Automating vulnerability remediation in Maven

CARL TERNBY & VIKTOR PETTERSSON
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Automating vulnerability remediation in Maven

Carl Ternby, Viktor Pettersson
psy15cte@student.lu.se, vi1221pe-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Martin Hell, Emil Wåreus

Examiner: Thomas Johansson

October 22, 2021

© 2021
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

The usage of open source software is growing and with it, the number of vulnerabilities that attackers can utilize in order to perform malicious activities. In order to mitigate them, it is therefore important to develop effective means of remediating said vulnerabilities. This thesis compares two different solutions for automating vulnerability remediation in regards to time efficiency. Both share the idea that a remediation should be performed by updating the vulnerable open source software to a version where the vulnerability is gone.

The first solution aims to do so by gradually updating the affected versions of open source software that a developer has directly imported in a project, until it finds an appropriate version. The second solution instead utilizes a graph database to store all available versions of an open source package and how it relates to other available open source packages. It can then be used to make secure versions directly query-able.

The simulations that were run in the project show that the graph database solution is far superior to the "brute-force" method when it comes to time-efficiency and also that such a graph would be scalable for use even with very large data sets.

Chapter 1

Acknowledgements

We would like to thank Debricked for making this thesis possible. Without the knowledge and foundations already present within the organisation, the results of this thesis would have been immensely harder to accomplish. We especially want give our gratitude to our supervisor at Debricked, Emil Wåreus for always being available for questions and opening up new ways of thinking. Lastly we would like to thank our supervisor at Elektro- och Informationsteknikinstitutionen, Martin Hell for the guidance he has given us when it comes to putting this thesis together.

Chapter 2

Popular Science Summary

Graph databases are proven to be very effective in representing large and complex relations of data. Can their power be harnessed to allow a user to, with a click of a button, be able to fix all security vulnerabilities that their software contains?

When developing a product, time and cost are important factors to have in mind. It is commonplace to use previous innovations and technology to a great extent in order to minimize these factors, since reinventing the wheel often proves to be costly and ineffective. An auto manufacturer may, for example, want to outsource the production of door locks to another company that has more experience in the field, rather than attempting to create new ones from scratch.

An analogy can be drawn to software development, where it is popular to make use of open source software when developing a product. Open source refers to software which is publicly available to see, modify or optimise. It can also be allowed usage in commercial products and the code is then often uploaded as packages to a package registry, making it easy to use. This way developers can avoid having to recreate basic components from scratch, while also being able to contribute to their further development. Open source packages can use code from other packages, making it a requirement to also include those in order for it to run properly. The needed package is usually referred to as a dependency of the package which requires it and it is possible for open source packages to have large trees of dependencies needed for it to run.

Since open source software is available for public scrutiny, it allows attackers to attempt to find weaknesses in the code which could be exploited. The strength in letting users modify the published code is that it allows bugs, such as security weaknesses, to quickly be fixed by releasing new versions of a package. This makes it important for developers to keep the open source packages they use up to date. Security weaknesses that can be exploited by an attacker are usually called vulnerabilities. In large dependency trees vulnerabilities can be found deeply nested in packages that you do not directly rely on, but that are dependencies of those packages.

This thesis aims to produce a reliable and efficient way to find the needed updates of directly used packages in a project so that vulnerabilities found in dependencies further down in the tree are fixed. When implemented, the solution should make it easy for users to secure their code. Two strategies were devised and pitted against each other, in order to find out which of them was the most time-efficient. The first strategy involved gradually updating the version of a directly

used package, that in turn has a vulnerable dependency, until it no longer depends on the harmful code. The second strategy aimed to re-create the entire trees of dependencies for all publicly available packages, together with their versions. One could then make requests to the database to find out how the packages should be updated to avoid the vulnerability.

The graph database proved far superior to the gradual updating of packages and allowed one to find a fix for a vulnerability in less than a second, compared to the other solution, which took several seconds and scaled poorly.

