CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Ensuring the Security of PyPI Packages

Master's thesis in Computer science and engineering

David Shakoori Gustafsson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Master's thesis 2023

Ensuring the Security of PyPI Packages

David Shakoori Gustafsson

UNIVERSITY OF GOTHENBURG

CHALMERS UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Ensuring the Security of PyPI Packages LaTeX

David Shakoori Gustafsson

Supervisor: Morten Fjeld, Computer Science and Engineering
Examiner: Alejandro Russo, Computer Science and Engineering

Master's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2023

Ensuring the Security of PyPI Packages LaTeX

David Shakoori Gustafsson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Developers often use open-source code libraries in order to achieve desired functionalities without needing to re-implement existing code. Python developers are no exceptions here, and frequently use the Python Package Index, PyPI, to download the specific code packages they want to use. However, PyPI has few restrictions on what can be uploaded, making certain attacks on its ecosystem relatively simple. This thesis seeks to analyse potential vulnerabilities of PyPI, discuss threats posed to the system and its users, and propose potential countermeasures. Countermeasures can be looked at from two sides; the side of PyPI service providers, and the side of PyPI users. In this thesis work, a threat model of PyPI is created, in which different entry points, assets, and potential threats are identified, ranked and categorised. Also, a number of different user-side tools for discovering malicious packages are discussed, and a small proof-of-concept program utilising those tools is created, after which the tools are evaluated. These are tools related to, for example, information gathering (e.g. the GitHub API or Safety DB), pattern-matching (regular expressions), and containerisation (Docker).

While the threat model is limited, several potential threats, as well as a number of respective countermeasures, are found and discussed. One example is the easy-to-perform typosquatting attack; there currently is no protection against such attacks in the PyPI system. Implementing some sort of community reporting feature could make the discovery of such packages easier. As for the user-side tools that we evaluate, they probably cannot detect malicious packages on their own. However, using a combination of multiple tools would likely decrease the chances of installing malicious code packages.

Keywords: Computer science, Security, Python, PyPI, Typosquatting.

Acknowledgements

I would like to thank my supervisor Morten Fjeld, as well as my examiner Alejandro Russo, for help and valuable feedback on my thesis. Furthermore, I would like to thank my friends Rasmus and Filip for their unwavering support during the thesis writing period and beyond. Finally, I would like to thank my sister Freia for being the best.

David Shakoori Gustafsson, Gothenburg, 2023-06-18

Contents

1 Introduction

The use of packages in the world of Python is commonplace, mostly due to the overhead of implementing the functionality that such packages can offer. Why would you create your own functionality for scientific computing or graph visualisation, when packages such as NumPy and Matplotlib already exist in the Python Package Index (PyPI)? However, PyPI does little explicit administration of its uploaded packages, relegating responsibility for potential security concerns to the user installing the packages. Given the open-source nature of Python and other similar languages, anyone can upload their packages to the respective repository, giving malicious actors an easy way of spreading potentially compromising code.

Ruohonen et al. [1] ran a static analysis on a majority of PyPI's available packages, and found that almost half of them had at least one security issue. The issues mostly consisted of generic vulnerabilities, such as hard-coded passwords in the source files, and injection vulnerabilities, which may allow for SQL injection attacks. Furthermore, the lack of administration in PyPI allows for so-called typosquatting attacks, where malicious packages with names similar to commonly used packages are uploaded to the index.

Given the widespread usage of code repositories by developers coding in programming languages such as Python, Ruby, JavaScript, etc., there are many reasons for keeping these packages free of malicious code. Reaching an adequate level of security is difficult, but with cooperation between the owner of the repository, and the user downloading the package, the risks can be reduced.

This thesis will focus on establishing a few ways of ensuring the safety of PyPI's Python packages. As such, the purpose of the thesis is twofold:

1. Analyse the current operation of PyPI administration and discuss potential solutions to detected security concerns.
2. Study how secure transactions with PyPI can be achieved from the user's side, e.g. through some proof-of-concept software implementation.

The first part entails a thorough analysis of PyPI, followed by presenting some alternative administrative ways of increasing its security. PyPI is hosted by the Python Software Foundation, a non-profit organisation. As such, the organisation does not have the resources of, for example, the Microsoft-owned GitHub when it comes to administration. With that in mind, it is quite easy to see how typosquatting attacks, among other security threats, have become large issues in the PyPI ecosystem [2].

For the second part, apart from theoretical reasoning, a program will be designed to detect possibly malicious packages, or at least help users make informed decisions about installing a package, or not. The program should be able to, based on information from PyPI and other resources, decide if a package's name is legit or a potential case of typosquatting. Furthermore, the program should check if a package is safe to install and run on a user's device. Some alternative tools for this purpose will be discussed and tested.

The listed items above can be condensed into three research questions, with the first two (RQ1 and RQ2) being related to the administrative side of PyPI, and the last one (RQ3) to the user side. They are as follows:

- **RQ1** – From a threat modelling perspective, what is the current state of PyPI's security?
- **RQ2** – What safety measures can the service providers at PyPI take to ensure the security of its users?
- **RQ3** – For Python developers to detect questionable packages, which tools are effective and why?

1.1 Related Work

RQ1/RQ2 – A number of studies of the PyPI service, such as Ruohonen et al. [1] and Alfadel et al. [3], using static and empirical analysis respectively, have detected security issues in the repository's packages. The latter study states that vulnerabilities in the PyPI ecosystem take approximately three years to be discovered. A similar problem was found in the JavaScript equivalent of PyPI, the Node Package Manager (NPM). However, Alfadel et al. found that a majority of PyPI vulnerabilities were fixed only after being publicly announced, which was not an issue in NPM. They pose that this is likely due to the fact that NPM gives package developers 45 days to fix a vulnerability before it is published.

Bagmar et al. [4] performed an overarching study of security threats in PyPI's ecosystem, and found numerous possible exploits, such as arbitrary code execution via package setup files, or the previously mentioned typosquatting attacks. They even noted that some package owners uploaded the same package under variations of the original name, in order to prevent typosquatting attacks from occurring. The paper also mentions how many packages wrongfully use a more open, less protective, license, than the license specified by many of the packages imported by them. That is, if package `x` has a less protective license than that of `y`, it should not import package `y`.

RQ3 – A study by Vu et al. [2] reviewed possible strategies used by attackers to deploy malicious code in the PyPI repository. They suggested a couple of approaches for detecting packages of a malicious nature, such as using string comparison algorithms to discover typosquatting attacks. Notably, they did not find it to be a flawless technique, as the chance of a false positive is high given the large number of packages that exist in PyPI (currently over 450,000). They also suggested looking at the package's linked GitHub repository, and comparing names between the two. However, the false positive rate is relevant here as well, as some packages share the same GitHub repository, and some legit packages use different names between the GitHub and PyPI repositories.

A similar study was performed by Kaplan et al. [5], where common threats in both PyPI and JavaScript's NPM were surveyed. The conclusions they reach are comparable to the previously mentioned paper, but they proposed some further solutions to the issues. Firstly, they recommend a sort of reviewing system for users of the service, where users (or at least, some privileged members) of the service can vote on the validity of packages. Secondly, they proposed that everyone that maintains a popular library should be required to use some sort of multi-factor authentication. Finally, Machine Learning (ML) is put forward as a potential deterrent to typosquatting and similar attacks related to malicious package code. ML would remove the need for a large moderation team but has other difficulties involved with it.

1.2 Background

This section gives a short overview of the important concepts involved with PyPI, as well as the index itself. Furthermore, some comments on ensuring package legitimacy are given.

1.2.1 Packages, Modules, and Libraries

There are three main ways of describing code or code bases in Python: as packages, modules or libraries [6]. A *module* refers to any single Python `.py` file, which may include related functions, classes, variables, and the like. A *package* is often a larger collection of related modules or smaller *subpackages*, which can be imported into other modules to make use of their functionality. Packages are explained in further detail in Section 2.1. The term *library* refers to a large collection of packages, but can be used interchangeably with package, as the PyPI service does not differentiate between the two. As such, this thesis will refer to libraries as packages.

However, there are also built-in functionalities in Python, such as basic mathematics and operating system control, that do not require installation from an external resource. These are known as the standard libraries, and will be referred to as such.

1.2.2 The Python Package Index

PyPI is Python's main code repository for sharing software packages used in Python programs. As mentioned, it is owned by the Python Software Foundation, and is developed and maintained by the Python community. At the time of writing, there is a calculated 15.6 TB of package data in the index [7], and the number of packages continues to increase. PyPI's functionality is similar to that of JavaScript's NPM or Ruby's RubyGems repository, and packages are usually downloaded via the Python package installer program, `pip`.

The community in charge of PyPI is a group of developers called the Python Packaging Authority (PyPA) [2]. They are the moderators and administrators of the repository. Notably, they are a relatively small group when compared to the mentioned continually growing size of PyPI, meaning that administration of new and old packages is severely limited.

Uploaded packages on the PyPI website often link to their respective source code on GitHub, but there is no inherent requirement to do so.

1.2.3 Project Legitimacy

Recognising an illegitimate package is not a trivial problem. However, there are some aspects that are often shared by established Python packages, which can likely be used to distinguish them from malicious ones. For example, Vu et al [2] observed that an established package often keeps the same package name as the respective name for the related GitHub page, which contains the package source code. As mentioned, the GitHub link is not a requirement in PyPI, but it is a convention used by most established packages.

Given this fact, some information on project legitimacy could likely be gleaned from looking at the GitHub page of a given project. Is the name the same? Are there many users (GitHub stars, forks)? Does the description list expected functionality?

To avoid malicious packages, especially of the typosquatting variety, looking at the available information could help greatly. The project's PyPI page, in conjunction with the mentioned GitHub details, grants some information, but other statistics could also be useful. Some tools for this information are discussed in Section 2.5 below.

