PySpoof: Detecting and Mitigating Typosquatting Attacks on PyPI

Hao Hu **Samuel Dahlberg**

Abstract

Typosquatting has emerged as a significant threat within the software development ecosystem, particularly in popular package repositories like PyPI. In response to this pervasive issue, we present PySpoof, a robust and efficient detection tool designed to safeguard developers and users from the risks associated with typosquatting attacks in Python repositories.

In our research, we have developed two methods to counteract typosquatting. One uses Damerau-Levenshtein and the other one uses yield generator to generate all kinds of typosquatting cases in the library and then compare them with the given string. One is able to detect 100% of the typosquatting attacks from our limited available data of historical attacks and the other is able to detect 66% while giving accurate type names of typosquatting. We access from this that our solution is effective in mitigating the risks associated with typosquatting.

In our comprehensive evaluation, PySpoof proves to be an effective defense against typosquatting. However, it is important to state our project is still a work in progress. Our aim is to improve the tool that can combine the two methods we have currently and make a fast, accurate tool that can give exact warnings on where the typosquatting attack is.

Contents

1 Introduction

1.1 Problem statement

The problem we aim to address is the potential for typosquatting in package repositories, such as the Python Package Index (PyPI). Package managers, such as npm for Node.js and pip for Python, greatly simplify the process of incorporating third-party dependencies into codebases by automatically resolving and installing the required packages [1].

However, the open nature of these repositories allows any developer to upload packages with names of their choosing. This creates an opportunity for typosquatting, where malicious actors intentionally upload packages with names closely resembling popular packages. The goal is to deceive users into mistakenly importing these typosquatting packages instead of the intended ones.

Typosquatting poses a significant risk as it can lead to unintended consequences, including security vulnerabilities, unexpected behavior, or reliance on outdated or untrusted code. PyPI contains over 220 thousand packages and accrues more than 4 billion monthly package downloads [2]. The impact of typosquatting attempts can be widespread.

Our objective is to mitigate this risk and enhance package dependency security by developing a system that detects potential typosquatting cases. By analyzing package names and comparing them to a curated list of legitimate package names, we can identify suspicious patterns or variations that may indicate typosquatting attempts. This system will provide developers and users with an additional layer of awareness and protection against inadvertently importing malicious or misleading packages.

By addressing this problem, we aim to improve the overall integrity and trustworthiness of package repositories, ensuring that developers can confidently rely on the authenticity of the packages they incorporate into their codebases.

1.2 Goal

The primary objective of this project is to develop a comprehensive tool that can effectively vet imported packages and alert users to potential typos in their package imports. By doing so, we aim to mitigate the risk of users unintentionally importing malicious or misleading packages that closely resemble legitimate ones.

The tool's purpose is to provide users with warnings and notifications whenever it detects possible typos in the imported packages. By implementing this functionality, we aim to enhance the overall security and integrity of the package management process.

By generating timely warnings, users can be alerted to the presence of suspicious or similar package names, allowing them to exercise caution and verify the authenticity of the packages they are importing. This proactive approach helps to minimize the likelihood of accidental importation of packages that may have detrimental effects on the codebase or compromise its security.

Overall, the project's goal is to empower users with a robust tool that detects and prevents the importation of potentially malicious or misleading packages, thereby ensuring a higher level of confidence and reliability in the package management workflow.

2 Background

2.1 Package Repositories

Repositories are where software development teams that develop languages share their code artifacts such as libraries and packages [3]. There are hundreds of thousands (PyPI, RubyGems) to millions (npm) of publicly available packages [4]. It is hard for developers to manually vet every package that integrates into their project because of the large amount of code existing in dependencies. This can easily cause Typosquatting.

2.2 Factors on Typosquatting

Any user can upload a package to repositories due to its open-resource nature, which grants equal trust to all packages regardless of their region. Users also find it difficult to vet the behavior of packages. The large amount of distributed packages can lead to a big amount of typosquatting attacks being launched [4].

2.3 Consequence of Typosquatting

Knowing the consequences of typosquatting can help individuals and businesses avoid the negative impact of this type of cyber attack. Typosquatting may result in a variety of serious consequences, including:

Security breaches: By enabling attackers to insert malware, spyware, or other malicious code into systems or networks, typosquatting can result in security breaches.