Table of Contents

1 Acknowledgements **iii**

2 Popular Science Summary **v**

3 Introduction **1**
3.1 Debricked 1
3.2 Goals & Challenges 2
3.3 Problem statement 2
3.4 Hardware 2
3.5 Contributions 3
3.6 Outline 3

4 Theory **5**
4.1 Packages/Dependencies 5
4.2 Maven 7
4.3 Software vulnerabilities 8
4.4 Relational Databases 10
4.5 Graph Databases 10

5 Implementation **13**
5.1 Remediation using a brute force strategy 13
5.2 Remediation using a graph database structure strategy 15
5.3 Simulations 18

6 Results **21**
6.1 Remediation using a brute force strategy 21
6.2 Remediation using a graph data structure strategy 21
6.3 Most common dependency depths for Maven packages 23
6.4 Package version count in the Maven registry 23

7 Analysis **25**
7.1 Remediation using a brute force strategy 25
7.2 Remediation using a graph data structure strategy 26
7.3 Most common dependency depths for Maven packages 27

7.4 Other observations 27

8 Conclusions **29**

9 Future work **31**
9.1 Develop simpler strategies for remediation 31
9.2 Scaling the graph database solution for production 31
9.3 Keeping the graph representation up to date 31
9.4 Widening the scope to more languages 32
9.5 The economic aspect 32

References **33**

A Figures **35**

List of Figures

4.1 Example pom.xml 7
4.2 An implementation of the example in Section 4.5.2 11
4.3 Example queries in the cypher language 12

5.1 Debricked data flow 14
5.2 Brute force strategy data flow 15
5.3 Graph database structure flow 16
5.4 Illustration of a root fix in the graph 18
5.5 Maven dependency node structure in Neo4j 19

6.1 Remediation using a brute force strategy 22
6.2 Remediation using a graph data structure strategy - minimal graph . 22
6.3 Remediation using a graph data structure strategy - big graph 23
6.4 Maven dependency depths 24

A.1 CPU 35
A.2 Memory - Handle 0x0013 36
A.3 Memory - Handle 0x001A 37
A.4 Memory - Handle 0x001D 38
A.5 Remediation using a graph data structure strategy - 10 node graph . 39
A.6 Remediation using a graph data structure strategy - 100 node graph 39
A.7 Remediation using a graph data structure strategy - 1000 node graph 40
A.8 Remediation using a graph data structure strategy - 10k node graph 40

Chapter 3

Introduction

This Master's thesis will address issues regarding the remediation of vulnerabilities in transitive dependencies when working with open source. The difficulty in preserving a secure project often lies not only in managing the dependencies that you import and depend on directly, but also in the dependencies of those dependencies. It is these dependencies, often called transitive, or indirect, that make dependency management difficult, since you cannot always decide which dependencies and versions that your direct dependencies rely on. The question then is; if a transitive dependency in a project is found to contain a vulnerability, how does one fix it?

We will investigate two different methods for solving this problem and benchmark them against each other, in an attempt to find which of them is preferred. The first method involves using brute force and building the project several times, each time bumping the version of the dependency that introduced the vulnerable transitive. By doing so, one would hopefully find a version of the direct dependency which no longer uses a vulnerable version of the transitive. The second method is to build a graph database over all dependencies known to exist, as well as their relations to other dependencies, in a graph structure. This enables the use of queries for finding out which version of a direct dependency is the first to no longer use the vulnerable transitive.

3.1 Debricked

The Master's project was done at Debricked AB [1] in Malmö and builds on their SaaS tool which, among other things, helps users to maintain security in their third-party software. To maintain the third-party code, or dependencies, Debricked scans customers dependency specification files and match the found dependencies to known vulnerabilities. The matches are then visualized and presented in a user interface along with relevant actions that can be done to remediate the vulnerabilities. One feature that Debricked offers is opening pull requests that fixes vulnerabilities directly in the users code base. This action is based on a brute-force solution where a remediation is searched for and tried until one successfully can remove the vulnerability. Right now, Debricked do not offer this action for customers using Java and the build automation tool Maven.

3.2 Goals & Challenges

The goals of this thesis are as follows:

- Implement a basic vulnerability remediation strategy that calculates dependency versions that fixes vulnerabilities in a brute force manner for code bases using the build automation tool Maven.
- Create a dependency graph database containing dependency versions and their relations from the Maven registry and implement a solution that communicates with it and uses its data to find remediations.
- Compare the solutions in terms of time efficiency.

