. Note that the package-to-repository mapping is not always a 1:1 match. Multiple packages can be found in a single repository. In total, we collected 947,936 npm packages with 767,389 unique GitHub repositories and 211,088 PyPI packages with 191,158 unique GitHub repositories.

Then, Scorecard runs a weekly scan of open source packages to generate the security score of those packages. However, we could not directly utilize this data for both ecosystems because, by the time of this study, Scorecard scores were only generated on 760k of 947K npm and 10K of 211K PyPI packages. Therefore, we submitted a pull request to the Scorecard repository, adding the GitHub repositories of missing packages to collect the scores from both ecosystems. The weekly Scorecard scan was able to run on those GitHub repositories after the Scorecard team successfully merged the PR.

Out of the 947,936 npm packages and 211,088 PyPI packages, we collected the generated score of 832,422 npm packages and 191,483 PyPI packages. We analyzed 50 randomly-chosen packages where the Scorecard failed to generate scores to determine why the score was not generated. We found the following two reasons: a) the GitHub repositories were deleted, and b) the GitHub repositories' permissions were private. We collected the Scorecard score on May 09, 2022.

For each package, we could obtain 15 out of 18 Scorecard security metrics and their aggregate score, with the missing 3 metrics being the CI-Test, SAST, and CONTRIBUTOR checks. The Scorecard team took out these three checks to scale the weekly job since computing these metrics is API intensive, and GitHub rate limiting can be a bottleneck for the weekly run. As a result, we could not collect data for these three metrics.

Ecosystem Security Practices

We observe each ecosystem's security practices and patterns by analyzing the Scorecard security checks and their assigned values. To that, we measure the frequency of packages with scores for the 15 security practices, depicted in Figure 1 in npm and PyPI, respectively. We consider each of the practices as a security check metric.

The notation `-1` in figure 1 indicates that Scorecard could not get conclusive evidence of implementing practices, or perhaps an internal error occurred due to a runtime error in Scorecard. The inconclusive outcome is graded as -1 instead of 0, to not give a penalty of failing a check to the package, since a value of 0 will affect the package's aggregate score. Seven of the 15 security checks had packages with a score of -1. The notation "1-10" denotes the percentage of packages achieving scores ranging from 1 to 10. A higher % (green cell in Figure 1) shows that the majority of the packages implement the practice. A lower % (red cell) indicates a higher % of packages failed the practice and received a score of 0. For example, in PyPI, the Fuzzing check had 0.1% packages scored between `1-10` and 18% package scored `-1`, hence, 81.9% packages scored `0`. The mean and standard deviation are measured to understand the central tendency and spread of score distribution in an ecosystem.

To learn about why a check passed or failed, we manually reviewed 25 sample GitHub repositories from each ecosystem for each practice. We ranked each metric by the highest number of dependents and selected 25 packages based on their score. One author reviewed 50 GitHub repositories (25 from each ecosystem) totaling 750 repositories for the 15 checks. A second reviewer then verified the findings by selecting 100 repositories at random. We used the Cohen Kappa statistic to test the inter-rater reliability and achieved a score of 0.961. We resolved our disagreement after discussing our findings, and the first reviewer cross-reviewed other repositories to make changes if required. Then, if we needed to examine more packages to understand a given score, we again chose further packages by highest dependents order.

We discuss each security check for both ecosystems and the frequency statistics in the following subsection.

Dangerous-Workflow: This check detects the following two patterns in workflows: untrusted code checkout; and script injection with untrusted context variables. More than 99% packages had passed the check. However, we found 1,938 npm packages and 508 PyPI packages where Scorecard found vulnerable code patterns.