Data theft: Typosquatting can also lead to data theft by allowing attackers to gain access to sensitive files or personally identifiable information.

Financial loss: Users can be redirected to fake websites by typosquatting, potentially leading to financial loss if they are tricked into providing sensitive financial information or conducting fraudulent transactions.

Legal issues: Infringement of the trademark or intellectual property rights of others by the typosquatter can result in legal issues associated with typosquatting.

2.4 Types of Typos

Common typos that could be leveraged to orchestrate these attacks included the transposition of consecutive characters, repeated characters, and omitted characters [5].

To prevent typosquatting, we chart out what kind of error types are commonly used for an attack:

- **Repeated Characters**: A random word character is repeated in a package name. This case accounts for potential errors where a word character is mistakenly repeated in a package name. By checking for repeated characters, we can identify and correct typographical errors that could lead to incorrect package names. For example, `boto3` can be mistakenly typed as `botoo3`.
- **Omitted Characters**: The omission of a single letter in a package name. Omitting a single letter in a package name can result in an entirely different package. By checking for omitted characters, we can catch mistakes where a user might accidentally leave out a letter in a package name. Correcting such errors is crucial to ensure the intended package is correctly identified and used. For example, `boto3` can be mistakenly typed as `bto3`.
- **Swapped Characters** : The transposition of two consecutive characters. Swapping two consecutive characters in a package name can create a different word altogether. By checking for swapped

characters, we can detect errors caused by unintentional transpositions. Identifying and rectifying these errors is essential to avoid potential conflicts or confusion when working with packages. For example, `boto3` can be mistakenly typed as `boot3`.

- **Swapped Words**: The transposition of two words in a package name. Swapping the order of two words in a package name can lead to errors, especially in cases where the order of the words carries significance. By checking for swapped words, we can prevent misunderstandings and ensure that the intended package is being referred to accurately. This is particularly important when dealing with packages that follow specific naming conventions or rely on a particular word order. For example, `python-levenshtein` can be mistakenly typed as `levenshtein-python`.
- **Common Typos**: Character substitution based on physical locality on a QWERTY keyboard. This case involves character substitution errors based on physical proximity on a QWERTY keyboard. Many typing errors occur due to the proximity of keys, and checking for common typos helps catch such mistakes. For example, `boto3` can be mistakenly typed as `boro3`.
- **Version Numbers**: The presence of integers located at the end of the package name. : Checking for version numbers is crucial for package management. By verifying the presence and accuracy of version numbers, we can avoid installing the wrong package version. Mistaking a package with a different version number can lead to compatibility issues, missing bug fixes or features, or potentially using a vulnerable version. For example, `urllib3==1.25.10` can be mistakenly typed as or be confused for `urllib3==1.25.11`.

By considering all of these cases during package name checks, we can minimize the risk of errors, improve the reliability of package management, and enhance the overall user experience.

2.5 Levenshtein Distance

In this part, we are going to introduce the main algorithm used in our tool which is the Levenshtein Distance algorithm. Levenshtein distance is a metric used to quantify the difference between two strings of characters. This distance, also referred to as the edit distance, is a metric that quantifies the minimum number of single-character modifications needed to transform one string into another. These single-character modifications may consist of inserting, deleting, or substituting a character in the string [6].

The advantage of using Levenshtein distance is that it is a simple and intuitive metric that can be easily calculated. However, it has some drawbacks. It does not take into account the meaning or context of the strings, only their character-level similarity. It also scores transpositions as a high Levenshtein distance which can give deceiving results when dealing with typosquatting cases that rely on swapped words.

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

Figure 1: Definition of Levenshtein

2.6 Damerau-Levenshtein Distance

The Damerau-Levenshtein distance improves upon the Levenshtein distance by introducing an additional operation: the transposition of two adjacent characters. This operation enhances the accuracy of measuring the similarity or difference between two strings.