Several challenges will be faced during the Master's project. One of the initial challenges will be to acquire a large number of versions for the dependencies that we are going to test. This is required in order to build an effective solution that works without compromise. The graph solution presented in this thesis will be a proof of concept, since the actual solution that will build upon this will require all versions of all dependencies in Maven. Another challenge will be the implementation of the basic remediation algorithm mentioned previously. This algorithm will use trial-and-error by bumping the versions of affected dependencies until the vulnerability is remediated. It is going to be important to implement the algorithm with efficiency in mind, since it will be constrained when it comes to resource use. Lastly, a graph database over all dependencies and versions that we have acquired will need to be created, together with an algorithm that effectively uses this database in order to find which versions of dependencies are required for the remediation.

3.3 Problem statement

When the graph solution is in place we will benchmark it by computing the time-efficiency with regards to the transitive depth of the vulnerable dependency. The transitive depth of the vulnerable dependency is of interest for both solutions. It possibly affects both the number of nodes that need to be traversed in the graph database solution, as well as the time it takes to build the dependency tree in the brute-force solution. By taking measurements on different depths for both solutions, we will be able to derive a rough estimate of their time-efficiencies. Using the average dependency depth in Maven as a whole, it is then possible to draw conclusions regarding which solution is preferred.

3.4 Hardware

The computer used for the simulations described in Chapter 5 is a ASUS ROG Zephyrus G14 GA401QM. Detailed information regarding the RAM and CPU can be found in Appendix A in Figure A.1, A.2, A.3 and A.4 respectively.

3.5 Contributions

The goal of this thesis will be to contribute the following:

- A generic algorithm for solving vulnerabilities by gradually increasing the version of a direct dependency until the vulnerability is remediated.
- An implementation of a graph database containing available dependency versions and their relations that can be queried in order to find remediations for vulnerabilities.

3.6 Outline

In Chapter 4 theory regarding the problem and the two proposed solutions will be introduced, in order to establish the terminology and concepts that were useful during the implementation and analysis. The actual implementation of the solution will be gone through in Chapter 5 with the results presented in Chapter 6. In Chapter 7 these results will then be analysed and discussed, giving the basis for the conclusion presented in Chapter 8. Lastly, Chapter 9 will bring up potential future work that can assist in further developing the solutions for use in a live application.

Chapter 4

Theoretical Background

4.1 Packages/Dependencies

Open source software is often described as a piece of code that anyone can see, modify or optimise, due to it being publicly available. Depending on the license connected to the software, it may or may not be used in certain applications [2]. This thesis will be focusing on dependencies of open source software, meaning external code that is required for the software to be used. Dependencies differ in their structure and the different types have varying names, such as packages, modules, gems and snaps etc. In this thesis, an openly published piece of open source software will be referred to as a *package*.

4.1.1 Open source software registries

There are several different approaches for connecting packages throughout open source. One approach is to wrap the package with defined meta code so that it can be used generically in projects using the same meta code structure. These packages using the same setup can then be published to a central registry store where they can be used by a package manager. One of the larger package managers with such a registry is Maven [3], which will be described more in detail in Section 4.2.

4.1.2 Dependency trees

During development, one can implement a new package that is built on top of already existing packages. These are often called dependencies to the new package. Later on, this new package might become a dependency to another newly built package. We now have a set of dependencies depending on each other in a chain. The dependencies of a dependency is usually called its *transitive dependencies* [4]. In $A \rightarrow B \rightarrow C, A$ is a package that has a direct dependency B and a transitive dependency C. To B, C is a direct dependency. A package manager is then used for assisting with handling all of these packages and their relations [5].

The management of transitive dependencies differs a lot. Some package managers do not permit the use of multiple different versions of a dependency, while others do. Some even lock all transitive dependencies to a certain set of versions in order to not get conflicting builds. Maven [3] and Composer [6] are examples of

package managers that do not accept multiple versions of the same dependency, while npm [7] is an example of a package managers that does.

4.1.3 Breaking changes

When actively maintained, packages often receive updates and fixes with differing reasons. It could be because of bugs inside the APIs or new functionality that is being added due to new demands or interests. Sometimes parts might no longer be used and could then be removed. Depending on the impact the changes make to the package, users might be affected. If a certain API is changed so that others cannot use it in the same way as before, that is usually called a *Breaking change* [8].