Security Check	Package Score Frequency				Score Stat			
	npm	PyPI	npm	PyPI	npm	PyPI	npm	PyPI
	-1	-1	1--10	1--10	Mean	Mean	STD	STD
✓ Dangerous-Workflow	0.0%	0.0%	99.8%	99.7%	10.0	10.0	0.5	0.5
Pinned-Dependencies	0.1%	0.3%	99.8%	99.2%	9.5	8.7	1.3	2.1
✓ Vulnerabilities	0.3%	0.2%	99.7%	99.8%	10.0	10.0	0.0	0.0
✓ Binary-Artifacts	0.0%	0.0%	99.5%	98.6%	9.9	9.7	0.8	1.4
✓ Token-Permissions	0.0%	0.0%	**84.4%**	**71.7%**	8.4	7.2	3.6	4.5
✓ License	0.0%	0.0%	**68.6%**	**88.0%**	6.9	8.8	4.6	3.3
✓ Code-Review	0.3%	0.2%	30.8%	34.8%	1.4	1.5	2.7	2.7
✓ Maintained	0.0%	0.0%	**13.9%**	**24.1%**	1.1	1.8	2.9	3.5
✓ Branch-Protection	1.7%	1.5%	10.2%	10.3%	0.5	0.5	1.7	1.6
✗ Dependency-Update-Tool	0.0%	0.0%	5.5%	2.9%	0.6	0.3	2.3	1.7
✓ Security-Policy	0.0%	0.0%	3.2%	2.6%	0.3	0.3	1.8	1.6
✗ Packaging	99.0%	94.1%	**1.0%**	**5.9%**	10.0	10.0	0.0	0.0
✗ CII-Best-Practices	0.0%	0.0%	0.2%	0.2%	0.0	0.0	0.2	0.1
✗ Signed-Releases	97.5%	93.0%	0.1%	0.5%	0.3	0.7	1.6	2.5
✗ Fuzzing	17.9%	18.0%	0.0%	0.1%	0.0	0.0	0.1	0.3

% of packages passing a security check with a score between 1-10. Towards red cell indicate that a higher % of packages failed the check with a score of 0.

% of packages had inconclusive results, hence, achieved a score of -1, instead of 0-10

✓ Packages can use Scorecard to verify these practices immediately

✗ Packages may not use Scorecard to verify these practices immediately due to check's inherited reliance on other systems

To display accurate stats of ecosystem practices, the Mean and STD(standard deviation) columns did not consider the [-1] value

Figure 1: npm and PyPI Ecosystems Security Practices measured by Scorecard Tool

Out of 50 repositories used for manual analysis, we had 8 packages with `-1`, all of which were the outcome of internal errors, and 11 packages with vulnerable code patterns in workflows, hence, scored `0`. Among them, 3 npm packages had untrusted code checkout patterns, and 5 PyPI and 2 npm packages had warnings about script injection with untrusted input. At the end of this section, we have demonstrated a case study explaining how an attacker can exploit such patterns in workflows.

Pinned-Dependencies: In both ecosystems, more than 99% of packages had a practice of using at least one pinned dependency in the repository. Among these, 81% npm packages and 66% PyPI packages got a score of 10, indicating that they do not have any unpinned dependencies in listed directories. The score, however, may not reflect an accurate scenario. We observed that the tool does not check `package.json` and `package-lock.json` files in a npm package repository. A `package.json` is a JSON file that exists at the root of npm packages, containing the metadata relevant to the project to manage the project's dependencies, scripts, versions, and a whole lot more. We found packages could achieve a score of 10 even if the package had multiple unpinned dependencies in JavaScript package's `package.json` file, indicating Scorecard findings do not indicate the accurate status of pinned dependencies in an ecosystem. We also observed that Scorecard does not verify the presence of `Dockerfiles`, `shell scripts`, and `GitHub workflows` files in a repository. If a repository did not have any of

those files, a package would receive a score of 10 for not having an unpinned dependency on those missing files.

Vulnerabilities: More than 99% packages did not have any open vulnerabilities in the OSV database. Hence, they scored 10. Scorecard found 7 npm packages and 5 PyPI packages with unfixed vulnerabilities. In addition, 2,703 npm packages and 322 PyPI packages got a score of `-1` for inconclusive results. Our manual repository review considered repositories where the package had inconclusive scores or open vulnerabilities, ranked by number of dependents. Note that, we did not review packages with score 10 since, these packages did not have any open vulnerabilities reported in OSV database. The reason behind the negative, inconclusive score was that those repositories were empty. In total, we found 39/50 empty repositories. One package had ten open vulnerabilities with a score of 0, and nine packages had one vulnerability open with a score of 9.