Including transpositions in the calculation allows the Damerau-Levenshtein distance to account for common errors or variations resulting from typographical mistakes, such as the swapping of adjacent letters. This feature proves particularly valuable in scenarios where typos or misspellings are prevalent, such as text correction, spell-checking, or identifying similarities between words.

To summarize, the Damerau-Levenshtein distance extends the Levenshtein distance by incorporating the transposition operation, providing a more comprehensive evaluation of string similarity [7].

$$
f_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0 \\ f_{a,b}(i-1, j) + 1 & \text{if } i > 0 \\ f_{a,b}(i-1, j) + 1 & \text{if } j > 0 \\ f_{a,b}(i-1, j-1) + 1 \ (a_i \text{ and } b_j \text{ are not equal}) & \text{if } i,j > 0 \\ f_{a,b}(i-2, j-2) + 1 & \text{if } i,j > 1 \text{ and } a_{i-1}=b_j \text{ and } a_i = b_{j-1} \end{cases}
$$

Figure 2: Definition of Damerau-Levenshtein

3 Method and Implementation

3.1 Data Collecting

To gather the necessary data, we established a monthly process to obtain a comprehensive dump of the top 5,000 most-downloaded packages from PyPI (Python Package Index). This process involves leveraging the invaluable resource available at "https://hugovk.github.io/top-pypi-packages/top-pypi-packages-30-days.min.json," a dedicated website specifically designed to provide access to the latest statistics. By accessing this website, we acquired a meticulously prepared JSON file containing vital information such as the package names and their corresponding download times for their dependencies.

To ensure compatibility and ease of analysis, we seamlessly transformed this JSON file into a user-friendly and widely compatible CSV format, facilitating further exploration and utilization of the acquired data.

3.2 Typosquatting Guard

Within our project, we have diligently implemented two distinct methods, each serving a specific purpose, to seamlessly handle the installation of PyPI files and cater to the diverse needs of our users.

The first method caters to users who prefer the convenience and flexibility of installing PyPI files directly through the command line interface (CLI). By leveraging this method, users can effortlessly install the required packages by executing the necessary commands in their preferred CLI environment. This approach ensures a streamlined installation process and enables users to swiftly incorporate the desired PyPI packages into their projects without any unnecessary complications.

In addition to the command line installation method, we have also developed a second method specifically tailored for users who make use of the ubiquitous "requirement.txt" file when managing their project dependencies. This method takes into account the widely adopted practice of listing project dependencies within a "requirement.txt" file, allowing users to clearly define the necessary packages and their respective versions. By integrating this method into our project, users can conveniently specify their desired package dependencies within the "requirement.txt" file and effortlessly ensure the consistent installation of the required packages.

These two distinct methods enable us to cater to a broad spectrum of user preferences and ensure a seamless installation experience regardless of the user's chosen approach. Whether users opt for the command line installation or prefer the simplicity of managing dependencies through a "requirement.txt" file, our project diligently accommodates their needs, facilitating a smooth and efficient integration of PyPI packages into their projects.

3.2.1 Command Line Check

To prevent typosquatting from the command line there is a need to understand the workflow of the installation of a command installed by a user. First, the user enters a command to trigger the package's installation, e.g. *pip install boto3* (step 1). Then the dependency tree of that package is built by the package manager (step 2), discarding from the tree the packages that are already installed (step 3). At this point, the workflow triggers the command line check we've implemented and it will check if any typosquatting is detected from these packages.

First, we check the command user enters in `install_guard.py` about whether it is a valid input. If it is a valid input, then we go check the input with the package names we have in our library by using the functions in `fast_typoguard.py`

The pseudo-code of the algorithm in `fast_typoguard.py` is as below:

```
Input: package i to be installed
Input: dependency tree G
if check_typo(i) == None then
  for g in G:
    if check_typo(g) == None then
      run_installation
    else: check_continue_installation(g)
else: check_continue_installation(i)
```

In our function `check_typo`, we wrote a simple method that checks for typos:

```
Input: string S to be checked
Input: dependency name library W
if S in W:
  return (None,' No typo found')
else:
  for w in W:
  if is_substing:
    return(word,'Substring)
    else if swapped_words:
      return (word, 'Swapped words')
    else: if Levenshtein_distance < 3:
      return (word, 'Levenstein distance')
    return (word, 'Not in the package list')
```

Here swapped_words and substring will look for the cases that the Levenshtein distance might not catch as the strings would require a lot of edits to achieve and if they are not triggered we will then check the Levenshtein score. After finishing the program will warn via the terminal if there were any possible typos and ask the user to confirm if they want to continue installing said package or not.