1.3 Potential Security Threats

Aside from the issue of vulnerabilities as a consequence of poor coding and the like, this thesis focuses on how malicious actors might utilise PyPI. Ohm et al. [8] describe two main strategies used to inject malicious code into a PyPI-like ecosystem: infecting an existing package, or submitting a new package. For the first strategy, the malicious actor has to somehow gain access to an already established, legitimate project, in order to inject their code into the package. In PyPI's case, this could be done by gaining access to a package's managing PyPI account, and uploading an altered version of that package.

The second strategy is easier to perform, but makes it harder to effectively spread the malicious code, compared to taking over an established code base. A common attack based on this strategy is the previously mentioned typosquatting attack. The concept is discussed further below.

1.3.1 Typosquatting

Typosquatting is a malicious activity with the goal of tricking users into downloading potentially malicious code. They are similar to spoofing attacks, i.e. attacks where data or some access points are disguised as legitimate sources. Typosquatting works by using names and titles derived from established, trusted sources, relying on either the user making a typo while accessing the source (looking for the trusted source, but finding the malicious one), or tricking the user by hoping that they do not discover the name discrepancy in, for example, the list of imported packages in some source code.

Vu et al. [2] performed an analysis of PyPI with the purpose of detecting potential typosquatting attacks, and found 67,000 packages that could not be confirmed safe without further investigation. That is, the packages had a name in close proximity to the name of another package on PyPI. Their paper identified a number of different methods that can be employed in typosquatting attacks. For example, a package pretending to be the `numpy` package, could be named `mumpy`. The alteration strategies include swapping characters, adding new characters, using similar characters, etc.

1.3.2 Possible Attacker Actions

A number of threats are viable when it comes to compromised or malicious packages. These packages can, for example:

- steal user credentials, such as SSH keys or passwords [9],
- swap out the infected user's Bitcoin address for the attackers, rerouting payments or transfers [10],
- install dependencies to take control of the user's mouse and keyboard, and take screenshots [11],
- install a trojan, such as the W4SP Stealer [12], or
- install a crypto miner on an infected user's device [13].

These attacker actions can be applied to potential typosquatting packages, as well as already established packages that have been breached through other means.

Furthermore, while the vulnerabilities of a package `x` of course affect package `x` itself, it could also affect package `y`, if `x` is a dependency of `y`. Using this fact to perform attacks on packages further up a chain is called a *supply chain attack* [8]. No matter how secure the main package is, guaranteeing the safety of every single one of its dependencies (as well as their dependencies, and so on) is very difficult. This makes supply chain attacks a common problem in software ecosystems such as PyPI or NPM.

2

Theory

This chapter focuses on the paper's most important theoretical and technical aspects, such as the threat modelling process, string comparison algorithms, and potential security tools.

2.1 The Package Upload Process

While the package installation procedure is quite fast, simply run the `pip install` command to install a specified package along with the package's listed dependencies, the package upload process is a bit more advanced. It is done using the two Python tools, `build` and `twine`, that respectively builds and uploads the package [14]. The initial package creation and uploading process is summarised in the following steps:

1. Create a folder containing a `src` folder, `pyproject` file, a license file and a `readme` file.
2. Generate a distribution package using the `build` module.
3. Upload the distribution package to PyPI, or to TestPyPI for testing purposes, using the `twine` module. A PyPI account is required, as login credentials will be prompted.

The `src` folder contains the actual package code (another name can be used for it), and `pyproject` is a configuration file for the metadata of the package (package name, authors, version, build library to use, etc). Furthermore, the `src` folder must contain an `__init__.py` file, recommended to be empty, for the directory to be recognised as a package.

When a package is uploaded, it is called a *release*, and the only way to update the code is to upload a new release. Each notable event, such as project creation or the upload of a new release, is recorded in the "security history" log of the package.

While the above-mentioned process can be used, package creators oftentimes want to create more advanced configurations. To do this, many developers utilise the `setuptools` build library, as provided by PyPA. Using `setuptools`, metadata, dependencies, and the like can be placed either in the mentioned `pyproject` file, or alternatively in a separate `setup.cfg` or `setup.py` file [15]. The latter option can potentially be used to introduce malicious code into the package installation procedure, which is discussed further in Section 2.4.

There are two roles that PyPI users can take when associated with a package: package owner and package maintainer. The latter is not allowed to delete files, releases or the entire package, nor are they allowed to invite new collaborators.

2.2 Threat Modelling

The act of threat modelling is mainly a structured method of identifying and classifying threats against an application or service. Threat modelling often looks at a system from the point of view of an attacker, rather than a defender, with the goal of building an understanding of the system and the solutions needed to stop potential threats [16].

While there are many different takes on threat modelling, the method used in this thesis is based on the OWASP foundation's guide on the subject [17]. The guide describes a three-step process, which begins by decomposing the application in question, then determines and ranks potential threats. Finally, possible countermeasures for the discovered threats are listed and reviewed.

2.2.1 Decomposing The Application

The first step of the threat modelling process is the decomposition of the application, which is done to achieve an understanding of the application, along with its possible interactions with users. This is done by creating a number of use cases to describe the interactions with the application of choice, identifying entry points and assets that would be of interest to an attacker, and describing how the access rights are distributed between related entities. The information gathered can then be used to create a Data Flow Diagram (DFD) to get an overview of the application's composition. In short, the decomposition tries to extract the following from the system:

- **Use Cases** – What is the application used for?
- **External Dependencies** – Items that lie outside of the actual application, but could still pose a threat if compromised, e.g. database servers, physical servers. They are external, but are still controlled by the organisation owning the system.
- **Entry Points** – Refers to the interfaces which could be potentially used by attackers to interact with the system, or supply it with data. E.g. login pages, upload commands.
- **Exit Points** – Instead refers to the use of data outputs, which are often related to some entry point. For example, abusing a cross-site-scripting vulnerability by using some forum posting functionality. The ability to write the post is the entry point, while the output is the exit point.

- **Assets** – Physical or abstract entities that are potential targets for an outside attacker. It could refer to something simple like personal details or login information, or something more abstract like the ability to access a web page.
- **Trust Levels** – Represents levels of access rights granted to entities and actors by the system. These are used to establish which users have adequate rights to access the entry points and assets of the system. Some examples are administrators, or users with valid login credentials.
- **Data Flow Diagrams** – A visual presentation of how the data flows through the system, with the entities and processes involved.

Notably, the threat model process is usually done by the organization that owns the system being analyzed, but the model done in this thesis is instead made from a PyPI user's point of view. In short, this means that some internal entry points, external dependencies, and similar facts might be unknown to us, such as how PyPI's web servers or databases function. But, such information is also unknown to potential attackers, meaning there is still merit in using the model from an outside perspective.

2.2.2 Threat Categorisation

The Microsoft-developed STRIDE is one of the main threat categorisation models recommended by the OWASP guide. It is based on the confidentiality, integrity, and availability (CIA) triad, with the addition of three more elements: authentication, nonrepudiation, and authorisation [16]. STRIDE focuses on security threats in six main categories, each associated with one of the mentioned elements of the extended CIA triad:

- **S**poofing – Attempts to access and use another user's data, usually usernames and passwords, by utilising faked information.
- **T**ampering – Attempts to maliciously alter data in an application.
- **R**epudiation – Performing potentially prohibited actions in a system, where those actions cannot be traced to their origin.
- **I**nformation Disclosure – Attempts to read data without required clearance.
- **D**enial of Service (DoS) – Deny the users of the application access to some or all of its resources.
- **E**levation of privilege – Attempts to gain a more privileged access to a system, to reach what was previously unauthorized data.

2.2.3 Threat Ranking

Following the identification of threats within the above-mentioned categories, a threat classification model called DREAD is used to determine the different threats' ranks. DREAD is a value-based risk-rating model focusing on five evaluation categories [16]:

- **D**amage – What degree of damage can the threat achieve? A higher value represents a greater potential degree of damage.
- **R**eproducibility – Can the threat be easily reproduced? A higher value represents an easily reproducible attack.
- **E**xploitability – How easy is it to exploit the vulnerability? A higher value represents an easier exploit to perform.
- **A**ffected Users – How many users are affected? A higher value represents a larger number of affected users.
- **D**iscoverability – How likely is the vulnerability to be discovered by a potential attacker? A higher value represents a threat that is easier to discover.

For every identified threat, a score is given for each of the mentioned categories, after which the average of the scores becomes the overall DREAD score for that threat. Importantly, the DREAD model is subjective, meaning that its scoring depends highly on the individual using the model. However, DREAD can still be used to create an approximate risk evaluation, which could help with prioritisation in the coming threat modelling steps about countermeasures and mitigation [16].

2.3 String Comparison Algorithms

There exist a number of possible string comparison algorithms. The following four are the most commonly used ones [18]:

- **Levenshtein Distance** – Counts the number of substitutions, deletions, and insertions needed to convert one string to the other.
- **Hamming Distance** – Counts only substitutions, and can thus only be used on strings of equal length.
- **Episode Distance** – Looks only at insertions, and is thus not symmetrical, meaning string `x` might not always be convertible to string `y`.
- **Longest Common Subsequence Distance** – Counts only insertions and deletions, i.e. the resulting distance is the number of unpaired characters.

In all of the algorithms above, each operation needed has a cost of 1 for calculating the distance between strings.

The Levenshtein Distance is the most widely used, and the one utilized in the analysis of PyPI by Vu et al. [2] to identify packages with similar names in the repository, i.e. to discover potential typosquatting attacks. The Levenshtein distance is, in short, the minimum number of single-character edits needed to convert one of the strings to the other [19]. These edits can, as mentioned, be insertions, deletions, or substitutions, which gives it a wider reach than the other algorithms. The Levenshtein distance algorithm looks as follows:

$$lev(a,b) = \begin{cases} |a|, & \text{if } |b| = 0, \\ |b|, & \text{if } |a| = 0, \\ lev(tail(a), tail(b)), & \text{if } a[0] = b[0], \\ 1 + min \begin{cases} lev(tail(a), b) \\ lev(a, tail(b)) \\ lev(tail(a), tail(b)) \end{cases} & \text{otherwise,} \end{cases}$$

where `(a,b)` refers to the strings, and `tail` is a function that omits the first character of the given string.