Binary-Artifacts: More than 99% packages had a score greater than 0. The manual review of 50 repositories found eight packages with a score of `0` and noticed that these packages had more than nine binary artifacts with a mean and standard deviation of 78.25 and 87.17, respectively. These packages were umbrella projects encompassing a variety of tools and libraries. Clients are forced to use them directly with the inability to inspect those files for dangerous behaviors. Another 32 packages in manual review were given a score from `1` to `10` based on the number of executable files ranging from 0 to 9. A score of 10 means no binaries, a score of 9 means the presence of one binary, and the scores continue to decrease toward 1 as the executable files increased towards 9. We also found a false positive in one npm package repository, where Scorecard identified 108 binaries, two of which were `.txt` files.

Token-Permissions: In this check, npm yielded a more promising result: nearly 84% of packages have read and write permissions declared, compared to 71% of PyPI packages. Our manual review found similar patterns as we observed in Pinned-Dependencies. Fourteen (14) packages did not have any GitHub Actions specified in the repository but Scorecard assigned `10` to those packages since the tool does not check the presence of workflows in the repository.

License: We observed that 68% npm packages and 88% PyPI packages had published licenses in GitHub repository, indicating, npm has a higher tendency to avoid licensing in the repository. Our manual review revealed that 4 npm packages and 8 PyPI packages had a license mentioned in the repository, specifically in `Readme.md` and `setup.py` files. However, Scorecard could not identify them. We found the following unique file name and extension `GPL-2.0`, `LICENSE`, `LICENCE`, `LICENSE.txt`, `LICENSE.rst`, `LICENSE.PSF`, `LICENSE.APACHE`, `LICENSE.BSD`, `LICENSE.md`, `LICENSE-MIT` from our review.

Code-Review: 30% npm packages and 34% PyPI packages had code review practices in their repository. One reason behind failing this check would be that the check is not applicable if the package has one maintainer. The trait is expected in the case of npm because most packages in npm are small and maintained by one maintainer. Study [10] reveals that, in 2021, 1.5 million npm packages had an average of 1.7 maintainers. However, our manual review found otherwise 9 packages scored 0 and had no code review practices even though they had more than one contributor in GitHub repositories. We also had `-1` in 5 sample repositories where the repos were empty. To verify this pattern we reviewed additional 10 repositories with `-1`. These repositories were empty on GitHub. Hence, indicating why Scorecard assigned `-1` as an inconclusive result. In total, we found 2,695 npm and 321 PyPI empty repositories with `-1` where the other four checks Vulnerabilities, Branch-Protection, Packaging, and Signed-Releases including Code-Review had `-1`.

Maintained: Our findings show that more than 85% packages in npm and 75% PyPI packages were unmaintained in GitHub. What is more crucial is that for npm, unmaintained packages

may have a more extended period than 90 days, as study [10] revealed that in 2021, more than 58% of packages in the npm registry were unmaintained for more than two years. Our manual inspections were consistent with Scorecard data where 9/50 packages were inactive in a range of 1 year to 7 years.

Branch-Protection: Only 10% packages passed this check in each ecosystem, indicating these repos had at least one tier of branch protection applied. Hence, 90% npm and PyPI packages had branch protection disabled in the repository. The numbers are considerably high, indicating that a large number of packages in both ecosystems did not create a branch protection rule in repositories. Out of five tiered scoring- enabling branch protection, by default inhibits force to push and branch deletion, a tier 1 check, and assigns a score of 3. The score would be raised to 6 as a tier 2 check if the package has at least one reviewer. Enabling status checks will result in a score of 8 as a tier 3 check, which will be increased to 9 if a second reviewer exists as a tier 4 check. Finally, a package will get a score of 10 if the admin dismisses the stale review.