For example, for the command `python3 check_typo.py pip install boto2`, if `boto2` exists and is available for download, since this is a similar name to the popular package boto3, we are expected to get:

```
Do you want to continue with the installation? (y/n): n
Installation canceled.
```

By examining the package name in the installation command, we can verify if it matches the actual name we expected. This helps ensure that the user is installing the intended package and not a similarly named package that could be a typosquatting attempt. For example, if the user intended to install `requests` but mistakenly types `reqeusts`, we can detect the difference and prompt for confirmation.

3.2.2 Requirements Check

Checking the `requirements.txt` file can help in detecting potential instances of typosquatting by comparing the package names and versions listed in the file with the intended dependencies.

It can be helpful with using package name verification: By regularly reviewing and auditing the package names listed in the requirements.txt file, we can ensure that the names align with the actual packages you intend to use. This verification step helps catch any potential typos or variations in package names that might be used for typosquatting purposes.

For the requirement list check we implemented a different method as we wanted to display more info in regards to the typo that was found.

First, we initiate the process by scanning the specified folder input, diligently searching for the presence of the `requirements.txt` file within the project directory.

The program will then compare every element in `requirements.txt` with our dependency name library to check if there is a possible match for typosquatting.

```
def scan_files(file_path):
    with open(file_path, 'r') as file:
        packages = [Requirement(line.strip()).name for line in file.readlines()]
    start_time = time.time()
    print(packages)
    print(f"Time used parsing: {time.time() - start_time} seconds")
    total_typo_list = []
    for word in packages:
        typo_list = compareString(word)
        total_typo_list.append(typo_list)
    endtime = time.time()
    print(f"Time used: {endtime - start_time} seconds")
    return total_typo_list
```

In the `compare_string` function, we first find the words in the library that has less than 3 length differences with the given string to minimize the number of strings to check. We then generate all possible typos of the words in the library and check if the string is among them. The string might be the typo of some words in the library, we return all the possible words of that. For example, for string `return` we return `[fire, rx,rq, rope, rel]`. If we get an empty list we say that the typo is not found.

```
def compare_string(string):
    suitable_matches = []
    lines = get_list()

    for word in lines:
        if abs(len(word) - len(string)) < 3:
            suitable_matches.append(word)

    match_list = []
    for word in suitable_matches:
        match_found = False
        if same_import(string, suitable_matches):
            continue
        else:
            for typo, desc in generate_typos_combinations(word):
                if string == typo:
                    match_found = True
                    break
            if match_found:
                match_list.append((word, desc))
```

Then we generated a variety of potential typos for a given word. It employs several techniques to create a comprehensive set of typo variations. Here is an explanation of each step:

Swapping Letters: It systematically swaps adjacent letters in the word, producing new typos by rearranging their positions.

Repeated Letter: This step introduces a repeated instance of each character in the word, resulting in typos where a specific letter is duplicated.

Deleting a Letter: It generates typos by omitting each letter in the word, producing variants where a single character is removed.

Common Typos: The function checks if any letters in the word have known common typos associated with them. For each such letter, it substitutes it with alternative characters to generate typo variations.

Swapping Words: If the word contains hyphens, this step swaps the order of adjacent words separated by hyphens. It generates typos by rearranging the position of word components.

In our implementation, we intentionally excluded version-number typos from the generated set of typos. This decision was based on our observation that version-number typos are typically accompanied by other code changes and are unlikely to occur as standalone typos. This limitation arises due to the PyPI safe policy, which ensures that package names cannot be altered while keeping only the version number different.

For instance, consider the scenario where an attacker attempts to perform typosquatting by uploading a package named `beautifulsoup4==14.1.1` as a substitute for the legitimate package `beautifulsoup4==13.1.1`. This type of typosquatting is not possible because the package name itself cannot be altered without triggering other code changes. PyPI's safety measures ensure that the integrity of package names is maintained.