4.1.4 Package versioning

Due to the complexity of managing a large tree of dependency relations, an issue sometimes faced when packages depend on each other is what is commonly known as *dependency hell*. Package versions are often allowed to be declared loosely using, for example, a version interval, or locked to a single version. One can imagine a large tree, where versions are not locked and different packages and their transitive dependencies get automatically updated within the allowed version interval at different points in time. In that case it is difficult to know if the updates will introduce breaking changes to a project. The simple solution to the problem would be to make sure that all packages and their dependencies have locked versions, but at the same time that means that the project will not receive new updates when needed. It also still requires the maintainer to have to do research every time a new version is to be used, in order to make sure that no breaking changes have been introduced [9].

One common solution for assisting in making it easier to understand what an update means to a project and reducing the risks of dependency hell is *Semantic versioning*, also known as *SemVer* [9]. Semantic versioning is a rule-set for managing different versions of packages and specifies how these should be incremented/decremented, in order to be transparent about what a change in version will lead to.

The versions in SemVer are divided into "MAJOR.MINOR.PATCH" (for example 2.4.3), where the following scheme is applied:

- "MAJOR": Incremented whenever changes to the open API are made in such a way that breaking changes are introduced.
- "MINOR": Incremented when new functionality is added to a package that is backwards compatible and will therefore not introduce any breaking changes. Changes in version are allowed within "MAJOR.X"
- "PATCH": Incremented when changes that fix bugs are made to already existing functionality in the package. Changes are allowed within "MAJOR.MINOR.x"

SemVer declares that "MAJOR", "MINOR" and "PATCH" should all increase numerically and the each identifier should be a positive integer, without any zeros before the actual integer.

SemVer also describes guidelines for how to compare SemVer defined versions. To determine what version is newer than the other, one must consider the numbers of each section where the major identifiers precedes minor identifiers which in turn precedes patch identifiers. The following is, for example, true for SemVer [9]:

$$\mathbf{1.0.0} < \mathbf{1.0.1} < \mathbf{1.1.0} < \mathbf{1.10.1} < \mathbf{2.1.1}$$

4.2 Maven

Maven is a package manager often used for Java projects. It does not only cover the management of packages but also has support for setting up other project operations, such as compiling, testing or deploying. These operations are normally done through so called plugins [10]. Maven's main concept is built around the project object model, POM and the corresponding file called *pom.xml* which is a project specification file in XML format [11]. A basic example of a *pom.xml* with a plugin called *maven-compiler-plugin* and a dependency called *maven-artifact* can be seen in Figure 4.1.