Given two strings `a` and `b`, the `lev`-function first checks if either string has a length of 0, in which case the length of the other string is the current distance between them. If the first character of each string is equal, the `lev`-function is called recursively with the tails of both strings. Otherwise, recursively check which of the possible string configurations (`tail(a)`, `tail(b)` or both) grants the lowest possible distance.

2.4 Malicious Setup Scripts

While the usage of malicious package code is dangerous on its own, even the act of downloading or installing the package could cause potential harm. This is largely due to the previously mentioned `setup.py` files that are often found in packages, and are used to list dependencies and other metadata. The problem lies in the fact that the `setup.py` file is a regular Python file, and can thus contain any code the creator wants to put in [20]. Many of the attacks mentioned in Section 1.3.2 are possible to perform here, such as installing trojans or stealing data.

The `setup.py` file usually contains some import lines (to import the necessary setup functionality) and a `setup()` function, where the latter contains the mentioned metadata [21]. As such, this file is meant to run on installation to setup the package. Notably, PyPI has introduced a way of packaging the build of the package within a `.whl` file, or wheel file. In a wheel file, the `setup.py` file's effects are already recorded, and will therefore not actually run on a user's device upon installation.

However, the old way of packaging, using `.tar.gz` files, have not been phased out. New packages are not required to use the wheel file format, and old packages might not yet have been updated to it. As such, there is nothing stopping a potentially malicious actor from uploading a package using the old format, where the `setup.py` is guaranteed to be executed on some user's device.

2.5 Potential Package Security Tools

There are a large number of tools available to decrease the risk of security threats from Python packages. They can do this through means of information gathering, vulnerability scanning, vulnerability databases, etc. A few possible tools are described in this section.

Safety DB – Safety DB is a database of currently known vulnerabilities of Python packages, which is synced with the data of cybersecurity company PyUp once a month [22]. It contains a list of historically or currently vulnerable packages, what CVE id (Common Vulnerabilities and Exposures) the vulnerability has, and what versions of the package are affected. Using the CVE id, the exact details of the vulnerability can be found.

Importantly, Safety DB also states that the list is not a “hall of shame”, i.e. the list should not necessarily be used to exclude packages for having had historical vulnerabilities. Instead, the list should be used for information gathering purposes.

PyPI Stats – Another way of information gathering is the PyPI Stats service, which grants download statistics for every PyPI package [23]. It provides daily, weekly, and monthly download rates for packages, that are sourced from download stats at Google BigQuery. While this seems like redundant information, it could possibly be used to determine the legitimacy of some packages, especially in cases of potential typosquatting attacks. If a package is trying to imitate, say, the contemporary NumPy package, seeing a low download rate should be a clear warning sign.

Regular Expressions – A way of matching patterns in text, used by word processors, search engines, password/e-mail input fields, and many other applications. Regular Expressions (Regex) could likely also be used to pattern match source files, such as the mentioned `setup.py` file. It could be used to detect anything outside the standard `setup.py` file of the PyPI documentation [21]. However, the Regex will have to be quite long to catch every possible case, and the chance of a false positive/negative is always present.

Docker – To avoid the mentioned hazards involved with installing potentially malicious packages, some sort of containerisation is needed. The Docker platform could be a suitable choice for this, as it can be used to run software in a limited environment, called a *container* [24]. The enveloping environment, i.e. the operating systems and the configurations, of the container is based on a pre-built *image*. Docker containers work similarly to virtual machines, but take up less space, both in memory and in computational needs.

Bandit – Vulnerability scanning of code is not a trivial problem, but it can be done to some degree using the Bandit security scanner [25]. Bandit uses the so-called AST module (Abstract Syntax Trees) to convert Python code into syntax trees, that can be statically scanned for particular patterns. The Bandit scanner looks for patterns that can lead to potential vulnerabilities, such as the code executing a new process with a shell, or hardcoded passwords [26]. Each vulnerability is connected to a particular *test*, and new ones can be added to Bandit at the user's discretion.

3

Methods

The focus of this chapter is to describe how the threat modelling of PyPI will be done, i.e. the steps taken to create the model. Furthermore, this section describes how the scanning program will be designed, its main components, and the tools it uses. The first section is focused on RQ1 and RQ2, while the second section describing the scanner program is related to RQ3.

3.1 Threat Modelling PyPI

This section describes how the OWASP threat model, as presented in Section 2.2, is used to model the PyPI repository.

Decomposing PyPI – The use cases of PyPI are listed, possible entry points for external threats are pinpointed, the involved assets and entities are identified, and the distribution of access rights among those entities is explored. The information gained is then used to create a DFD of the system, to show an overview of the repository and its functionality.

Decomposition begins with a list of PyPI's different use cases, after which a table of trust levels, i.e. the roles of actors within the system, is created. Then, the possible entry points of PyPI are listed, i.e. interfaces of the application where potential attackers can interact with the system or supply it with data. Finally, the assets of the system are listed. The OWASP guide also mentions external dependencies, but they are mostly omitted as we have little information about those aspects of PyPI's workings, seeing that we are making the model from a user's perspective.

Threat Categorisation – The STRIDE framework is used to classify the discovered threats in accordance with Section 2.2.2. They are listed in a table under each relevant STRIDE category.

Threat Ranking – For each threat, each of the five DREAD categories is given a point from 1–10 depending on their severity, after which they can be ranked accordingly. As mentioned, these are subjective ratings, but it is a helpful way of creating an approximate ranking system.

Countermeasures and Mitigation – When threats have been established, classified, and ranked, they can be mapped to possible countermeasures. These mappings are presented in a table, after which they are to be analysed regarding their plausibility in the PyPI system, i.e. are the solutions tenable?

3.2 The Scanning Program

The scanning program features three main components, which are all to be run within a containerised application, using Docker. It takes a single package's name as an input, and will output a report based on findings from the three components. Other than the main package itself, the scanner will also gather some information on the package's dependencies, i.e. its list of required packages. As this is a proof-of-concept program, it will only look at the first level of dependencies, i.e. the dependencies of the main package. Notably, the program itself is implemented in the Python programming language.

The components are:

- **The Package Scanner** – A component with a focus on downloading and installing the main package as well as its dependencies, and then using some tools to gather information on them locally.
- **The Typosquat Scanner** – This component focuses entirely on comparing the similarity between the names of the packages with a list of contemporary packages' names, as well as names of Python standard libraries.
- **The Web Scanner** – A component with the purpose of collecting information from web resources.

Below, each of the main components, as well as the containerisation itself, is explained.

3.2.1 The Package Scanner

As mentioned, the package scanner looks at downloading, installing, and analysing the files locally. For this, a number of tools are used:

pip install/download/show – Other than the mentioned `pip install` command, there are plenty more `pip`-related commands. The two relevant ones (other than the installation command) are `pip download` and `pip show`, where the first downloads the package (along with dependencies) rather than installing it, and the latter gives some information about an installed package. The information gained through `pip show` includes name, current version, homepage, author, license, a list of required packages, and more.

These three commands are used to gain more insight into the main package. Initially, `pip install` is used to install the main package, as having the package installed is a requirement for the `pip show` command. Using the latter command, we can extract some package information, which is an important part of the information gathering process, as well as the main way of getting the package's list of dependencies.

For the final part of the package scanner's functionality, the purpose is to look for potentially malicious `setup.py` files. Thus, we use the `pip download` command to download the main package, along with its required packages, to a local folder for convenience. After a successful download, the folder now contains some compressed packages, either in the form of the newer `.whl` wheel files, or the older `tar.gz` files. As the `.whl` files are supposedly safe from the dangers of setup scripts, the scanner will unpack and analyse only the non-wheel `tar.gz` files.

Regular Expressions – After extraction, each relevant package is scanned for its `setup.py` file, which is then sent to a Regex pattern that accepts files in accordance with PyPI's guidelines on setup scripts, as explained in Section 2.4. If the files contain anything outside of the expected pattern, this is noted by the scanner.

Thus, the results of the package scanner contain gathered information on the main package, lists of which packages were wheel or non-wheel files, and information on which of the non-wheel packages contained a non-standard `setup.py` file.

3.2.2 The Typosquat Scanner

The typosquat scanner has one purpose, looking at if the given package or its dependencies may be part of a typosquatting attack. In order to do this, it has to compare the names of the packages with names of commonly used packages, or the names of standard libraries.

Names of common packages are gathered using a list of the 5000 most downloaded packages over the last 30 days, collected online monthly by Kemenade et al. [27]. The names of Python standard libraries come from the `stdlib-list` package, which lists each standard library for a given Python version.

The string comparison itself is done using the Levensthein distance, as explained in Section 2.3. Thus, the `distance` function of the `Levensthein` package is used on each pair of main package/required package and common package name/standard library. Two names are considered similar if the Levenshtein distance between them is less than or equal to two. A higher distance threshold could drastically increase the risks of false positives (as discussed by Vu et al. [2]).

When the typosquat scanner is finished, it returns a report listing the names of the main package/dependencies as well as a list of found similar package names.

3.2.3 The Web Scanner

The final component of the scanner program looks at information gathering using web-based resources. The following tools are utilised:

PyPI Stats – As mentioned in Section 2.5, PyPI Stats grants download statistics for a given package. Through the `PyPIStats` package we get access to those download rates, so we collect the monthly download rate for each relevant package, i.e. the main package and its required packages.

Safety DB – The scanner looks up each relevant package in the vulnerability database, and extracts information on previous vulnerabilities, i.e. their CVE id and which versions were affected.