From the manual review, we found that Scorecard check default branch and any branch that was used for creating a release and use GraphQL API to verify the protection. However, we verified the branch-protection by looking into GitHub branches api [12]. Then we also found 13 out of 50 packages had a score of `-1` due to internal error because Scorecard: a) looked for the incorrect branch name that did not exist in the repository, b) could not locate the branch even though it existed, c) the main branch had a different name than the "main" or "master", and d) branch protections were disabled in main and release branch. In our study, we have 13,863 npm packages and 2,894 PyPI packages with a score of `-1`.

Dependency-Update-Tool: 94% npm packages and 97% PyPI packages failed this check since dependabot and renovatebot were not used as dependency update tools. A project that uses other tools or manually updates dependencies, will obtain a score of 0 on this check, just like other packages with outdated dependencies. Note that, this check can only confirm if the dependency update tool is enabled; it can not confirm if the dependency-update-tool is running or if the tool's pull requests are merged.

Security-Policy: Only 3.2% npm and 2.5% PyPI packages have a `security.md` file. After looking into 50 sample packages, we observed: a) 25 packages do not adhere to standard security policies; b) 11 packages have a different reporting procedure for vulnerabilities. Users can, for example, submit bugs in other places such as GitHub issues, specific email addresses, and different bug databases outside of GitHub, or use a different security policy reporting file `security.rst`.

Packaging: Only 1% of npm packages and 5.8% of PyPI items passed the packaging check. Since the software can be packaged in multiple ways, the challenges of coordinating several package release protocols may prohibit developers from releasing packages on GitHub Actions, which can be one reason for the limited number of packaging in the GitHub packaging workflows. To the date of this study, Scorecard can not query the package registries directly, hence, packages that do not use GitHub actions get `-1` instead of 0. Note that a package's aggregate score will be penalized if it has a 0 score and inconclusive checks, or `-1`, has no effect on the aggregate score. Our manual inspection identified only 2 npm packages and 6 PyPI packages used GitHub packaging workflow, while 47/50 packages had releases on GitHub. Additionally, the Scorecard failed to detect four packages (two from each ecosystem) that had a publishing GitHub workflow. The name of these files are [publish, ci, release].yml.

CII-Best-Practices: Scorecard found the CII Best Practices Badge in just 1,665 (0.2%) npm and 341(0.1%) PyPI packages. The CII Best Practices program is a way for Free/Libre and Open Source Software (FLOSS) projects to demonstrate that they follow best practices. Projects can voluntarily self-certify to report how they follow each best practice. According to the CII Best Practice Program website only 4,766 FLOSS projects have reported their security policies and

received different degrees of badges, indicating why both ecosystems fell short of this check.

Signed-Releases: Only 578(0.1%) npm and 936(0.5%) PyPI packages had signed releases. Moreover, almost 100% packages failed this check. Even though the number is low, this is expected behavior for both ecosystems, as package developers release versions to the package registry (`npmjs.org or pypi.org`) rather than code hosting platforms like (`github.com`). Additionally, we observed that Github, PyPI, and npm all have different regulations to control package release to a registry. To publish in both registries, one must take additional steps to confirm the release, which can be incompatible with their workflow [13]. For instance, the GitHub registry accepts only scoped packages. Therefore, if a JavaScript package is currently named *X*, it must be renamed *@username/X* to publish in GitHub.

Scorecard assigns `-1` instead of 0 if the tool can not detect the signed release. In addition, our manual review revealed that Scorecard often verifies older signed versions rather than checking for signatures on the new five releases. For example, one package received an 8/10 score, meaning 4/5 of recent releases of that package had signed artifacts. However, We found the signed artifacts that Scorecard awarded points were from older versions which contradict the defined rules of Scorecard. Then we also observed repositories tagged commits as a release rather than creating a release on GitHub. However, none of the commits were GitHub verified, and Scorecard does not identify tagged releases. Scorecard can extend this check by looking for tagged commits with releases in the API tags endpoint.