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-module</artifactId>
  <version>1</version>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
      </plugin>
    </plugins>
  </build>
  <dependencies>
    <dependency>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-artifact</artifactId>
      <version>1.2.3</version>
     </dependency>
  </dependencies>
</project>
```

Figure 4.1: Example pom.xml

4.2.1 Dependency management

Defining what dependencies a Maven project requires is done under the tag *dependencies*. When the project is built according to the POM, the dependencies defined gets resolved and downloaded from a defined registry. The default Maven registry is called the *Maven Central Repository*, however it is possible to within the POM declare other registries that are to be used [12]. Since no other registry has been defined in Figure 4.1 the dependencies and plugins would be downloaded directly from the Maven Central Repository [13].

Maven supports having setups with dependencies depending on each other, i.e., transitive dependencies. As stated in Section 4.1.2 Maven does not accept several versions of a dependency in the tree. So a dependency x with version y cannot have a another version on another depth. Java uses its *classpath* to resolve imports in projects. This is helpful in the case that the same dependency has two different versions, since they otherwise would end up on the same *classpath* [14] making it impossible to know what to reference during compilation [4].

Maven uses so called *Dependency mediation* to decide what version will be resolved in the case where two different versions occur as requirements during build time. The one that gets resolved is the one that is closest to the direct dependency in the tree. The Maven project defines it as the *nearest definition* [4]. An example of how this works is the case when we have a dependency x with version y on level 1 (0 being direct dependencies) and on level 2, the dependency x is required again but this time with version z. Here Maven would use version y since it is closest to the direct dependency level. Because of this functionality, a transitive dependency can always be forced to a certain version if it is declared a direct dependency in the POM declaration.

4.3 Software vulnerabilities

Software can be affected by a range of different issues, varying in severity. Some might have devastating effects, while others barely have any impact at all. These issues sometimes result in software vulnerabilities that are exploitable by malicious actors. There are several organisations that try to address or aid in the seemingly never ending fight between the bug fixers and attackers. One such organisation is the National Institute of Standards and Technology (NIST) and their project the National Vulnerability Database (NVD). Some of the contributions made by the project have become standards for managing software vulnerability data. They also host an openly published vulnerability database that matches software packages with vulnerability definitions. The matches describe what versions are vulnerable to a certain weakness, what the issue is and sometimes how it can be addressed. This is either done through code changes or pure upgrades or downgrades of the package. The matched vulnerability is called a CVE, which stands for Common Vulnerabilities and Exposures, and will be described more thoroughly in Section 4.3.1. NVD uses a definition they call CPE, or Common Platform Enumeration, for defining packages [15], which will be described in Section 4.3.2.

4.3.1 CVE

As mentioned above, NVD has chosen the CVE identifier for vulnerabilities in their database of package-vulnerability matches. CVE has become a widely used identifier for a specific vulnerability in information technology systems. The standard is published and maintained by the MITRE Corporation which has copyrighted it. A CVE Entry must include the following [16]:

- The name of the package affected by it.
- The version(s) of the package that is affected or that fixes the vulnerability.
- A CVE ID, for example CVE-2021-1337.
- One of either vulnerability type, root cause or impact.
- At least one public reference.
- A prose description.
- An indication of whether or not the vulnerability only affects products that have stopped being supported.

4.3.2 CPE

A CPE, or Common Platform Enumeration is another information security related standard. This standard is used to generalise and identify software packages though naming [17]. It is published and held by NIST which have also published a CPE dictionary for packages [18]. A CPE identifier contains, among other things, the following:

- The version of the CPE standard used.
- Related programming language or runtime environment.
- The name of the vendor.
- The name of the package/product.
- Version of the package.

An example of CPE could be the following JavaScript package called lodash that has the SemVer version 0.1.0:

cpe:2.3:a:lodash:lodash:0.1.0:*:*:*:*:node.js:*:*

When specificed in a CVE, a CPE can sometimes be more general, by having an asterisk '*' in the version tag as well. This would mean that all versions of that package is affected by the CVE. Another example often used by NVD is specifying a version interval, within which a package is vulnerable, which can be seen in for example [19].

4.4 Relational Databases

A relational database consists of data stored in tables containing rows that represent records and columns that represent attributes. It is based on the relational model concieved by E. F. Codd in 1970 [20]. The ideas behind the relational spawned from an era where storage was both limited and expensive and therefore data needed to be compressed. The relational database is still to this day effective and useful in a lot of situations where relationships are of the *one-to-one*, *one-to-many* or *many-to-one* sort, but when it comes to *many-to-many* relations, the performance tends to leave a lot to be desired [21].

4.4.1 MySQL

MySQL is an example of a relational database management system using SQL as language for creating queries. It can hold a variety of different data types, which are defined in the table where the data is stored. In a MySQL database, primary keys and foreign keys are used in order to define relationships between tables [22]. Due to these relationships being highly connected, it can result in issues with performance for larger databases. The more tables and larger a relational database is, the more joins are often required for retrieving the data, resulting in higher query retrieval times [23].

4.5 Graph Databases

A graph database is an alternative to the traditional relational database, consisting of nodes and edges. It is part of a family of databases called NoSQL which, unlike relational databases, do not use links between tables to define relationships, allowing one to access objects in a quick and simple way, due to the explicit direct relations between data and lack of expensive table joins. The concept of graph databases was developed as a way of effectively handling *many-to-many* relations and making it easy to model and traverse data with high performance capabilities [24].

4.5.1 Components

Unlike primary and foreign keys in a relational database, the edges in a graph database define the relationship between the nodes in a node-edge-node pattern. Both the nodes and the edges are able to have properties and labels assigned to them that helps describe both what the node represents and the necessary information about the node's relation to other nodes [21]. To give an example of why this is often very useful when creating a graph database, we can consider a graph database that has nodes describing both people and different physical and conceptual objects. If we want to describe that a person "John" is fluent in French, we could add a node with the label "person" and property "name: John", another node with the label "language" and property "name: French" and an edge with the label "is fluent in", connecting the two nodes. If we were to populate the database with several other nodes and relationships we would then be able to easily query

the database to find all languages that John is fluent in. Due to the fact that edges are allowed to have properties, we could further develop this and rename the label of the edge to "proficiency" and instead include the property "skill_level: fluent". By doing this we are now able to do a single simple query and get all Johns proficiencies in one go. In the same database, other types of objects and relationships could be added with similar functionality by using properties and labels.

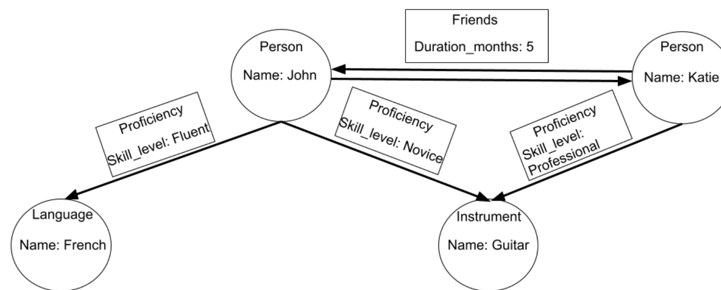

Figure 4.2: An implementation of the example in Section 4.5.2

4.5.2 Neo4J

Neo4J, developed by the Swedish company Neo4J Inc. is an example of a graph database management system that uses nodes, edges and properties to build a graph structure. It supports the labelled property graph model, allowing nodes and edges to have several different labels. It also allows for bidirectional edges [20]. Figure 4.2 illustrates an example Neo4J graph where the top relation between John and Katie is bidirectional, while the relations between the people and the proficiencies are unidirectional. All the edges in the example contain properties, allowing for more advanced queries to be used to fetch data. Neo4J uses the *cypher* query language for both the creation and retrieval of nodes and edges in its database [20]. The cypher language allows for a wide variety of operations, and the creation of the graph in Figure 4.2 can be done through the queries shown in Figure 4.3. In the figure, there are also queries for retrieving data from the graph.