Web Scraping/Github API – For more information gathering, we turn to the related Github page of each relevant package (if there exists one). However, the package name on PyPI does not always correspond to the package name in Github, so the web scraping `BeautifulSoup4` package is used to scan a package's PyPI web page for their Github API link. When the link has been retrieved, a lot of information can be obtained, such as the size of the project, when it was created, when it was updated last, the current number of Github stars, etc. Not all packages have a linked Github, but as that is usually the convention, the lack of a link is also useful information. When finished, the web scanner returns the collected information on download rates, vulnerabilities, and Github-based data.

3.2.4 Containerisation

As mentioned, the scanner program is made to be run in a Docker container to avoid contamination of the main device. Docker offers many different images, but the scanner program image is based on the Python image [28], which essentially is a Linux Debian operating system with Python tools installed. From the Python image, we build the scanner program image, where the main difference is the installation of the mentioned packages used by the scanner program (such as the `Levenshtein` and `Beautifulsoup4` packages).

With the image complete, a new Docker container based on the image can easily be supplied with a package to scan, and be started. Using the `Docker` Python package, this is simple to set up, and the resulting output of the container is collected for the user. Afterwards, the container can be removed, and the possible side-effects of any of the package's installed packages will likely not affect the device outside. Figure 3.1 shows this containerisation visually.

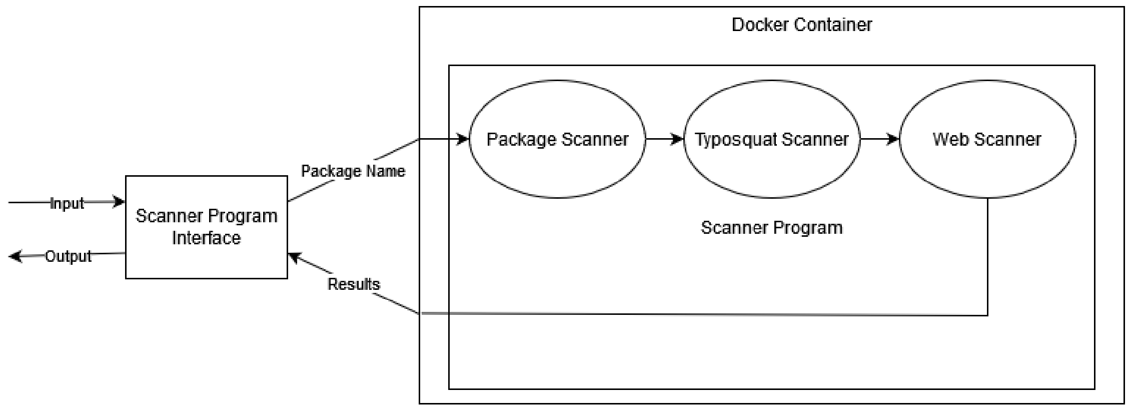

Figure 3.1: Containerisation of the scanner program. The interface's input is the name of a package, which is handed to the program within the Docker container. In the container, each part performs its functions, after which the resulting report is sent back as the program's output.

3.2.5 Program Evaluation

The scanner program will be evaluated by running it on a number of packages, and looking at the level of information gathering achieved. 100 of the most recently updated packages (as of writing), which are compatible with Python version 3 were chosen as the testing dataset. The resulting data is then evaluated in order to see if the metrics used are viable for testing package legitimacy.

4

PyPI Threat Model

This chapter documents the PyPI threat model, listing the results of each of its separate steps. That is, the PyPI decomposition, threat categorisation and ranking, and finally the listing of potential countermeasures.

4.1 Decomposition of PyPI

Here, the use cases, tables and diagrams related to the decomposition of PyPI are presented. While some of these tables, the assets table especially, can be extended to great proportions, they have been limited to the most relevant entries for a thesis of this scale.

Use Cases – PyPI has a number of relevant use cases for its functionality as a service, which mirrors the functionality of similar code repositories:

- Prospective users should be able to create a PyPI account using PyPI's web service.
- Python developers with a PyPI account should be able to upload Python code packages to the repository, for sharing.
- Prospective developers should be able to download packaged code from the repository to use for themselves.
- Package owners should be able to manage their packages via the web interface.

Trust Levels – Table 4.1 features the essential trust levels of the PyPI service. The anonymous user entry refers to the conventional user of PyPI, i.e. a Python developer that installs some needed package via `pip`. A PyPI user, i.e. a user with no uploaded packages, largely has the same capabilities as the anonymous user, until they upload a package and become a package owner, as the main point of a PyPI account is to upload and manage packages. The PyPA administrator can reasonably remove any package, which they are to do if it is deemed malicious, while a package owner can only remove their own packages.

Table 4.1: The different entity trust levels of PyPI.

Trust Levels	
Name	Description
Anonymous User	Any user with a device able to connect to PyPI
PyPI User	A User with a PyPI account
Package Maintainer	PyPI account with upload permissions for a package
Package Owner	PyPI account owning a package/packages uploaded to the index
PyPA Administrator	PyPA member with administrative powers

Entry Points – The essential entry points of PyPI have been listed in Table 4.2. Each table entry contains a related trust level, i.e. which entities of the trust level table (Table 4.1) have potential access to that specific entry point. Uploaded package contents is an entry point both to the PyPI system itself, and to the users downloading the package.

Table 4.2: Entry points of PyPI, with a short description and relevant trust levels.

Entry Points		
Name	Description	Trust Levels
PyPI Login Page	PyPI's web-based login page	All
`twine` Upload Command	Command used to upload a package to the index	PyPI User, Package Owner, Package Maintainer, PyPA Administrator
Uploaded Package Contents	The content of an uploaded package	Package Owner, Package Maintainer, PyPA Administrator

Exit Points – The main exit point that exists in PyPI is the login prompt, which can return some informative error messages, i.e. "No user found with that name" when using a nonexistent username, or "The password is invalid" when the account exists, but the password is wrong. The same return messages are not given upon using the `twine` upload command, which only states "Invalid or non-existent authentication information".

Assets – The assets, i.e. the areas of interest for attackers of the system, are listed in Table 4.3. Like the entry points table, the entries of this table also contain the relevant trust level entities with access to the particular asset. As mentioned in Section 2.1, the ability to upload releases is shared between package owner and package maintainer, but only the former can remove packages or releases from a project.

It is important to note that release or full project deletion is irreversible according to the PyPI package management page. After project deletion, the name of the project is made available to any other PyPI users, and reuploading the same package again (containing the same filenames) is not allowed. Thus, if a package is accidentally or maliciously deleted, undoing the damage is quite difficult.

Table 4.3: The assets of PyPI with a short description and relevant trust levels.

Assets		
Name	Description	Trust Levels
User Details	Account details related to some user	PyPI User, Package Owner, PyPA Administrator
Package Contents	Access to the content of some package on PyPI	All
Ability to Upload Releases	The ability to upload a new release of some package	Package Owner, Package Maintainer
Ability to Remove Package	The ability to remove a package/release	PyPA Administrator, Package Owner
Ability to Upload Package	The ability to upload a new package to the repository	PyPI User, Package Owner, Package Maintainer

Data Flow Diagram – The DFD in Figure 4.1 shows an overview of the interactions between the PyPI service and its users. The dotted lines represent privilege boundaries where a change of trust level is needed to perform the interaction.

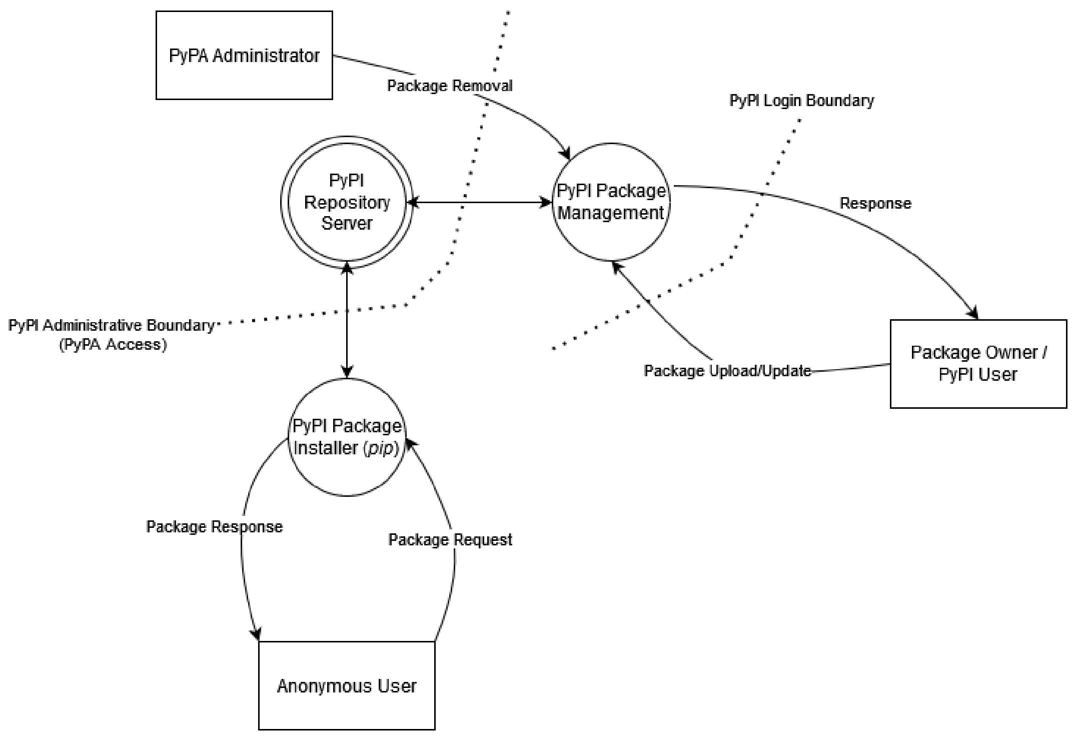

Figure 4.1: DFD over interactions with the PyPI service. Trust level boundaries are marked by dotted lines.

4.2 Threat Categorisation

As for the STRIDE threat categorisation part of the model, Table 4.4 states the categories and the most related element from the PyPI decomposition. For each category, the possible threats against the selected elements will be explained further below.

