1. ABSTRACT

Software vulnerabilities refer to flaws or bugs in a software system that attackers can exploit to compromise the integrity, confidentiality, or availability of the software or the data it handles. These vulnerabilities can exist at various levels of a software stack, including application software, operating systems, libraries, and firmware. Therefore, detecting and fixing software vulnerabilities becomes critical in software maintenance. This project aims to build a tool that can crawl through various sources on the internet to compile a high-quality dataset for benchmarking and automating vulnerability detection and repair solutions.

In the scope of this project, we chose **snyk** to be our designated source for vulnerabilities. The Snyk Vulnerability Database is a comprehensive list of known security issues. It provides key security information used by Snyk products to find and fix code vulnerabilities.

2. APPROACH

We first introduce a way to crawl data from our designated source, followed by cleaning and standardizing data for future uses; Lastly, obtained data is reviewed and used to gain insights about trends in software vulnerabilities.

2.1. Data Crawling

Our crawler uses Python Scrapy to crawl through snyk to compile a high-quality dataset for benchmarking and automating vulnerability detection and repair solutions. Scrapy is an open-source and collaborative web crawling framework that allows developers to write spiders to scrape data from websites.

Using scrapy, we can get all of our necessary information of a vulnerability provided by snyk, along with an external link to a git commit for obtaining source code.

2.2. Data Standardization

Responses received using scrapy will be preprocessed and reformatted into a more ‘suitable’ form for dataset (i.e. json format). Any irrelevant or duplicated attribute is cleaned in this process.

2.3. Source Code Handling

If there is a github commit patch available for the vulnerability, code changes history will also be included in the data. Obtaining code changes of a snyk vulnerability allows you to see the exact code lines that introduced or fixed a vulnerability, and it can help understand the root cause and impact of the vulnerability in code context, ultimately improving code security and quality.

For this problem, we will utilize GitHub API, a set of web services which allows us to interact with GitHub programmatically and acquire code changes and time of specific commits as needed.

2.4. Implementation

For further information about our works, acquired data, documentation about dataset and source code, please visit our Github repository Metanyu/vuln-crawler

3. RESULTS & ANALYSES

3.1. Dataset

The dataset we obtained for this project contains **1707** snyk **application** vulnerabilities, including **888** entries with patched or source code from the most recent 2 months.

