Beatrice Casey
Secure Software Engineering
Homework 1

Technical Report

Part I

Introduction

This program is intended to detect known vulnerable dependencies from the National Vulnerability Database (NVD) given a pom.xml file. It has a knowledge base in the form of an SQLite database that holds the information on vulnerabilities from the NVD API.

Design Decisions

Creating the request using the NVD API:

I chose to use the API because it allows me to capture all the vulnerabilities the NVD has. However, due to the way the API is organized, multiple requests need to be made to capture all of the entries in the NVD. Once the response is received, the json is parsed and only the relevant information is selected to reduce the size of the database: CVEId, the URI, the severity of the CVE and the versions affected. Unfortunately, the entries are inconsistent such that not every entry has an entry for the versions affected (instead, that information is stored in the URI).

Additionally, due to certain protection measures taken by the NVD, only a certain number of requests can be made per 30 seconds. To overcome any potential issues caused by this, the program sleeps for 5 seconds before making a new request. This increases the time it takes to load the database (it takes about 8 minutes to do so), however it ensures the program can run without fault.

Parsing the pom file

Taking inspiration from the link in the instruction sheet (**https://stackoverflow.com/questions/16802732/reading-maven-pom-xml-in-python/36672072#36672072**), I parsed the pom file in the same way. To assist in matching the URI from the NVD database, I then parse the groupId and artifactId to get the exact vendor and product name as given in the URI, and concatenate them as follows: groupId:artifactId. I then store all the dependencies in a dictionary, where the concatenated groupId and artifactId are the key and the value is the version.

Matching from the database to the dependencies

Once the database is filled and the pom file is parsed, then the database is searched for matching dependencies. Using the LIKE operator in SQL, the database it queried for instances where the URI is like the key from the dependencies dictionary, with wildcards on both ends. This ensures that all rows who's URI contains the key (regardless of what precedes it or follows it) will be returned. A list of all the matches is returned.

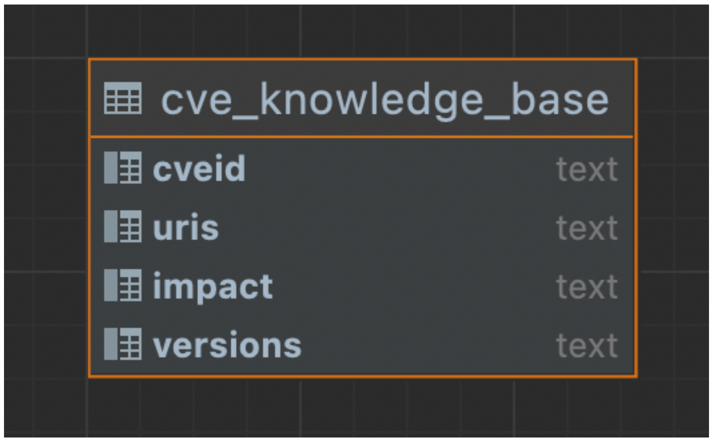
Checking all dependencies with the dependencies returned from the database

This step is important to make sure only dependencies that have a match in the database, and are a version that is vulnerable, are included in the output. If there are matches from the database, the dictionary of dependencies are then checked. If there was no version included in the json response, they are parsed from the URI (if possible). Using the versions library (**https://stackoverflow.com/questions/11887762/how-do-i-compare-version-numbers-in-python**), the vulnerable version is checked against the version specified in the pom file. If it is a vulnerable version, then it is added to the list of vulnerable dependencies.

Printing the Output

If the list of vulnerable dependencies is empty, the program writes to the results file that there are no dependencies. If there are vulnerable dependencies, then the appropriate information is parsed from the database results and output to the file.

Database Diagram



This is a diagram of the database. Each element is a text entry. There are 211,321 entries that range from the years 1999 to 2023. These are all of the entries that the NVD has. The only column that contains null values is the versions column, due to the fact that in the json response, not every CVE includes data on the affected versions.

Part II

1. **What is the motivation and problem being tackled by this paper and how they solve this problem?**

The problem being tackled by this paper is vulnerabilities in Open Source Software (OSS). The motivation behind this problem is that OSS is becoming more and more popular and increasingly used in projects. However, along with this growth comes the growth of vulnerabilities that are found in OSS and are publicly disclosed. A prime example of this is what we have talked about in class: a developer of some popular OSS modules purposefully created a vulnerability in his software because he was tired of companies using his software without paying him. His work caused multiple applications to crash and caused many problems for the companies using his software. Although OSS provides convenient tools for developers, it is still important to mitigate vulnerabilities that show up in these softwares.

In order to solve this problem, the researchers use a code-centric approach that combines dynamic and static techniques to detect vulnerabilities and more specifically, whether an application will reach the vulnerability via a dependency. They define constructs to be a structural element of the code, and define a construct change to be a tuple of the construct, a change operation, an AST (abstract syntax tree) of the fixed construct and an AST of the vulnerable construct. The researchers define a vulnerability as ‘reachable’ if dynamic analysis shows that the vulnerable construct is actually executed and static analysis shows the potential execution paths. The reason why this is important is that if a vulnerable construct will be executed, then it can also be exploited.

2. **Given the ideas explored in the paper above, how could you change your vulnerable dependency finder to incorporate some ideas described in it?**

To enhance the vulnerability dependency finder, I could implement the ideas presented in this paper to give the developer a better idea of which dependencies are more pressing to replace. For example, if a vulnerability in a dependency is reachable by the application, then there can be a note saying something along the lines of "Your application executes the vulnerable code in this dependency. Change it as soon as possible to improve the security of your application". Or something of this nature. In essence, rather than listing a bunch of vulnerable dependencies, it would give a ranking of the more pressing dependencies to change, similar to what is shown in figures 6 and 7 in the paper.

In addition to this, informing the developer of the Development Effort (that is detailed in the paper) would be helpful for the developer to understand how difficult implementing the changes would include. The results can be sorted as follows: the dependencies with reachable vulnerabilities are first, and the ones with the least development effort (eg are easiest to fix) show up first. They can also be sorted using the other metrics detailed, like Callee Stability, Reachable Body Stability, etc. This sorting information wouldn't be stored in the database because the program would need to be analyzed. So, after obtaining the vulnerable dependencies from the database and determining which dependencies are affected, then the static and dynamic analyzer in the program would be run to generate the ranking before the results are shared.