Table 4.4: STRIDE categories and their related elements from the PyPI decomposition above.

STRIDE–Related Elements		
STRIDE Category	Entry Points	Assets
Spoofing	PyPI Login Page	User Details, Ability to Upload Package
Tampering	-	Ability to Alter Package
Repudiation	-	Ability to Upload Releases, Ability to Remove Package, Ability to Upload Package
Information Disclosure	PyPI Login Page	User Details
Denial of Service	`twine` Upload Command	Ability to Upload Releases, Ability to Remove Package, Ability to Upload Package
Elevation of Privilege	PyPI Login Page	All

Spoofing – There are many avenues for spoofing attacks to take. The previously mentioned typosquatting attacks are an example of a sort of spoofing. Spoofing user details is another option, and using them to try to bargain for access to some package contents. A potentially more successful version of that is to somehow steal credentials belonging to some PyPI user, or package owner, and use the new trust level for malicious means.

Tampering – Tampering attacks can come in many different forms in the PyPI service, and are mainly connected to the contents of packages. For example, a compromised package owner account could have their packages tampered with for malicious means, such as data extraction from users of the package (by uploading a new release containing the manipulated code). Tampering with the package code to introduce security bugs, such as an SQL injection vulnerability, is another possible risk.

Repudiation – As mentioned in Section 2.1, each PyPI package logs potential security events in the security history of the project. This makes repudiation on a package-based level difficult, as alteration can be traced by time, collaborator, and IP address of whoever caused the change. However, if a package owner's or maintainer's account is breached, impersonation is possible.

Assuming that manipulation of logs is possible for PyPA administrators, a breach of such an account could lead to even greater risks for the system, as the attacker could easily hide their tracks.

Information Disclosure – As a result of, for example, a tampering attack, information disclosure becomes a major threat. A tampered package could be used to gain access to users' data, such as files and passwords. Information disclosure is also a risk in typosquatting attacks, with the same potential threat to user data.

The previously mentioned exit point in the PyPI login page can be used to gain information on whether or not a username exists in the PyPI userbase. With this information, a potential attacker can look up that username in the many lists of previously breached passwords found online, and potentially gain access to that account.

Denial of Service – From the perspective of the PyPI service, a denial of service attack could potentially disrupt or severely limit all user interaction with the system. Doing something at that scale likely requires a high trust level (PyPA administrator or even higher), and is thus potentially a difficult attack to perform.

An easier option for malicious actors is to make the denial of service attack a part of, for example, a previous spoofing or tampering attack of some package's content. I.e. they could insert some disruptive code into the package, which upon activation would interfere with the device the code is running on. This latter method is a threat from the perspective of PyPI's users, rather than the service itself. Notably, as the removal of a package is deemed irreversible, maliciously removing a package would be considered denial of service as well.

Elevation of Privilege – Any attack with the goal of reaching a trust level with more access than before, e.g. package owner from anonymous user, is an elevation of privilege. With a new trust level, the attacker may gains access to new functionality, such as the ability to remove files or entire packages.

A summary of the main discovered threats is listed in Table 4.5.

Table 4.5: Discovered threats, a short description of them, and their related STRIDE categories.

Potential PyPI Threats		
Threats	Description	STRIDE Category
Typosquatting	A malicious package impersonating a commonly used one	Spoofing, Information Disclosure, Denial Of Service
Denial of Service	An attack with the purpose of preventing access to some resource, e.g. by removing a package	Denial of Service, Tampering
Repudiation	Performing an activity without leaving tracks	Repudiation, Tampering
Package Tampering	Upload a tampered release, with potential vulnerabilities or malicious code	Tampering, Denial of Service, Information Disclosure
Breached User Password	Breaching or using a previously breached account password to gain access to a package or user details	Elevation of Privilege, Information Disclosure, Denial of Service, Repudiation, Tampering

4.3 Threat Ranking

Table 4.6: The DREAD scoring from 1 to 10, and DREAD score average for each threat.

DREAD Scoring						
Threats	D	R	E	A	D	Average
Typosquatting	7	10	7	5	9	7.6
Denial of Service	9	6	3	8	4	6.0
Repudiation	3	4	3	5	3	3.6
Package Tampering	8	6	3	7	3	5.4
Breached User Password	7	3	5	3	5	4.6

The DREAD scores and their average are shown in Table 4.6, where the columns refer to the DREAD categories described in Section 2.2.3 (**D**amage, **R**eproducibility, **E**xploitability, **A**ffected Users, **D**iscoverability). A visualisation of the same data in stacked chart form is found in Figure 4.2.

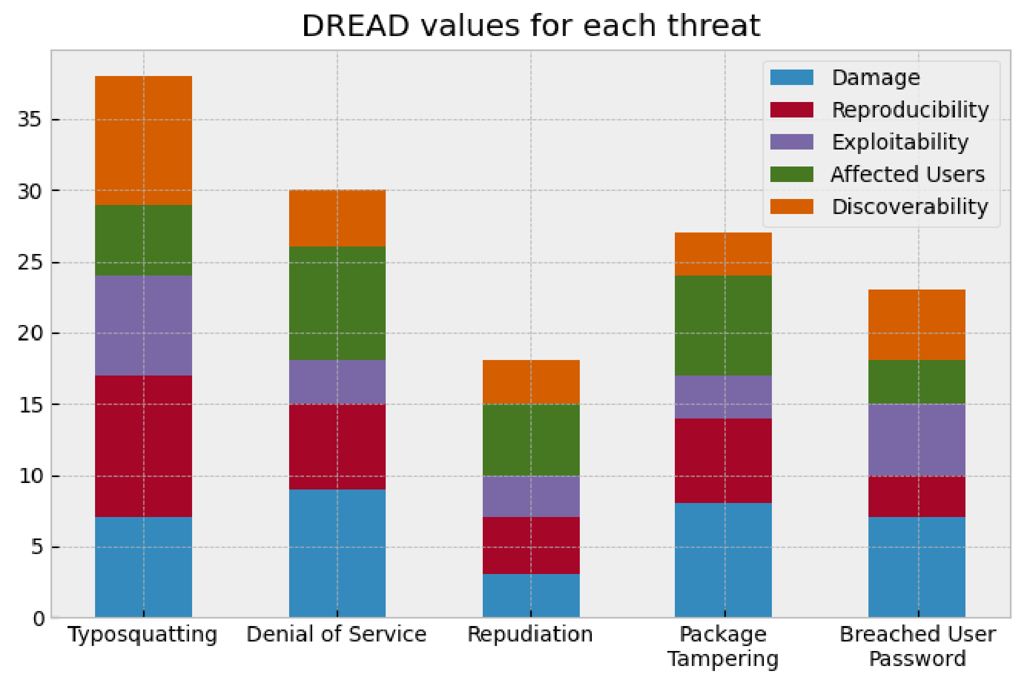

Figure 4.2: Stacked bar chart over DREAD values for each threat.

Typosquatting – The typosquatting threat has been given the highest score, mostly because of how easy the attack is to perform. All an attacker has to do is pick a previously established package name, change it slightly, and upload some malicious code under that name. Since the imitated package is as open-source as the rest of the available packages in PyPI, the attacker can even copy its code, and hide their alterations within.

Denial of Service – These attacks have a great potential for damage, as loss of access to a certain package, or even access to PyPI itself, would be a major detriment to most Python developers. However, such an attack is hard to pull off successfully. Deleting a package or release can only be done with at least a package owner trust level, and hindering access to PyPI itself is likely only available to PyPA administrators or higher.

Repudiation – As mentioned, the security history (logs) of each package makes it difficult to perform operations without it being logged. The most viable way of repudiation is by impersonating some package collaborator, owner or maintainer, and performing the malicious actions via their account. To avoid detection via the recorded IP address, an attacker can easily use Virtual Private Network software (VPN). A hijacked PyPA administrator can likely perform even more dangerous actions, but we cannot speak as to how difficult breaching such an account would be, as we are not privy to the internal workings of the PyPI system.

Package Tampering – As mentioned in Section 1.3.2, there are many possible variations of tampering that can take place in compromised packages, such as data harvesting or the installation of trojans. While tampering can cause a lot of damage, these attacks require package access to work. Thus, whether it is through social engineering (i.e. getting user credentials through social convincing/coercion) or account takeover, there are many steps to perform before tampering can ensue.

Breached User Password – A compromised account has the potential to lead to some of the other threats discussed, like denial of service or package tampering, but relies on the attacker successfully breaching the account. Bruteforcing an account password is difficult, as the PyPI login page disallows login attempts for a while after five unsuccessful attempts (this can be bypassed by VPN, but the process is slowed). Furthermore, the failed login attempts are logged in the profile, which could alert the user to the attempts.

An easier avenue of attack, is to look for external user database leaks (from other websites and services), where usernames and password combinations can be found, and test them out in the PyPI service in a sort of dictionary attack. This might not give access to a specific account, but chances are high that some other user reused a password.

4.4 Countermeasures and Mitigation

Table 4.7: The discovered threats, and their relevant countermeasures.

Mitigation Mapping	
Threat	Countermeasure
Typosquatting	Naming Limitations, Community Reporting Features
Denial of Service	SSH key Signature, Multi-Factor Authentication
Repudiation	Logging
Package Tampering	SSH key Signature, Multi-Factor Authentication, Package Code Scanning
Breached User Password	Multi-Factor Authentication

Typosquatting – The current naming conventions of PyPI allow any name that is not currently used by some other project, hence why typosquatting attacks are so easy to perform. Implementing some sort of naming limitation on very similar names is a possible countermeasure. Furthermore, there is currently no way of reporting possibly malicious packages directly from the package project's PyPI web page. The main way to report a security issue, as advertised on the PyPI web site, is to send an email to PyPI's security team with the necessary details. Adding such a feature to the web site would likely make questionable packages easier to detect.