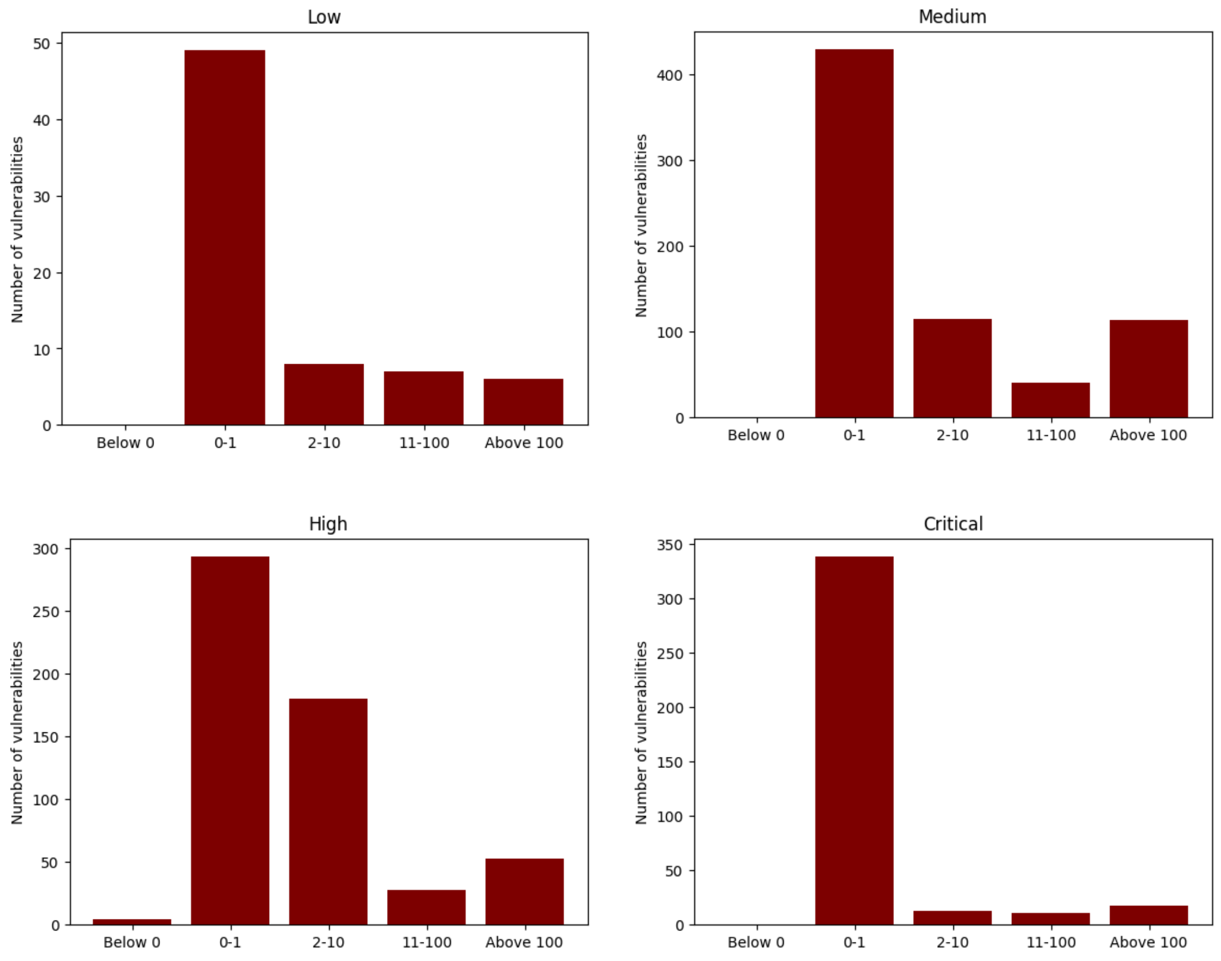
3.2. Analyses questions

We are interested in how the *severity* of a vulnerability would affect actions of maintainers and snyk team, and what are the most common trends in discovered vulnerabilities in late 2023 - start of 2024. This includes but not limited too:

- The amount of time snyk team took to publish an entry,
- Distribution of severity,
- Some of the most commonly seen issues in a package,
- Some package managers & language may hold more vulnerabilities than other,
- And remediation status for the vulnerabilities.

3.3. Result

3.3.1. Publication and disclosure time gap (days) based on severity



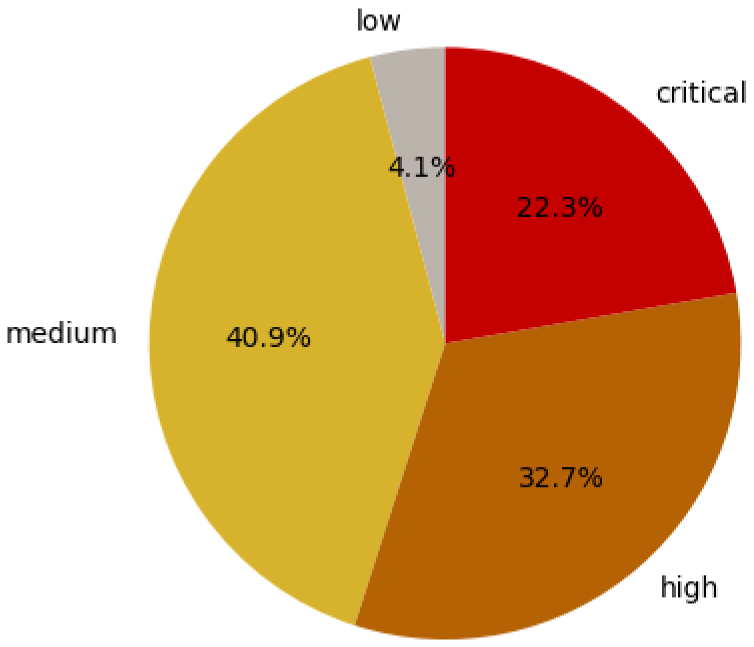
Generally, Snyk tries to minimize the gap between the publicationTime and the disclosureTime, with most of the issues being checked and published on the same day it is disclosed, by continuously monitoring various sources of vulnerability information, such as the NVD, GitHub, and npm.