```
//Create nodes

CREATE (x:Person{name:"John"});

CREATE (x:Person{name:"Katie"});

CREATE (x:Language{name:"French"});

CREATE (x:Instrument{name:"Guitar"});


//Create edges

MATCH (x:Person), (y:Person) WHERE x.name = 'John'
    AND y.name = 'Katie' CREATE (x)-[r:Friends (Duration
       \_months):5)]->(y);

MATCH (x:Person), (y:Person) WHERE x.name = 'John'
    AND y.name = 'Katie' CREATE (x)<-[r:Friends (
       Duration\_months):5)]-(y);

MATCH (x:Person), (y:Language) WHERE x.name = 'John'
    AND y.name = 'French' CREATE
    (x)-[r:Proficiency {Skill_level: 'Fluent'}]->(y);

MATCH (x:Person), (y:Instrument) WHERE x.name = 'John'
    AND y.name = 'Guitar' CREATE
    (x)-[r:Proficiency {Skill_level: 'Novice'}]->(y);

MATCH (x:Person), (y:Instrument) WHERE x.name = 'Katie'
    AND y.name = 'Guitar' CREATE (x)-
    [r:Proficiency {Skill_level: 'Professional'}]->(y);


//Retrieving data

MATCH (x:Person{name:"John"})-[skills:Proficiency]->(y)
   RETURN skills;
    //Returns the "French" and "Guitar" nodes