In our implementation, we utilized the yield keyword to create generator functions for generating typos and typo combinations. The use of generator functions offers several advantages in terms of memory efficiency and code readability, including memory efficiency that allows us to generate typos and combinations on the fly without storing all of them in memory at once. And also it is lazy so that we can avoid unnecessary computation.

```
def generate_typos_combinations(word):
    typos = generate_typos(word)

    for typo, error in typos:
        yield typo, error
        for typo2, error2 in generate_typos(typo):
            combined_error = error + ' + ' + error2
            yield typo2, combined_error
```

If we were to have a `requirements.txt` file that looks like follows:

```
beautifulsoup4==4.9.1
levenshtei-python==0.12.0
```

The result of running the program would then be:

```
No type error found for beautifulsoup4
Input levenshtei-python may refer to another package:
Deleted letter and Swapped words detected, you might be looking for python-levenshtein
```

To make the results more visible, we print the packages where typosquatting is present in red. If there are no errors then "All packages are correct" is shown in green.

3.3 Method for Function Checking

A hypothesis test is used here for checking the performance and accuracy of our function. It helps us check whether our function is correct for every kind of typo. The function `check_typo` for checking the command line check passed all the tests within 0.36 seconds. The function `compare_string` passed the hypothesis test in 4 seconds with an accuracy of 100%. We take hypothesis test for `compare_sting` as an example.

In the file `test_hypothesis.py`, we check the accuracy with the library `Hypothesis`. Property-based testing is used in this test. The progress is that, first for all data in the CSV file, we generate typos with the function `generate_typo()` and then give a "maybe right list" for this typo by using the function `compare_string()`. Then if the original data is inside the list generated, we assert that there is no error.

```
@settings(deadline=5000)
@given(st.just("resources/test.csv"))
def test_typo_checker(csv_file):
    with open(csv_file, 'r', newline='') as file:
        reader = csv.reader(file)
        rows = list(reader)
    for row in rows:
        word = row[0]
        typo_list = compareString(generate_typo(word))
        assert word in typo_list
```

We used the same way of generating Python in the typoGuard.

```
def generate_typo(word):
    if "-" not in word:
        typo_type = random.choice([omitted_letter_typo, repeated_letter_typo,
swapped_letter_typo, common_typos_typo])
    else:
        typo_type = random.choice([omitted_letter_typo, repeated_letter_typo,
swapped_letter_typo, common_typos_typo, swapped_words_typo])
    return typo_type(word)
```

To make function `compare_string` able to do the hypothesis test, we slightly modified the return to a list.

```
def compare_string(string):
    suitableMatches = []
    lines = get_list()
    for word in lines:
        if abs(len(word) - len(string)) < 3:
            suitableMatches.append(word)
    match_list = []
    typo_list = []  # List to store the typo-string pairs
    for word in  suitableMatches:
        match_found = False
        if same_import(string, lines):
            continue
```

```
        else:
            for typo, desc in generate_typos_combinations(word):
                if string == typo:
                    match_found = True
                    break
            if match_found:
                match_list.append((word,desc))
                typo_list.append(word)  # Append the typo-string pair to the list
    if match_list != []:
        output = f"Input {string} may have type error:\n"
        for typo,desc in match_list:
            output += f"{desc} it might should be {typo}\n"
        print(output)
    else:
        print(f"No type error found for {string}")
    return typo_list
```

The usage of the hypothesis test here makes us ensure that the typo checker correctly identifies potential typos for different words. It helped to build the robustness of our type checker.

3.4 Instruction on How to Run the code

Below are brief instructions on how to run the project both for the single command line and for checking a requirement file

3.4.1 Command Line Typo Check

First, navigate to the directory where the project is at. For example, `cd TDA602-project-2023`.

Then, enter the command `python3 install_guard.py pip install dependency_you_want_to install`.

The dependency itself and all the other dependencies on its dependency tree will be checked. If they are correct then they will be installed. Else if they cannot be found, warnings will be printed on the command line. Otherwise, if it is identified as a typosquatting then users will be asked whether they want to install the package or not.

