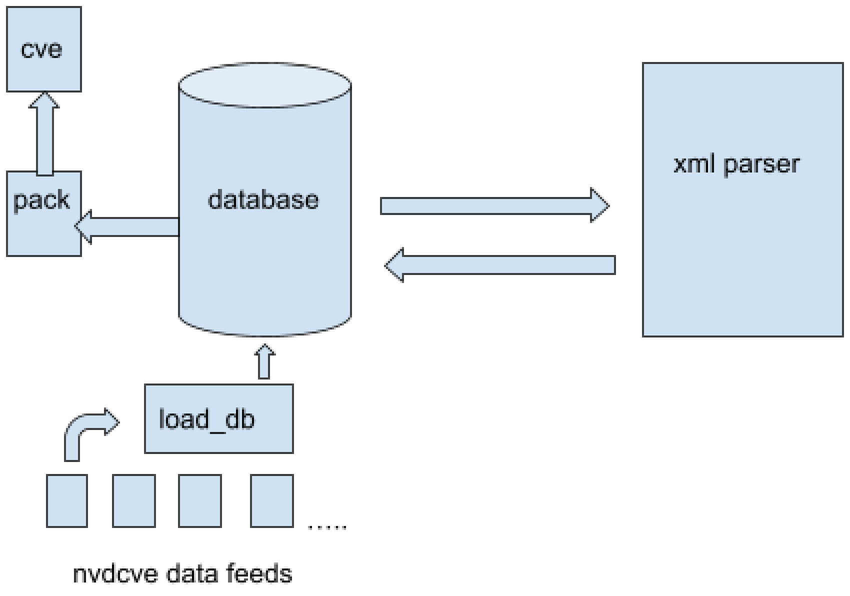
Homework 1
Secure Software Engineering
Report

The implementation of this project involves the design of a database structure, logic to fill the database, and logic to parse an XML file.



The diagram above illustrates the architecture of the program. The database contains two tables. One with package information, and one with CVE information. The package table maps each package that has a vulnerability to the CVE table through a foreign key-like relationship. This mapping is done by the load_db function, which parses the nvdcve json documents. The reason a separate table is used for CVE information is that one package may point to a number of CVE entries. The package table entries are keyed by a unique identifier which allows as many duplicate package entries as required.

Once the database has been created and filled, the xml_parser function may take a maven pom.xml file as input, and query the database for vulnerabilities in the listed dependencies. First, any CVE which lists the product name and manufacturer is returned to the xml_parser. Then, the versions listed in the CVE are compared against the version of the dependency listed in the pom.xml. There are many possible ways the CVE could have specified the version(s) of the vulnerable dependency. Because of this, some logic was needed to sort through inclusive and exclusive vulnerable version ranges, as well as situations where only a maximum or minimum version was stated, but not both.

CSE-60770 Part 2:

1. The motivation for this paper is the need for better vulnerability identification in large software projects that make use of open source components. The problem with many contemporary solutions is that they rely upon metadata and configuration files that list project dependencies. These sources of dependency information may not always be complete or available. Furthermore, there is a large cost involved in fixing these vulnerabilities located in project dependencies. An updated component may result in a change in the interface or functionality of the software, which needs to be thoroughly tested and combed through manually in most cases.

The solution presented here is the tool "Vulas", which combines an initial search for vulnerable packages using a knowledge base and commit history with routines for both static and dynamic code analysis. When searching for vulnerabilities, the program can make use of repository commit histories to identify where exactly patches occurred to fix the vulnerability. Programs are divided into logical constructs with associated dependencies and call hierarchies. Vulas can determine whether the main project's code actually reaches the vulnerable constructs in the dependency through static and dynamic analysis. Having this insight allows developers to prioritize the severity of vulnerabilities and determine whether their program is susceptible to exploitation at all.

2. Taking the ideas presented in this paper into account, updating the vulnerable dependency finder in this homework would need to involve some code inspection of the original project and dependencies. In judging the importance of a dependency to a project, the paper makes the assumption that the number of function calls to the vulnerable dependency construct correlates to the amount of work required to incorporate the patch, and the likelihood that it will reach exploitable code. A naive approach, yet something certainly worth the effort, would be to do a simple parse of the codebase, counting usages of the vulnerable dependency. This would offer a basic report of how tightly coupled the project is to the vulnerability. On a small scale, it would be practical for a developer to use this and compare the results to the vulnerable sections of the dependency to determine whether or not the project uses those parts of the dependency.
To make this addition more useful, some columns could be added to the database containing vulnerable function names if they can be determined. With this information, it can be determined whether the project uses any of the functions at all.

Another addition to the homework project would naturally be to check the entire hierarchy of dependencies, rather than just those listed in the project's maven configuration file. These chained dependencies can be found out through a link that is sometimes included in the pom.xml, or they could be queried on maven.org where the dependency's pom.xml can be retrieved and checked recursively. As seen in the paper, a vulnerability can be present in the project that originates from a distant child dependency. Since code analysis of each level of dependencies may not be feasible, the search for vulnerable function usage in the base project would be useful in measuring the potential impact of a vulnerability in a distant dependency. Continuing with the idea that usage count correlates to the development effort required to update the dependency, having the entire dependency graph available would allow the developer to see any mutual dependencies shared between the dependencies of the project, as well as the general connectivity of the dependencies. Having shared/connected dependencies may add another scalar to the formula for development effort presented in the paper.