ATTENTION: Please make sure to upload the report as **PDF** instead of docx.

Homework 1 Report

Student Name – Dan Schrage

Design Decisions

Describe in this section your design choices when developing your solution. Specifically, include a description of:

- How are you extracting information from NVD (JSON data feeds or by using their API)?
- How did you tackle parsing of data feeds/pom files?
- How do you match CVE information to dependencies in a pom file? What heuristic(s) are you using?

You can also describe any technical challenges faced and how you overcame it.

My Response:

I'm extracting information from NVD via their JSON data feeds. I'm doing a request at this URL: f"https://nvd.nist.gov/feeds/json/cve/2.0/nvdcve-2.0-{year}.json.gz". Year is a variable that is changed in a for loop. This can be altered to change the range of data in the database.

I'm parsing data feeds using a similar format to parser.py in the class repo. However, I just removed the print statements and added each row to a list. At the end of each JSON file, I add all that data to the SQLite database. Additionally, I did have to add severity to the database, which involved some extra logic because of the CVSS metric. However, after digging through the return from the JSON feed and finding the severity type, I was able to add severity to the database as well. I referred to the stack overflow page referenced on the homework to create a read_pom() function that takes a pom file as an input and outputs a list of lists. Each inner list contains the following: [groupID, artifactID, version]. This is then output from the function. I figured out how to parse these files with the help of that page.

To match CVE information to the dependencies in the pom file, I first iterate over all dependencies. For each dependency, I run a SQL command to check if the groupID and artifactID are similar to the cpe_uri using the SQL "LIKE" command. I then iterate over all the returned results to eliminate overlap and ensure that versions match.

Ensuring that the version of the dependencies matched the version listed under the CVE was a difficult task. This was because the CVE had an inconsistency in the way it presents the versions. Sometimes it presents a singular version while other times it presents a range or a single version that it must be older/newer than. This just required some extra logic to solve.

Knowledge Base Statistics

Describe here how your submission is persisting data, i.e., your database schema. Include a database diagram that showcase the table(s) you have on your database.

For example: “The database includes the tables shown in Figure 1. It contains 2 tables. The Question table….”

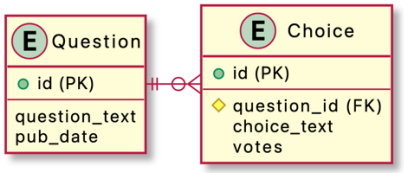
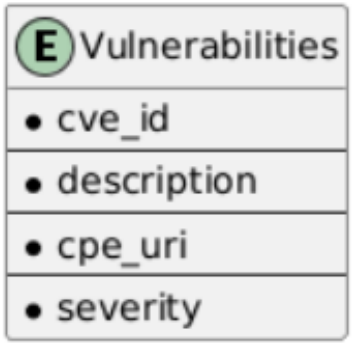


Figure 1 database schema

Also include basic statistics about the database by indicating:

- when was the knowledge base last updated (ex: August 27, 2025 at 2:14PM)
- how many rows in each table



I kept my database fairly simplistic. The following is a depiction of the SQLite database (vulnerabilities.db). CVE_ID was the ID number for each vulnerability found on NVD. It had the following format: “CVE_YEAR_ID” such as CVE-2021-3002. Description was a text description about what the vulnerability is and how it works. Cpe_uri was the corresponding CPE ID that contained information such as vendor, product, and version information. Finally, severity was a single word description, which was LOW, MEDIUM, HIGH, or CRITICAL. The knowledge base was last updated September 9th, 2025 at 6:45 PM. The current database has 273,959 rows. However, the number of

rows will vary based on the year range that is input. Currently, this table only has the 2021 JSON in it.

Research Synthesis

Include in here an answer to each question in Part 2.

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

   Most practical vulnerability scanners rely on metadata and vulnerability descriptions to be available and accurate. In reality, these are often missing or inconsistent. This can lead to a case where the vulnerable library is present, but there is no vulnerable code present. Or, incomplete data can lead to vulnerabilities not being recognized. Both of these situations provide misleading information for users and can lead to problems.

   To fix this problem, this paper suggests a code-centric approach. This approach identifies methods and classes identified by vulnerability fixes and use those as code-level signatures of a vulnerability. It then looks for those methods and classes in the loaded in the code rather than relying on metadata like names and versions. It then uses some graph analysis and dynamic execution techniques to determine whether these things are actually reachable in the code. This tool enables the distinction between present libraries and reachable methods/classes.

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

The first thing I would want to do is expand the current database to contain more pertinent information. This would include the languages and methods/classes that are specifically involved in a given vulnerability. I would also need additional information on the code we are testing for vulnerabilities. We would need access to all the libraries involved in the code. This data could all then be used to implement the strategies shown in this paper.

However, in a more realistic approach, I could use the additional information available in the JSON feeds from NVD. This would allow me to inform the user of my dependency finder about each vulnerability I report. This would allow the user to understand how a vulnerability works, but they would still have to determine for themselves if the vulnerability exists. This isn't as full-proof as the tool in the paper, but it is a much simpler approach. Providing the user with as much information as possible is helpful.