3.4.2 Requirement.txt Typo Check

To run entire requirement lists you would instead run the command `python3 requirement_guard.py requirement_directory` where requirement_directory would be the directory where the requirement.txt file is located. in our example we have a directory named example in the same directory as our `requirement_guard.py` file, we then just need to enter `python3 requirement_guard.py example` and it will execute a typo check for all packages listed in the requirement file. This can take a bit of time to run.

3.5 Instruction on Testing the Accuracy

To test the accuracy of the command line typo check, we can run `python3 check_fast.py`.

To test the accuracy of the requirements typo check, we can run `python3 check_full.py`.

4 Result

The following is the result obtained from testing our implementation on both typos that were generated by us and on historical typosquatting cases that has been found on PyPI.

Party for our testing we ran with generated typos to see our program's ability to catch our handmade package names. Then for our main test to determine how accurate our implementation is we used a list of packages that consist of package names that have been detected as attacks previously, this list of historical attacks can be seen in Figure 4 below.

typosquatting_perpetrator	typosquatting_target	repository
python3-dateutil	python-dateutil	pypi
jeilyfish	jellyfish	pypi
smplejson	simplejson	pypi
djago	django	pypi
diango	django	pypi
dajngo	django	pypi
colourama	colorama	pypi
djanga	django	pypi
acqusition	acquisition	pypi
apidev-coop	apidev-coop_cms	pypi
bzip	bz2file	pypi
crypt	crypto	pypi
django-server	django-server-guardian-api	pypi
pwd	pwdhash	pypi
setup-tools	setuptools	pypi
telnet	telnetsrvlib	pypi
urlib3	urllib3	pypi
urllib	urllib3	pypi

Figure 3: History attack on PyPI

These tests were then performed on both the command line check and the requirement list check.

4.1 Result of Command-line PySpoof

During our exploration and testing, we encountered a significant challenge in finding an available typosquatting package on PyPI. Since the PyPI ecosystem has implemented robust security measures against typosquatting it is difficult to find current typosquatting cases active.

As a result, when attempting to generate random typos for testing purposes, we consistently received a request code of 404, indicating that the requested package was not found. This outcome is expected, as the PyPI platform actively safeguards against the presence of typosquatting packages, ensuring the integrity and security of the Python package ecosystem.

Although we were unable to directly test the response of Spelling Error Guard to spelling attempts, we were able to evaluate its effectiveness in detecting and flagging potential spelling errors based on predefined spelling error patterns and common typographical errors and generate typo warnings even though the package can not be found.

```
hh@hh:~/TDA602-project-2023$ python3 install_guard.py pip install boto2
Metadata request for boto2 returned status code 404
The package 'boto2' might have a typo.
```

4.1.1 Accuracy of Command-line PySpoof

When testing on our limited sample of historical typosquatting packages we managed to achieve an accuracy of 100%. All the tests passed the check for the command-line PySpoof. This outcome indicates that the command-line PySpoof performed admirably in identifying and highlighting potential typos in the command-line inputs. It correctly flagged instances where a user might have inadvertently mistyped a package name or made a typographical error while specifying the package dependencies.

4.2 Result of requirements.txt PySpoof

As part of our testing and evaluation, we chose a random project we found called "pypi-scan" from the GitHub repository available at https://github.com/IQTLabs/pypi-scan. Below is the initial `requirements.txt`: We then generated some random typos to substitute some of the packages in it. Shown below is the requirement file and on the left side are the typos from the initial package name and the ones with an arrow indicates a changed package.

```
beautifulsoup4==4.9.1 -> beautifilsoup4==4.9.1
cwtiifi==2020.6.20
chardet==3.0.4
idna==2.10 -> idnaa==2.10
jellyfish==0.8.2
jsotreee==0.5.1
mrs-spellings==1.0.3
python-levenshtein==0.13.0 -> levenshtein-pythonn==0.12.0
requests==2.24.0
soupsieve==2.0.1
termcolor==1.1.0
urllib3==1.25.10
```

Below is the output of our program after running the requirement guard on the project.

```
hh@hh:~/TDA602-project-2023$ python3 request_guard.py example
Input beautifilsoup4 may have a type error:
Common typos + Common typos, it might should be beautifulsoup4

Input idnaa may have a type error:
Repeated letter, it might should be idna

Input levenshtein-pythonn may have a type error:
Swapped words + Repeated letter, it might should be python-levenshtein

Please check the spelling of these packages
beautifilsoup4
idnaa
levenshtein-pythonn
```

4.2.1 Accuracy of requirements.txt PySpoof

When running the requirement tests for the historical attack package names we managed to detect 12 out of the 18 packages which can be seen in the figure below, giving us a success rate of 66%.