From the graph, we could infer that the time gap for critical vulnerabilities appears to be slightly smaller when compared to other severity, with most of the issues being published on the same day as it is disclosed, which helped users to avoid critically harmful packages in a timely manner.

3.3.2. The severity distribution

In last 2 months, the number of vuln occurs depending on each type of severity as follow:

- Low: 70
- Medium: 699
- High: 558
- Critical: 380



In general, the majority of vulnerabilities published fall into medium severity, followed by high, critical and lastly low. This suggests that most of the vulnerabilities pose a significant risk if left unattended.

3.3.3. Most common issues type:

Type	Distribution within all issues (%)
Malicious Package	15.70
Cross-site Scripting (XSS)	5.45
Improper Access Control	3.57
Information Exposure	3.51
Use After Free	3.34
Uncontrolled Resource Consumption ('Resource Exhaustion')	2.87
Denial of Service (DoS)	2.69
Cross-Site Request Forgery (CSRF)	2.64
Improper Input Validation	2.52
Integer Overflow or Wraparound	2.46

According to this, **Malicious Package** accounts for the largest portion of all vulnerabilities at **15.70%**. While most commonly used packages are safe to use, packages with malicious intent remain a significant concern. Therefore, it is good best practice to choose your dependencies wisely and do research before adding a dependency to your project.

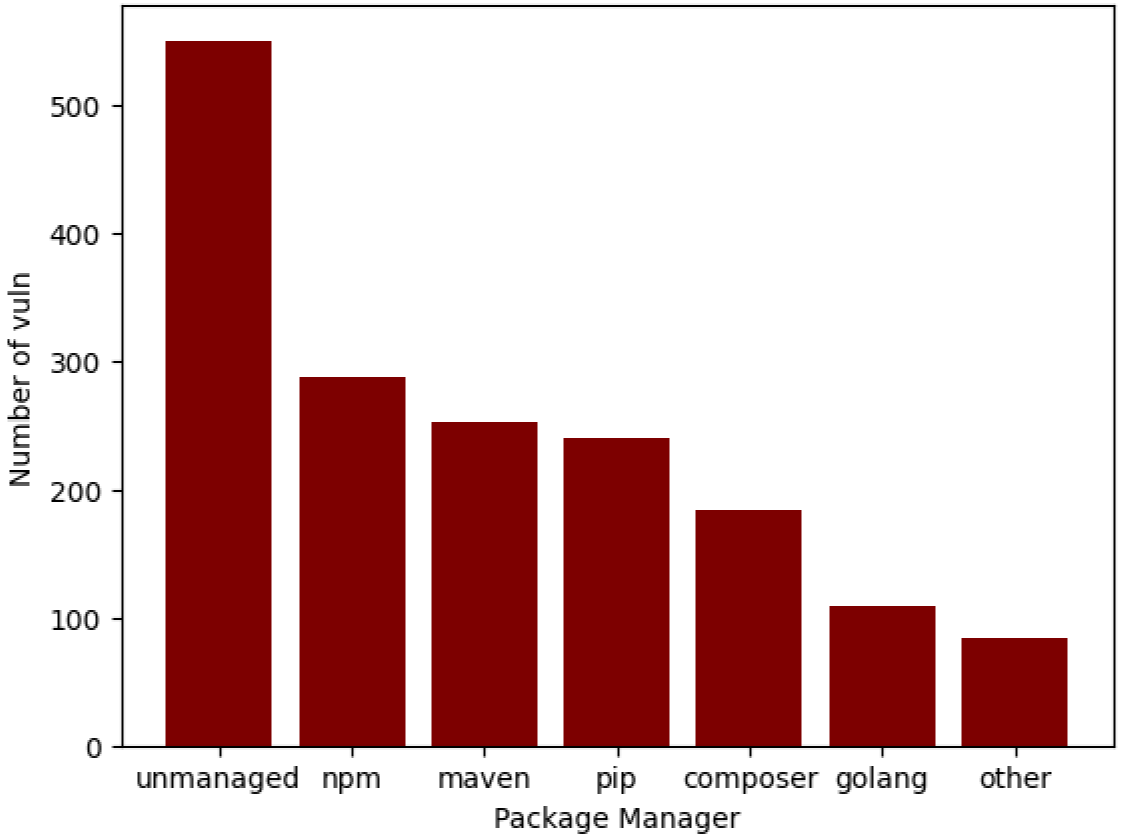
Generally, users can review the top common security issues found in application code listed above and pay extra attention to assess these risks and develop effective coding strategies.