Denial of Service – While multi-factor authentication exists in PyPI, it is not required for any account to activate it. Having it be mandatory for maintainers and package owners of the larger, more contemporary packages, which have many users, could minimise the chances of account breaches and the possible resulting attacks. Another solution is to introduce a GitHub-like SSH key signature system, where new releases of a package (that is, new updates) can only be uploaded from devices with an approved private SSH key.

Repudiation – Logging is the main solution to repudiation threats, other than using mentioned methods to keep user accounts safe. PyPI already features many logging features, both in accounts and projects.

Package Tampering – The methods discussed regarding denial of service apply here as well, with the possible addition of some methods for scanning uploaded package code for possible vulnerabilities.

Breached User Password – The main way of hindering user account breaches, other than adequate password requirements, is the use of the mentioned multi-factor authentication method.

5
The Scanning Program

The purpose of this chapter is to showcase some of the data gathered while using the scanner program tool on a set of packages. While a lot of data is gathered by the scanner program, only the central metrics, such as GitHub stars/forks and download rates, are discussed here. If this was a larger project, connections and implications from other data points could have been discussed more in-depth. Furthermore, correlations between the found data could have been more thoroughly analysed.

5.1 Scanner Program Test Results

In the test, 100 packages were scanned using the scanner program. Including the dependencies of each of the 100 packages, 312 packages in total were checked. Out of those packages, 54 were non-wheel files, and 47 of those had a non-standard `setup.py` file. Furthermore, 54 packages were found to have previous vulnerabilities, and 71 packages lacked an accessible GitHub API via their PyPI project page.

The typosquat scanner found that for 179 packages there were no similarly named packages in the lists used for comparison. As for the packages where similar names were detected, Figure 5.1 shows that most packages had less than 20 similar package names. Some outliers had over 40 similar package names. Narrowing the range to packages with less than 20 similar names, as seen in Figure 5.2, shows that a majority of packages are named similarly to 2–3 common packages/libraries. Over the results collected, the average number of similar names found for a given package was around 5. The median amount of similarly named packages was 2, and the standard deviation for the values was approximately 7.6.

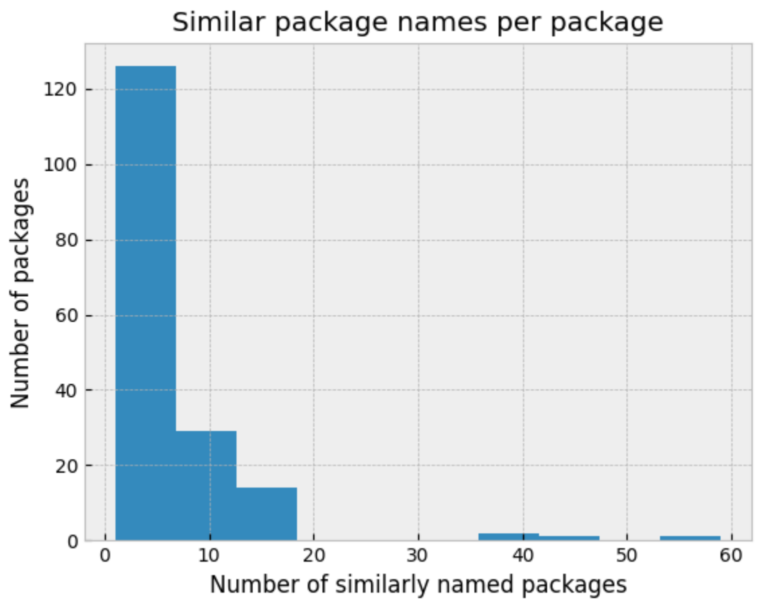

Figure 5.1: Amount of similar package names per package, as found by the typosquat scanner. This dataset does not include the packages which had no detected similarly named package.

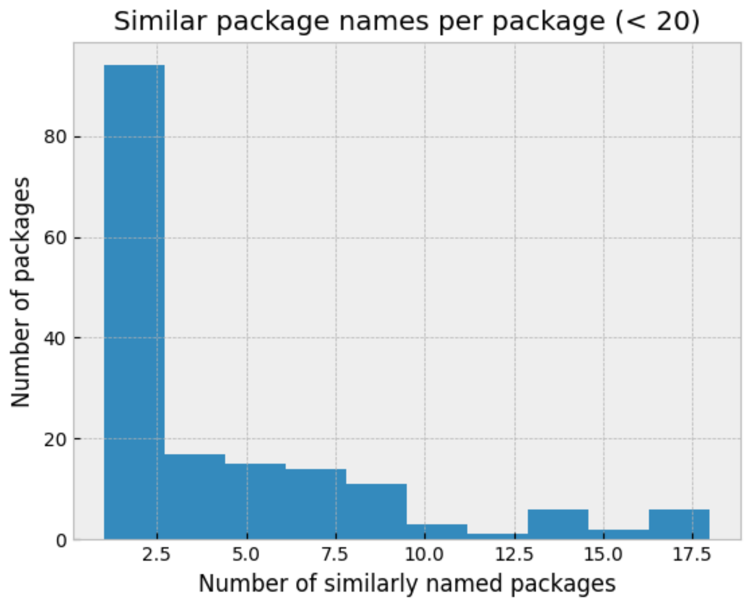

Figure 5.2: Amount of similar package names per package, using a more filtered dataset. Outliers are left out by limiting the dataset to packages with less than 20 similar names.

As for the web scanner, the GitHub stars information gathered from the packages with an accessible GitHub API, is shown in Figure 5.3. Once again, the graph is widened by outliers with many stars, so filtering out those with more than 75,000 stars gives us Figure 5.4. It is clear that a majority of packages checked have fewer than 10,000 stars. The average number of GitHub stars among checked packages was approximately 10,675. The median was circa 1245, and the standard deviation was roughly 21,355.

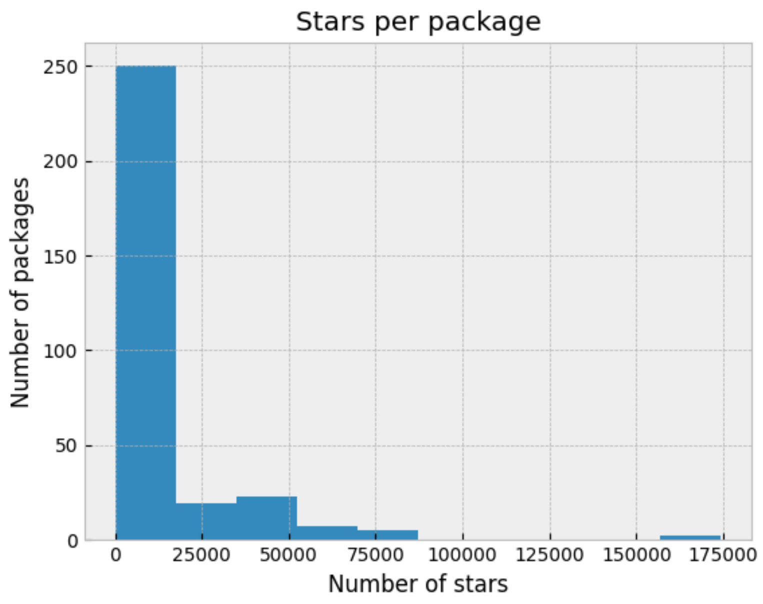

Figure 5.3: Amount of GitHub stars per package, as found by the web scanner. This dataset only includes the packages where a GitHub API was detected.

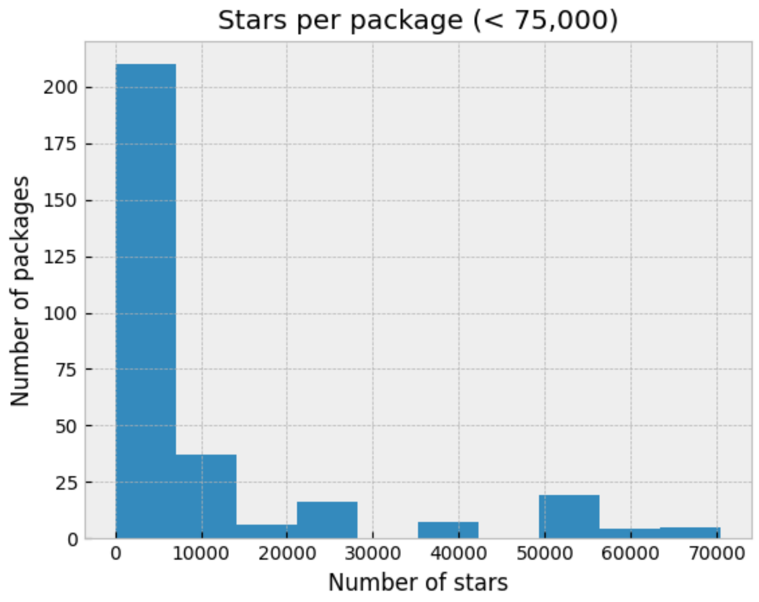

Figure 5.4: Amount of GitHub stars per package, using a more filtered dataset. Once again, outliers are excluded by limiting the dataset to packages with less than 75,000 stars.

A similar result is seen in the number of forks of a given package's GitHub project, shown in Figure 5.5. Filtering out the outliers with more than 15,000 forks, we get Figure 5.6, where it is clear that most packages have less than 1000 forks. The average amount of forks was approximately 3167, the median was circa 212, and the standard deviation was around 8872.