Fuzzing: Both ecosystems fell short on this check. Scorecard validates fuzzing exclusively through the tracking of packages in OSS-Fuzz. OSS-Fuzz has been tested only in 550 open-source packages as of January 2022 and a package that uses fuzzing with other tools would fail the check similar to Dependency-Update-Tool check, hinting why the npm and PyPI ecosystems failed this check. Out of 550 open-source packages that use OSS-Fuzz, we found 50 npm packages and 104 PyPI packages. Despite the fact that this check was used by only a few packages, PyPI has more fuzzing practice (50 percent more than npm) than npm. One reason why npm packages do not use fuzzing could be that fuzzing JavaScript(JS) engines is tricky and requires expertise. This is because, rather than processing user-supplied seed, JS engines scan and interpret user seed into an abstract syntax tree (AST) [14] which impacts the performance of fuzzers. Our manual analysis yielded no different results from what we expected. Only two PyPI packages used OSS-Fuzz, and 48 other packages had no fuzzer. Then, Fuzzing had most of the `-1` after Signed-Releases and Packaging, but in our manual analysis re-running 12 packages with `-1`, scored `0` in new run, indicating run time error occurred during the first run.

Deep Dive on Dangerous Workflow Check:

Dangerous-Workflow checks dangerous coding patterns in the code files of GitHub Actions. Potentially dangerous misuse of the GitHub workflows trigger may lead to malicious authors (i.e., attackers) being able to perform data theft and data integrity breaches. The usage of an injection attack in the `issue title` and `issue comment` is a simple demonstration of such an attack vector, allowing the attacker to break the action environment and launch a process on the runner environment.

We explored if Scorecard can accurately detect these patterns in a GitHub repository. To that, instead of attacking existing repositories that failed this check, we execute Scorecard tool on a dummy GitHub repository where we build workflow with an intentionally-vulnerable issue action, inspired by [15]. Our vulnerable workflow (Figure 2) is executed on a GitHub runner whenever a new issue is created by anyone. For our attack, we crafted a malicious issue title in the dummy repository as malicious author to open a reverse shell connection in our local machine.

For reverse shell, we created the following `issue` in the dummy repository from a different GitHub account user- `New malicious issue title" && bash -i >& /dev/tcp/4.tcp.ngrok.io/{ngrok endpoint} 0>&1 && echo"`. Here, we used ngrok service on local machine to expose local server ports to the Internet. Whenever the `issue` is created on the dummy repository,

17 lines (17 sloc) | 551 Bytes **Step 1: Vulnerable workflow in the repo** Raw Blame

```yaml
name: Demo vulnerable workflow
on:
  issues:
    types: [opened]
env:
  GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
jobs:
  vuln_job:
    runs-on: ubuntu-latest
    steps:
     # Checkout used for demonstration purposes
      - uses: actions/checkout@v2
      - run: |
          echo "ISSUE TITLE: ${{github.event.issue.title}}"
          echo "ISSUE DESCRIPTION: ${{github.event.issue.body}}"
      - run: |
          curl -X POST -H "Authorization: Token ${{ secrets.GIT_AUTH_TOKEN }}" -d '{"labels": ["New Issue"]}' ${{ github.event.issue.url }}/labels
```

Step 2: Scorecard to detect the vulnerable pattern

```
RESULTS
-------
Aggregate score: 0.0 / 10

Check scores:
|--------|--------------------|---------------------------------|-----------------------------------------|
| SCORE  |        NAME        |              REASON             |                 DETAILS                 |
|--------|--------------------|---------------------------------|-----------------------------------------|
| 0 / 10 | Dangerous-Workflow | dangerous workflow patterns     | Warn: script injection with untrusted   |
|        |                    | detected                        | input 'github.event.issue.title':       |
|        |                    |                                 | .github/workflows/scriptInjection.yml:13 |
|        |                    |                                 | Warn: script injection with untrusted   |
|        |                    |                                 | input 'github.event.issue.body':        |
|        |                    |                                 | .github/workflows/scriptInjection.yml:13 |
|--------|--------------------|---------------------------------|-----------------------------------------|
```