3.3.4. Comparison on package manager

Number of vulnerabilities by package:

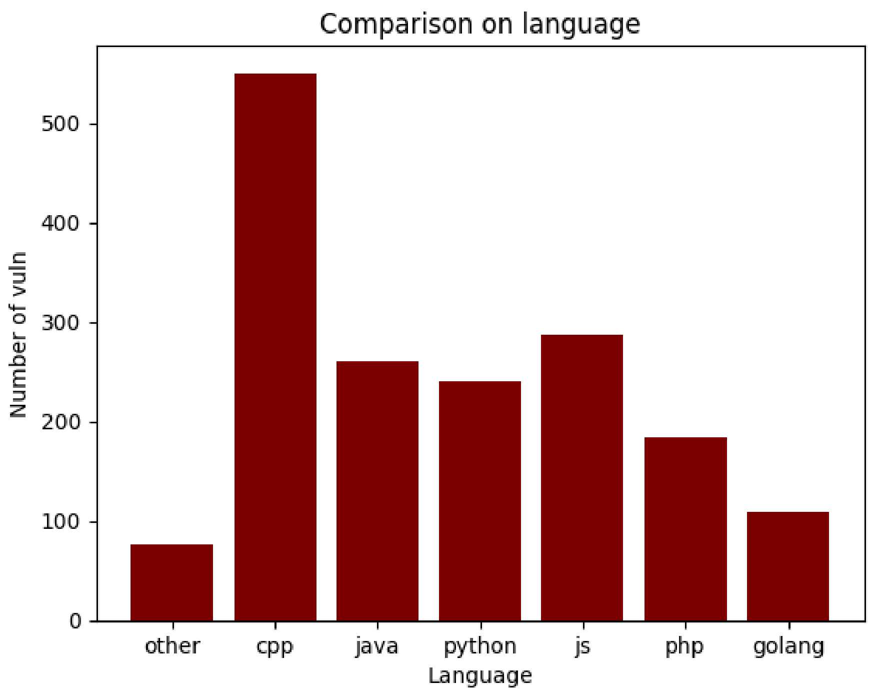
- Unmanaged (C/C++): 289
- Maven: 160
- Pip: 153
- Npm: 133
- Golang: 60
- Composer: 65
- Other: 40