This is less than we hoped for but also expected as the requirement check method as of now has fewer cases of typos implemented.

One example would be that it does not check for substrings of packages so typosquatting packages that have a lot of words appended to them like django-server-guardian-api instead of django-server are not caught.

Package name	Detected
python-dateutil	no
jellyfish	yes
simplejson	yes
django	yes
django	yes
django	yes
colorama	yes
django	yes
acquisition	yes
apidev-coop_cms	no
bz2file	no
crypto	yes
django-server-guardian-api	no
pwdhash	yes
setuptools	no
telnetsrvlib	no
urllib3	yes
urllib3	yes

Figure 4: Detect of PySpoof on the historical attack for method on requirements.txt

While a higher level of accuracy may be required, it would also require additional computing resources and considerable time to accurately determine whether the provided package name indicates the presence of a domain spoof. It also generates a large number of warnings, which can be overwhelming and annoying to users.

5 Discussion

5.1 Data collection

The dataset we use is the top monthly 5,000 downloaded PyPI Packages from https://hugovk.github.io/top-pypi-packages/. Initially, we used `beautifulsoup4` to scrape data from PyPI.

```
n_threads = 32
output_filename = 'resources/pypi_download_counts_new.csv'
lock = threading.Lock()

response = requests.get('https://pypi.org/simple/')
soup = BeautifulSoup(response.content, 'lxml')

log_file = open(output_filename, 'a')
log_file.write('package_name,weekly_downloads\n')
def log(m):
    lock.acquire()
    log_file.write('{}\n'.format(m))
    lock.release()
```

The approach using beautifulsoup4 to download the dependencies was very time-consuming and we ended up not being able to get all the dependencies we wanted and many of them are not useful due to low popularity. Therefore we changed our approach and used the top 5000 PyPI results instead. This results in a limited amount of libraries. We initially had the function to ask users to add a dependency library to the white list to account for missing packages which can be seen below. Further work needs to be done by adding more packages to the data library in a reliable way.

```
def add_to_whitelist(package_name):
    with open('resources/data.csv', 'a', newline='') as csv_file:
        writer = csv.writer(csv_file)
        writer.writerow([package_name])
```

5.2 Comparison of the Two Methods

In the previous section, we wrote two methods to compare the string. This is due to the fact that there are two different possible ways of being attacked by typosquatting. One type of attack is when the user installs from the command line, and the other is when running `pip install -r requirements.txt`. The former needs to identify the typo quickly, while the latter needs to report where the possible errors are due to the sheer volume of packages.

To make it more clear, we compared the two methods and gave the advantages and disadvantages of them.

	Advantage	**Disadvantage**
Command Line	Efficient check	Sensitivity to Levenshtein Typo location Excess warning
Requirements.txt	Easy to find typo location	Excess running time

For the method to identify typosquatting in the command line,

Advantage :

- **Efficient check**: It can check typosquatting in a short time and had a high accuracy rate of 99.44% in historical attacks

Disadvantages :

- **Sensitivity to Levenshtein distance threshold**: the method uses a fixed threshold of 3 for the Levenshtein distance comparison. This threshold may need to be adjusted based on the specific use case and desired level of tolerance for typos.
- **Typo location**: Without warning what is exactly the typo, users may have difficulty discovering the location of the typo
- **Excess warning**: The program may generate too many warnings and the user may feel annoyed by them and ignore them.

For the method used in requirements.txt,

Advantage :

- **Easy to find typo locations**: We showed the typo categories and possible right answers in the command line. This can help users easily find where the typo is.