Figure 2: Case study on Dangerous Workflow to detect vulnerable code pattern

GitHub workflows will print the issue details to the log, and label it as "New Issue" using a PAT (Personal Access Token). Here, the line 14 in step 1 of Figure 2 `"ISSUE TITLE: {{github.event.issue.title}}"` is vulnerable to command injection because the hosted runners replaces the macros `{{ ... }}` blindly and echo `"{{github.event.issue.title}}"` becomes echo `"{{New malicious issue title}}"`, thus, giving an attacker to run a reverse shell inside the hosted runner as part of the arbitrary code execution capabilities. An attackers can read sensitive files like `.credential` from the runner folder.

We tested our dummy repository with Scorecard tool to verify whether the Scorecard identify the vulnerable pattern in our workflow and step 2 in Figure 2 shows that the Scorecard was able to identify the vulnerable pattern referring to the exact line. This case study is an example of how the dangerous workflow check can help to identify vulnerable code patterns. We found 2,446 npm (1,938) and PyPI (508) packages that might be vulnerable to similar malicious attacks.

Discussion:

Our work is an effort to utilize OpenSSF Scorecard framework and tool to understand and measure cross-ecosystem package security practices and gaps. The Scorecard project seeks to provide a head start for everyone, including expert and non-expert decision-makers, to understand how a package performs against other OSS packages in an ecosystem and how one can identify the gap in security practices and improve the security posture. For example, if a package has a set of dependencies, then the package owner can test all these candidate dependencies by using Scorecard tool to understand the standard security practices. Suppose one of the dependencies has open vulnerabilities with no maintained or code review status. In that case, the project owner can decide whether to accept these risks or work with the relevant stakeholders to improve the package security.

We have also observed and been told that the Scorecard team welcomes new security metrics and discussions which indicate the Scorecard is

evolving with time. Scorecard adoption at the community level will help with increasing package security, identifying Scorecard's existing issues, and determining how Scorecard may evolve and improve in response to diverse ecosystem particularities.

Security Check Metrics Evaluation

The metrics in the Scorecard are of different levels of usefulness: some are abstract, whereas others are concrete. A deliberate degree of freedom is necessary to calculate the aggregate score. For example, CII-Best-Practices open up debate about whether they are beneficial in some cases since it is a voluntary effort of maintainers to self-certify a report on how they follow different best practices and having failed this check may not necessarily always advocate package do not implement these security practices. On the other hand, practitioners can use the Scorecard tool to measure the security practices of Dangerous-Workflow, Vulnerabilities, Binary-Artifacts, Token-Permissions, License, Code-Review, Maintained, Branch-Protection and Security-Policy metrics.

Then, the standards specified by the Scorecard tool for Dependency-Update-Tool, Fuzzing, Signed-Releases and Packaging security checks, were weakly adopted in npm and PyPI. At the time of the study, Scorecard assigned -1 if they can not verify the evidence of Signed-Releases and Packaging practices, hence it does not influence the aggregate score of a package. However, Scorecard intend to integrate the package registry to improve these metrics measurement. Dependency-Update-Tool, Fuzzing metrics can be measured differently due to the possibility of using different tools. In OSS communities, there is no agreement on the list of tools or how to confirmed whether a package makes use of any of them. For better OSS security, the community needs to agree or standardize these metrics so that Scorecard can integrate the standard practices. We do acknowledge, however, that this could be challenging. Hence, Scorecard may separate these metrics from the aggregate score calculation; if a package implements these practices, it may get a bonus point instead of directly impacting the aggregate score. Steps like these will aid in depicting a more accurate picture of ecosystem security best practices.