Note that: other = cargo + nuget + rubygems + cocoapods + swift



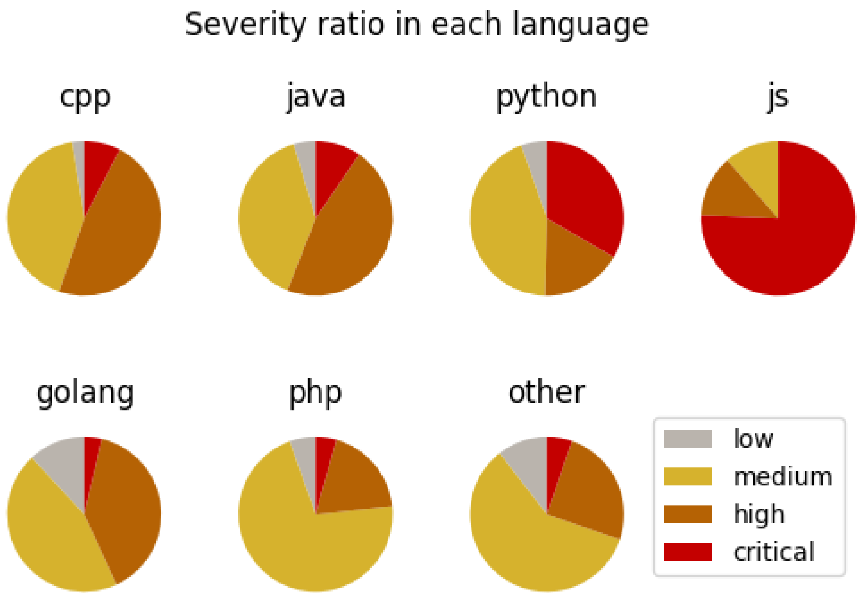
Unmanaged (C/C++) has the highest number of vulnerabilities at 289. This could be due to the complexity and popularity of C/C++ programming, which can lead to more security issues if not properly managed. Other packages' data usually indicate a smaller user base, fewer packages or better security practices.

3.3.5. Severity distribution based on languages



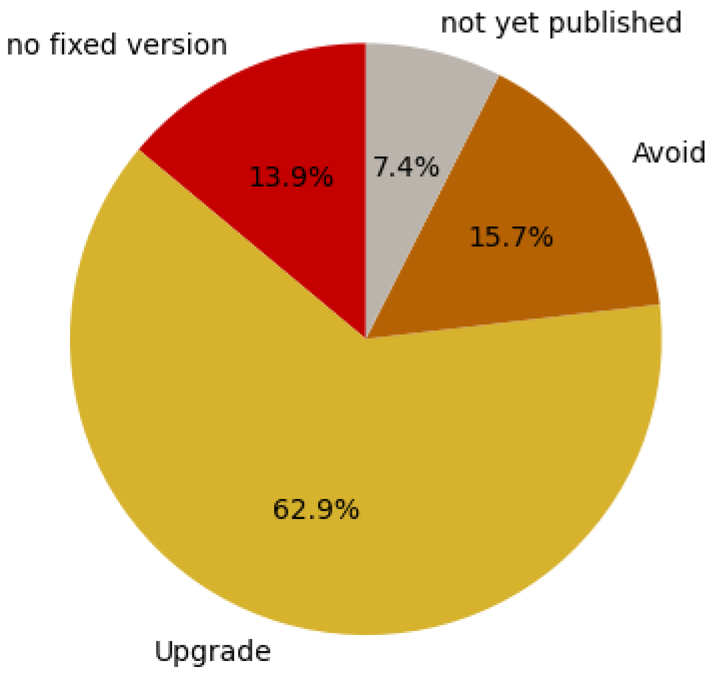
‘other’ languages include: rust, dotnet, ruby, objective-c and swift.

cpp’s popularity surpassed all other languages; this is to be expected since our analysis on package manager produced the same result.



Two notable points of this chart is that the percentage of critical vulnerabilities in python and js are significantly larger when compared to other languages. This is likely due to the fact that most of the malicious packages found by snyk are usually written in these program languages.

3.3.6. Remediation status



- ‘not yet published’ is for vulnerabilities with a fix pushed into the master branch but not yet published.

We can conclude that once a vulnerability is found, project maintainers will typically include a fix (if possible) in a future version, so keeping dependencies up to date is generally a good way to stay on top of known security vulnerabilities. Most malicious packages can be simply avoided; and workaround may be provided by snyk if there is no fixes available from the maintainers.