Disadvantages :

- **Excessive running time**： When the required package is not in the library, it takes too long to list all the possible typos before we can conclude that the file is not in the library. This makes the performance of this method degrade.
- **Specific formatting**: As of now the requirement.txt file must follow a specific formatting style for the method to function as intended, specifically it has to follow the format of "<package>==<version>" only, while a requirement file for pip has additional functionality that can be specified.

Based on the comparison of the two methods for identifying typosquatting attacks in the command line and requirements.txt, it is difficult to definitively determine which one is better. Each method has its own advantages and disadvantages, catering to different use cases and requirements.

5.3 Future Work

There is still much to improve in our PySpoof tool. To improve PySpoofs for typosquatting in the future, we can consider the following strategies:

More effective Algorithm: According to the discussion above, we have two methods to check typosquatting. One is fast but cannot give exact warnings, the other one can give warnings but not enough accuracy. The future work of these PySpoof tools can be finding a good way or creating a better algorithm for combining them and making a quick, accurate tool that can give detailed warnings. Another improvement could be by incorporating popularity score as a metric as implemented by Taylor et al. [4], a popularity score is used in their algorithm as a metric to determine typosquatting cases, this would be interesting to delve into as well and incorporate into our two methods for future iterations.

More dataset: We also did not get enough data sets for all kinds of typosquatting. If we have enough time, it would be good to expand our library with the data-collecting method. We can also establish a system where users can report suspicious domain name impersonation attempts. Collecting and analyzing user-reported data helps identify new domain impersonation patterns and stay updated with

emerging techniques. Encourage users to report suspicious package names or potential domain impersonation incidents to continually enhance the PySpoof system.

Machine Learning: We can also leverage machine learning techniques, such as natural language processing (NLP), deep learning, or neural networks, to train models that can effectively distinguish between potential typosquatting patterns and legitimate package names. These models can be developed to detect suspicious package names by analyzing patterns and employing advanced linguistic analysis [8].

Adapt to Other Languages: Expanding the functionality of PySpoof to programming languages other than Python, such as Java or Haskell, would greatly enhance its practicality and broaden its scope. We can extend PySpoof's capabilities beyond Python to include support for other programming languages. By fetching packages from different languages, we can store them in our library for analysis and detection purposes.

By adopting these suggested improvements, we can optimize the code to improve the efficiency and accuracy of the string comparison process. This optimization will help us to identify potential typos in package names more reliably.

6 Contribution

The project was equally distributed and the whole document has been written and edited by both members.

References

[1] “Pypi download stats.” [Online]. Available: https://pypistats.org/top

[2] Stats, P. (2020). Analytics for pypi packages. https://pypistats.org/.

[3] CloudRipo,“Python repositories¶,” 2023. [Online]. Available: https://www.cloudrepo.io/docs/python-repositories.html#:~:text=Python%20repositories%20are%20where%20software,can%20be%20uploaded%20and%20downloaded

[4] M. Taylor, R. Vaidya, D. Davidson, L. De Carli, and V. Rastogi, “Defending against package typosquatting,” in Netw. System Secur., Cham, 2020, pp. 112–131.

[5] Wang, Y.-M., Beck, D., Wang, J., Verbowski, C., & Daniels, B. (2006). Strider typo-patrol: Discovery and analysis of systematic typo-squatting. SRUTI, 6(31-36), 2–2

[6] L. Yujian and L. Bo, “A Normalized Levenshtein Distance Metric,” in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 29, no. 6, pp. 1091-1095, June 2007, doi: 10.1109/TPAMI.2007.1078.

[7] F. J. Damerau, “A technique for computer detection and correction of spelling errors,” vol. 7, New York, NY, USA: Association for Computing Machinery, 1964, p. 171. [Online]. Available: https://doi.org/10.1145/363958.363994

[8] Y. Huang, Y. L. Murphey and Y. Ge, “Automotive diagnosis typo correction using domain knowledge and machine learning,” 2013 IEEE Symposium on Computational Intelligence and Data Mining (CIDM), Singapore, 2013, pp. 267-274, doi: 10.1109/CIDM.2013.6597246.