The Pinned-Dependencies check requires revision based on different ecosystem. For example Pinned-Dependencies do not check the `package.json` and `package-lock.json` file for the dependency version. Then Signed-Releases, and License matrices can be improved. Signed releases, for example, ignore the tagged release commit and do not search the package registry directly for the releases. Then, enhancing the list of keywords can help boost the License stats. For better ecosystem evaluation, we should filter out packages with empty repositories, but Scorecard generated aggregated scores for those repos because metrics like Dangerous-Workflow, Binary-Artifacts, Pinned-Dependencies, and Token-Permissions looked for the existence of specific properties in GitHub workflows, which empty repositories did not contain. Hence, Scorecard assigned a score of 10 instead of `0, -1`, whereas other 11 metrics values were between `0,-1`. These metrics open up debate about whether Scorecard should check for existence of GitHub workflows first, before verifying the good or bad practices. For example, if a package had no GitHub workflows, tool will scored 10 in Dangerous-Workflow and Token-Permissions metrics, however, it does not actually verify that package follows good workflows patterns.

Apart from that, both ecosystems have a gap in practicing Code-Review, Maintained, License Branch-Protection and Security-Policy practices in the GitHub repository. In terms of License and Maintained, PyPI outperformed the npm ecosystem. Only 68% npm packages had a published license in the repository, compared to over 88% of PyPI packages. Only 13% npm and 24% PyPI packages were actively maintained. Additionally, 90% packages in both ecosystems did not even have a default branch protection enabled in their repositories. Practitioners should improve the security practices of these metrics.

Conclusion:

This research attempts to abstract from the GitHub repository level and bring security check metrics to the ecosystem level for baseline measurement. As projects within one ecosystem use different tools and regulations and have wildly

varying levels of activities, it is hard to find complete baseline measurements. The Scorecard attempts to establish a baseline for OSS security measurement. Even if some metrics are not functional for all packages, knowing about them will inspire and direct practitioners on what to do if they want to adopt these practices or identify the gap that is preventing them from doing so.

ACKNOWLEDGMENTS

This work was supported and funded by Cisco. We thank the OpenSSF Scorecard Team for their valuable feedback and assistance in generating Scorecard data for such a vast number of repositories.

REFERENCES

1. M. Souppaya, K. Scarfone, and D. Dodson, "Secure software development framework (ssdf) version 1.1," *NIST Special Publication*, vol. 800, p. 218, 2022.
2. OWASP, "Software component verification standard (scvs)," *https://owasp-scvs.gitbook.io/scvs/*, 2020.
3. Scorecard, "Security scorecards for open source projects," *https://github.com/ossf/scorecard*, 2021.
4. Google, "Open source vulnerability database," *https://osv.dev/*, 2021.
5. D. The White House, Washington, "Executive order on improving the nation's cybersecurity," *https://www.federalregister.gov/documents/2021/05/17/2021-10460/improving-the-nations-cybersecurity*, 2021.
6. OSI, "Open source insight(osi)," *https://deps.dev/*, 2022.
7. CHAOSS, "Community health analytics open source software," *http://chaoss.community*, 2022.
8. G. J. Link, "Open source project health," *USENIX PATRONS*, p. 31, 2020.
9. OSSMETER, "Ossmeter – automated measurement and analysis of open source software," *http://www.ossmeter.org/*, 2017.
10. N. Zahan, T. Zimmermann, P. Godefroid, B. Murphy, C. Maddila, and L. Williams, "What are weak links in the npm supply chain?" in *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2022, pp. 331–340.
11. PyPI, "Pypi api for package name," *https://pypi.org/simple/*, 2022.
12. GitHub, "The github branches api," *api.github.com/repos/OWNER/REPO/branches/branch-name*, 2022.
13. D. Wermke, N. Wöhler, J. H. Klemmer, M. Fourné, Y. Acar, and S. Fahl, "Committed to trust: A qualitative study on security & trust in open source software projects," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2022, pp. 1572–1572.
14. S. T. Dinh, H. Cho, K. Martin, A. Oest, K. Zeng, A. Kapravelos, G.-J. Ahn, T. Bao, R. Wang, A. Doupé *et al.*, "Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases," in *Proceedings 2021 Network and Distributed System Security Symposium, Virtual*, 2021.
15. N. Dotam, "Vulnerable github actions workflows part 1: Privilege escalation inside your ci/cd pipeline," *https://www.legitsecurity.com/blog/github-privilege-escalation-vulnerability*, 2022.