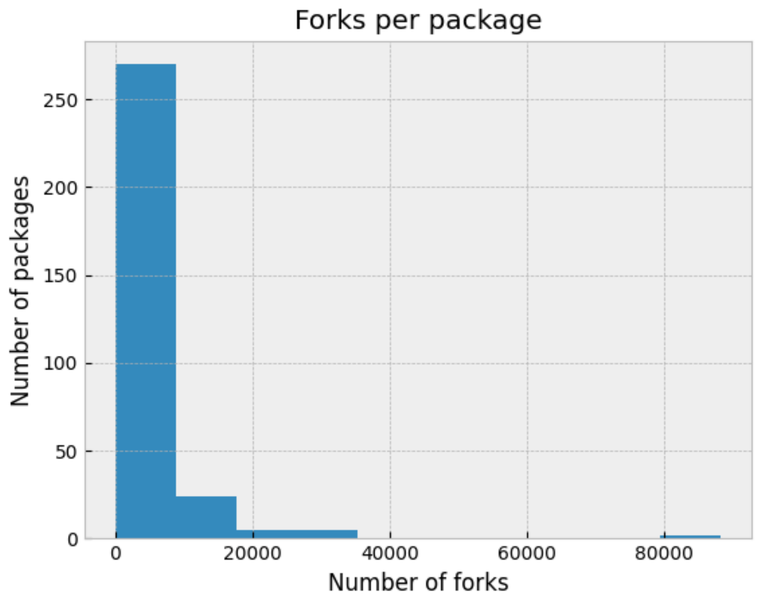

Figure 5.5: Amount of GitHub forks per package, as found by the web scanner. Similarly to Figure 5.3, this dataset only includes the packages where a GitHub API was detected.

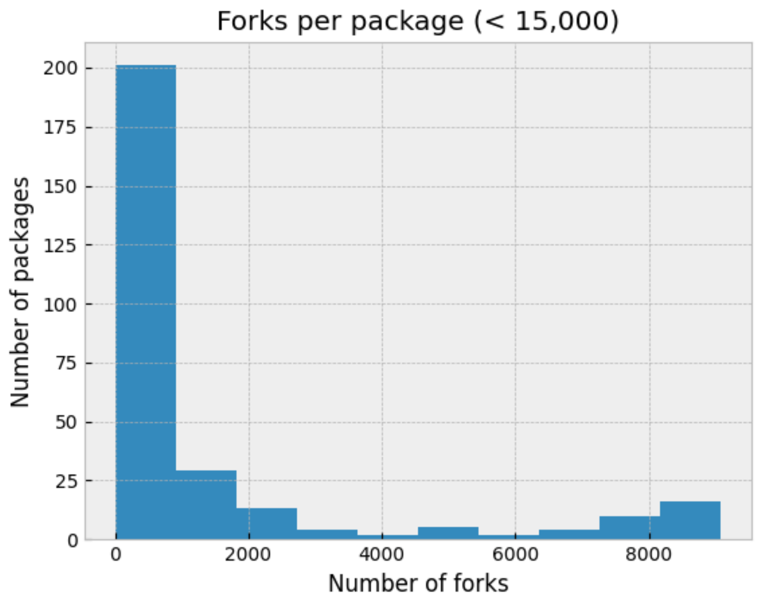

Figure 5.6: Amount of GitHub forks per package, with a filtered dataset. Outliers are excluded by limiting the dataset to packages with less than 15,000 forks.

The monthly download rates for the checked packages follow the same trend of a few outliers, and the majority having a comparatively low download rate. Figure 5.7 features every packages, while Figure 5.8 is limited to those below 1,000,000 downloads.

The download rate average was found to be around 17,360,608, with a median of 125,900, and a massive standard deviation of circa 46,671,483.

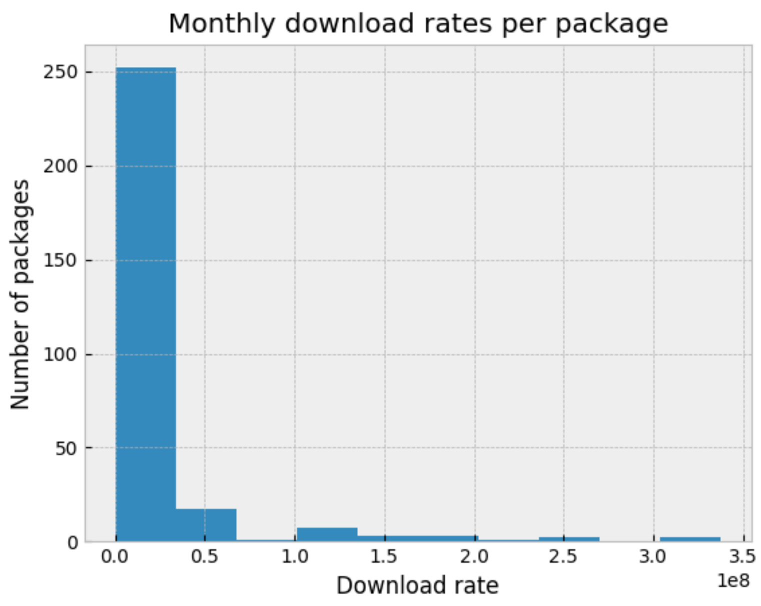

Figure 5.7: Monthly download rates per package, as found by PyPI Stats in the web scanner.

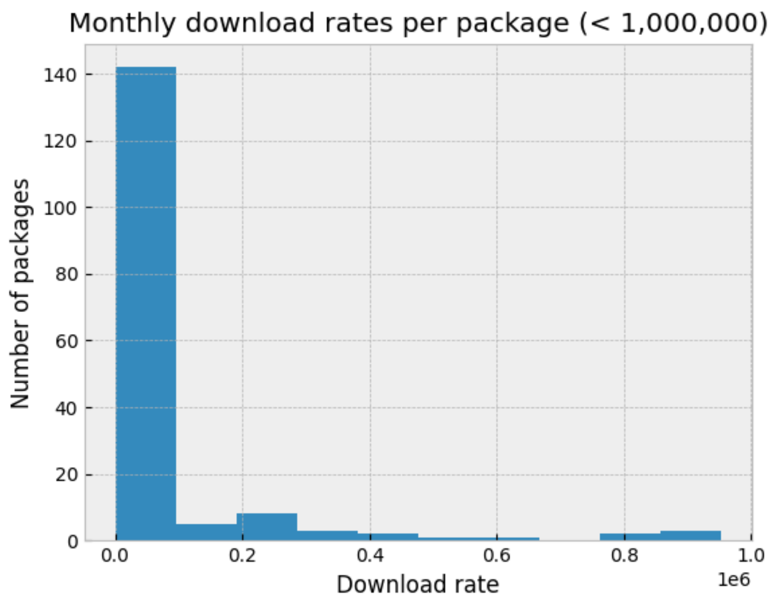

Figure 5.8: Monthly download rates per package, with a filtered dataset. Similarly to the previous figures, outliers are excluded by limiting the dataset to packages with less than 1,000,000 downloads.

6 Conclusion

This last chapter features a discussion of the results in regard to the research questions posed in the introductory chapter. The first one discusses the threat model of PyPI (RQ1), the second one focuses on the mentioned countermeasures (RQ2), and the third one looks at the viability of the scanner program tools (RQ3). Lastly, the chapter features a final general conclusion, and some comments on potential future work in the field.

6.1 Research Question 1

As can be seen in the threat model, there are many possible avenues of attack in the PyPI system. While there are few entry points (PyPI login page, `twine` upload command, contents of uploaded packages), there are many attacks that can be performed with little effort. Furthermore, the mentioned exit point, the error messages of the login prompt, can be used to some degree to determine the existence of some account, making dictionary attacks easier. This design choice is of course meant to be helpful to users who might have forgotten which username is used where, but can have adverse effects.

While the security logging of PyPI is efficient, if an attacker gains access to a package owner account, they are in ultimate control of the package. As mentioned, package deletion is to some extent irreversible, so a version of a denial of service attack is very possible from this position, not to mention uploading vulnerable or malicious releases of the package code.

The typosquatting attack is overall one of the easiest attacks to perform, hence why its DREAD score in the model was the highest. For example, given that most large contemporary packages are open-source, with accessible code, anyone could download the code, make some malicious adjustments, and upload it again under a similar name. An even easier approach is to just add a malicious package as a dependency before the upload, forcing a user to download it along with the typosquatting package. There are no current countermeasures in PyPI against this.

6.2 Research Question 2

Several possible countermeasures and mitigation techniques were mentioned in Section 4.4, which are each variably difficult to implement. The easiest one, as it already exists on the PyPI website, is multi-factor authentication. Having it be mandatory, at least for the powerful package owners, would greatly improve account and package security in general. We also proposed the usage of SSH keys which, similar to GitHub, are used to authenticate the uploads of new releases. This would ensure that uploads can only be done from accepted devices.

Repudiation is one of the smaller issues in PyPI, given their current robust logging system. Package tampering is a more pressing concern, and applying some sort of basic vulnerability scanning could prevent the uploading of some malicious or poorly written package code. It could be possible to describe the scan status of each package on the package's project page, and in that way inform possible users. Similarly, `pip` could present such information prior to the installation of packages.

Regarding the project pages, adding the ability for PyPI users to report potentially harmful packages could decrease the time it takes for malicious packages to be discovered (previously it could take up to 3 years [3]). This ability already exists in both Ruby's Rubygems and JavaScript's NPM.

This community reporting countermeasure could also be used to prevent typosquatting attacks, which are otherwise hard to avoid. There is the option of implementing some sort of naming limitations, something that is relatively difficult to do effectively. Especially when there are over 450,000 packages (and growing), and many are possibly related to the same project or organisation, hence why their names are similar.

6.3 Research Question 3

As for the user's side of things, there are many possible tools at their disposal. Some can be effectively used for information gathering purposes, such as the vulnerability database Safety DB, PyPI Stats, or the GitHub API. Looking at the results in the figures of Chapter 5, it is apparent that most packages have a similar fork/star amount and download rate. As such, it is difficult to make assumptions of legitimacy based on those metrics alone, especially considering the large standard deviations found for each metric, as well as the generally low median values.

However, using these resources to determine the legitimacy of a given package, e.g. to avoid a typosquatting attack, is still a viable option. Furthermore, knowing the previous vulnerabilities of a given package is important information, especially if a package demands the usage of older releases of some required package. The vulnerable release could still exist, and be possible to download. The same idea can be applied to the download rate of a package; if one wants to use a commonplace, contemporary package, but finds that it or one of its dependencies has a low download rate, some closer inspection could be warranted.

The package scanner quite easily found the packages that were of the older, non-wheel format, which is an indicator of potentially malicious `setup.py` files. However, many packages, especially old ones, use the non-wheel format, so detecting irregular `setup.py` files has to be done separately. The Regex method used in the package scanner found nearly all of the non-wheel packages to have non-standard `setup.py` files, which is likely many false positives. The Regex method is quite inaccurate, partly because it is hard to design an effective pattern, and partly because programmers oftentimes do not adhere to the standard set by the documentation, or use some different version as a standard. That is to say, while many packages' `setup.py` files are non-standard, they are not necessarily malicious. Another tool, such as the Python AST module mentioned in Section 2.5, could be a viable option for detecting irregular files.

It is important to note that the program only looked at the first level of dependencies available, and could therefore easily miss a potential supply-chain attack further down the chain of dependencies.

Of course, the fact that a package uses the non-wheel format is still of interest to the prospecting user, and manual, or a different automatic control, could be applied to ensure security. Furthermore, unless the old format is absolutely necessary for some packages' functions, or backward compatibility, PyPI could phase it out in favour of the new wheel format.

Using the Levenshtein distance for detecting similarly named packages was shown to be an efficient method. However, there were some outliers in the results of the scanner program test (as seen in Figure 5.1). These outliers are likely packages with very short names, such as `pip`, as package names of that length are a very short string distance from one another. Once again, it would be up to the user's discretion to recognise if a package name is clearly trying to imitate another, and using a combination of the information gathering tools mentioned could help with that distinction.

Finally, the Docker containerisation solution works well to containerise the testing of the packages. However, Docker, like most virtual machines, is not really meant to be used in a quarantining fashion, thus we cannot assume that there are no ways for a malicious package to infect the outside device. The role of containerisation instead becomes to make it increasingly difficult for a malicious actor to carry out an attack using the package. There is likely some other containerisation software that could have been used instead, but the result would reasonably have been similar. Nonetheless, containerisation adds to the difficulty of the attack by forcing a malicious package to somehow find a way out of the container.

6.4 Conclusion

As has been previously mentioned, security in PyPI-like software ecosystems is no trivial matter, and reaching a totally secure state is practically impossible. Furthermore, PyPI being owned by a non-profit organisation makes it even more difficult to develop a more secure system, as the work has to be done by select community members (PyPA). For a limited group like them, keeping over 450,000 packages in check is no easy task, and it gets increasingly harder as the repository grows.

There are many suggested solutions for PyPI to take, some from this paper, and some from other related works. But regarding PyPI's current state, a security-aware Python developer should be advised to use some sort of scanner program or tool to ensure that the package they want to use (or some of its dependencies) will not perform unwanted actions. Whether they use some of the tools mentioned and tested here, or some of the many other ones (such as the previously mentioned Bandit program), the main point and takeaway is to increase the overall security of our programs.

6.5 Future Work

This paper has focused on the vulnerabilities of the PyPI software repository, but similar methods can be used to analyse other repositories, such as JavaScript's NPM or Ruby's Rubygems. While some of the same ideas can be applied, the threat models would likely be very different, and completely different tools might be considered.

The use of the different metrics, such as the GitHub API statistics or the PyPI Stats download rates, could use some further research, to figure out which is the most interesting to look at. E.g., is there some point in looking at the number of GitHub stars a given package has, in order to decide its quality?

Furthermore, there could be some interest in creating some sort of wrapper for Python's `pip` installer, that uses the containerisation and tools mentioned in this paper. This way, it can grant more information on the package a user is about to download, by analysing it in a contained environment first. Such a program could make it easier for prospective Python developers to make more informed decisions about using some given package.

Bibliography

[1] J. Ruohonen, K. Hjerppe, and K. Rindell, “A large-scale security-oriented static analysis of python packages in pypi,” in *2021 18th International Conference on Privacy, Security and Trust (PST)*, 2021, pp. 1–10. DOI: `10.1109/PST52912.2021.9647791`.

[2] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, “Typosquatting and combosquatting attacks on the python ecosystem,” in *2020 IEEE European Symposium on Security and Privacy Workshops*, 2020, pp. 509–514. DOI: `10.1109/EuroSPW51379.2020.00074`.

[3] M. Alfadel, D. E. Costa, and E. Shihab, “Empirical analysis of security vulnerabilities in python packages,” in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2021, pp. 446–457. DOI: `10.1109/SANER50967.2021.00048`.

[4] A. Bagmar, J. Wedgwood, D. Levin, and J. Purtilo, “I know what you imported last summer: A study of security threats in thepython ecosystem,” *CoRR*, vol. abs/2102.06301, 2021. arXiv: `2102.06301`. [Online]. Available: `https://arxiv.org/abs/2102.06301`.

[5] B. Kaplan and J. Qian, “A survey on common threats in npm and pypi registries,” in *Deployable Machine Learning for Security Defense*, G. Wang, A. Ciptadi, and A. Ahmadzadeh, Eds., Cham: Springer International Publishing, 2021, pp. 132–156, ISBN: 978-3-030-87839-9.

[6] K. Koidan, “Difference between python modules, packages, libraries, and frameworks,” *LearnPython.com*, 2021, (Accessed: 2023-06-18). [Online]. Available: `https://learnpython.com/blog/python-modules-packages-libraries-frameworks/`.

[7] *Pypi statistics*, `https://pypi.org/stats/`, (Accessed: 2023-03-18), 2023.

[8] M. Ohm, H. Plate, A. Sykosch, and M. Meier, “Backstabber’s knife collection: A review of open source software supply chain attacks,” in *Detection of Intrusions and Malware, and Vulnerability Assessment*, C. Maurice, L. Bilge, G. Stringhini, and N. Neves, Eds., Cham: Springer International Publishing, 2020, pp. 23–43, ISBN: 978-3-030-52683-2.

[9] B. Toulas, “10 malicious pypi packages found stealing developer’s credentials,” *BleepingComputer*, 2022, (Accessed: 2023-06-18). [Online]. Available: `https://www.bleepingcomputer.com/news/security/10-malicious-pypi-packages-found-stealing-developers-credentials/`.

[10] C. Cimpanu, “Twelve malicious python libraries found and removed from pypi,” *ZDNET*, 2018, (Accessed: 2023-06-18). [Online]. Available: `https://`

www.zdnet.com/article/twelve-malicious-python-libraries-found-and-removed-from-pypi/.

[11] R. Lakshmanan, "Malicious pypi packages using cloudflare tunnels to sneak through firewalls," *The Hacker News*, 2023, (Accessed: 2023-06-18). [Online]. Available: https://thehackernews.com/2023/01/malicious-pypi-packages-using.html/.

[12] R. Lakshmanan, "Researchers uncover 29 malicious pypi packages targeted developers with w4sp stealer," *The Hacker News*, 2022, (Accessed: 2023-06-18). [Online]. Available: https://thehackernews.com/2022/11/researchers-uncover-29-malicious-pypi.html.

[13] R. Daws, "Pypi package installs cryptominer on linux systems," *Developer Tech*, 2022, (Accessed: 2023-06-18). [Online]. Available: https://www.developer-tech.com/news/2023/feb/15/clipper-malware-found-in-over-451-pypi-packages/.

[14] *Packaging python projects*, https://packaging.python.org/en/latest/tutorials/packaging-projects/, (Accessed: 2023-06-18), 2023.

[15] *Setuptools: Quickstart*, https://setuptools.pypa.io/en/latest/userguide/quickstart.html, (Accessed: 2023-06-18), 2023.

[16] K. H. Kim, K. Kim, and H. K. Kim, "Stride-based threat modeling and dread evaluation for the distributed control system in the oil refinery," *ETRI Journal*, vol. 44, no. 6, pp. 991–1003, 2022. DOI: https://doi.org/10.4218/etrij.2021-0181. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.4218/etrij.2021-0181. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.4218/etrij.2021-0181.

[17] L. Conklin, *Threat modeling process*, https://owasp.org/www-community/Threat_Modeling_Process, (Accessed: 2023-06-18).

[18] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, pp. 32–88, Apr. 2000. DOI: 10.1145/375360.375365.

[19] E. Nam, "Understanding the levenshtein distance equation for beginners," *Medium*, 2019, (Accessed: 2023-06-18). [Online]. Available: https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0.

[20] Y. Gelb, "Automatic execution of code upon package download on python package manager," *Medium*, 2022, (Accessed: 2023-06-18). [Online]. Available: https://medium.com/checkmarx-security/automatic-execution-of-code-upon-package-download-on-python-package-manager-cd6ed9e366a8.

[21] *Writing the setup script*, https://docs.python.org/3/distutils/setupscript.html, (Accessed: 2023-06-18), 2023.

[22] *Safety db*, https://github.com/pyupio/safety-db, (Accessed: 2023-06-18), 2023.

[23] *Pypi download stats*, https://pypistats.org/about, (Accessed: 2023-06-18), 2023.

[24] *What is a container? | docker*, https://www.docker.com/resources/what-container/, (Accessed: 2023-06-18), 2023.

[25] D. K. Konoor, R. Marathu, and P. Reddy, "Secure openstack cloud with bandit," in *2016 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, 2016, pp. 178–181. DOI: `10.1109/CCEM.2016.044`.

[26] *Welcome to bandit*, `https://bandit.readthedocs.io/en/latest/`, (Accessed: 2023-06-18), 2022.

[27] H. van Kemenade, R. Si, and Z. Dollenstein, *Hugovk/top-pypi-packages: Release 2023.04*, version 2023.04, Apr. 2023. DOI: `10.5281/zenodo.7790907`. [Online]. Available: `https://doi.org/10.5281/zenodo.7790907`.

[28] *Python - official image | docker*, `https://hub.docker.com/_/python`, (Accessed: 2023-06-18), 2023.