MATCH (x)-[friendship:Friends]-(y: Person{name: "John"})
    WHERE friendship.Duration_months > 4 RETURN x;
    //Returns the "Katie" node
```

Figure 4.3: Example queries in the cypher language

Chapter 5

Implementation

In this chapter we will present the implementation of the two remediation strategies. Each strategy will have its own relying infrastructure, described in the first section of both strategies. An in-depth description of each strategy will then follow.

5.1 Remediation using a brute force strategy

The idea behind the name "brute force strategy" stems from that the strategy theoretically is quite simple. It aims to continuously bump the version of the direct dependency that transitively has imported a vulnerable dependency. This is to be done until a version of the direct dependency is found that has either removed the vulnerable transitive dependency or has changed its version to one outside of the vulnerable range. By making as small changes as possible between each bump we would, for versions using SemVer reduce the risk of breaking changes, since we would prioritize patches and minor version changes over major version changes.

5.1.1 Infrastructure

The brute force strategy is implemented in Debricked's core infrastructure, using PHP 8.0 [25] and MySql [26]. Figure 5.1 shows the flow of that core.

The brute force strategy logic is operating as its own entity next to *Rule engine* and *GitHub Consumer*, called the *Root Fix Consumer*. It uses *Redis*, *Workers*, *MySQL* and the *Webservice & API* to operate. It is however not shown in Figure 5.1 but is described more in detail in Section 5.1.2.

5.1.2 Design

Root Fix

The brute force strategy starts with a entity called root fix. The name originates from the fact that this entity contains the new version for a root dependency that fixes a vulnerability in a project.

Inside Debricked's service Maven projects are scanned and in that process, parsed dependencies are matched with CVEs. The scanning also connects found vulnerable dependencies with its related dependency ancestors making it possible to fetch those after the scanning is complete.

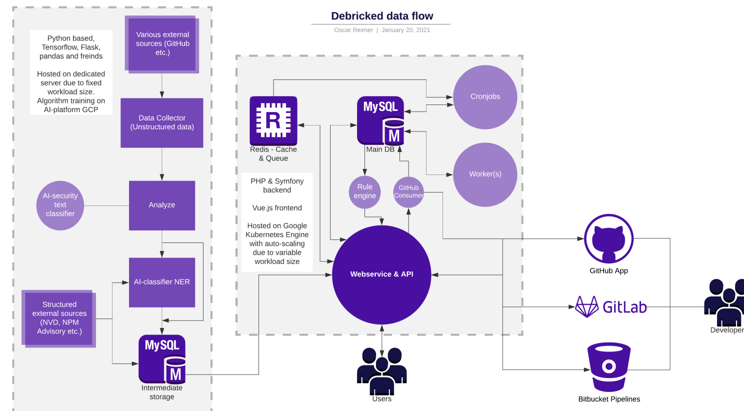

Figure 5.1: Debricked data flow

The root fix entity has four components.

1. CVE related data.
2. The vulnerable dependency.
3. The direct dependency in the Maven project that transitively imported the vulnerable one.
4. The version of the direct dependency that remediates the vulnerable dependency. i.e the fix. Before the remediation is done this is just allocated space.

Root fix consumer

In order to handle big loads of root fix entries, the service uses a queue system set up specifically for it. It also has a consumer class that handles queued root fix entry jobs called the Root Fix Consumer.

The Root Fix Consumer has a method called *Invoke* that describes what to be done when a root fix enters.

Maven Remediator

When *Invoke* is called in the Root Fix Consumer, the Maven Remediator is in turn used to call its two methods *fetch versions* and *remediate*. *Fetch versions* takes data regarding the direct dependency from the root fix entry and makes requests to Maven Central Repository to find all existing versions through an intermediate class called the *Maven Version Fetcher*.

Remediate then works its way through the version set returned from *fetch versions* with the aim of finding a fix for the root fix entry and thereby the vulnerability. The search for such a solution has the following steps, also shown in in Figure 5.2:

1. Get the next available version in the version set.

2. Generate an isolated pom.xml file with one direct dependency (the one defined in the root fix) with the version from Step 1 applied. After this, the pom.xml gets resolved with Maven, using a plugin called *dependency:tree* that visualizes the dependency tree in a format that is interpretable.
3. Analyze the generated dependency tree. If the vulnerable dependency is gone, the algorithm terminates and the secure version is returned and set in the root fix entry. If the vulnerability is still present and there are more versions to try, the algorithm returns to Step 1 again. If instead the version set is empty, the algorithm terminates without a found fix.

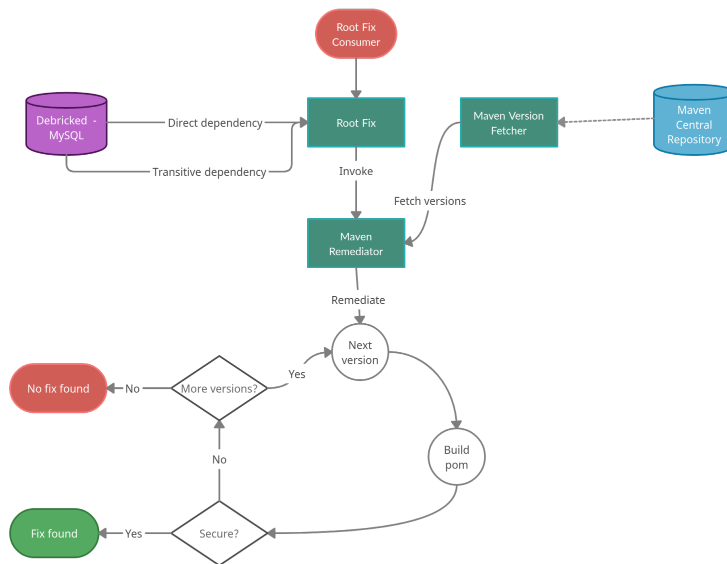

Figure 5.2: Brute force strategy data flow

5.2 Remediation using a graph database structure strategy

The second strategy involves creating a graph database, containing available Maven dependency versions and their relations. A way of querying it also needs to be implemented in order to find the root dependency versions required to fix a vulnerability. By finding root dependencies that are either not related to the vulnerable transitive dependency or related to a safe version of it. The hope is to be able to get similar results, in terms of safe versions, as the brute force strategy, but through a less convoluted procedure.

5.2.1 Infrastructure

We decided to use Python 3.9 [27] with a microframework called Flask [28] to implement the application communicating with the graph database. The reason

for choosing Flask was that it is fairly lightweight, easy to use and flexible when it comes to decisions such as which databases to use. This choice was also influenced by the fact that the solution did not have to be implemented into Debricked's core infrastructure, like the brute force strategy described in Section 5.1. Instead it was to be designed as a standalone application reachable through a number of REST API endpoints. For the actual graph database implementation we used Neo4j, described more in Section 4.5.2 with the py2neo library for communicating with the graph through Python. In order to be able to queue repository scan requests, we implemented a Redis [29] queue. Lastly, we used Docker [30] with the tool Docker Compose [31] to package our application into three different containers, one for the flask app, one for the Redis queue and one for Neo4j.

The flow of this setup can be followed more in detail in Figure 5.3.

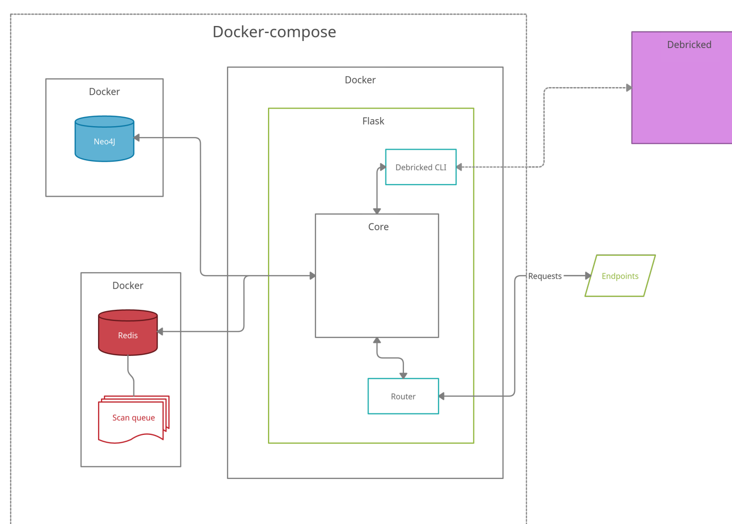

Figure 5.3: Graph database structure flow

5.2.2 Design

Flask

The graph database solution uses Debricked's tool for parsing dependencies and relations from Maven projects, and fetches those results when adding it to the graph. When adding a repository to the graph, the first step is to clone the repository and branch chosen locally, and then use Debricked's CLI tool to upload the files and also perform a dependency scan in the main tool. The CLI tool keeps track of the scanning progress and when it is finished, the next step is to get the dependencies and relations from the scan. This is done through an API endpoint that we implemented that fetches all vulnerable and non-vulnerable dependency names, versions and relations for the upload. Once they have been fetched, they are translated into **Dependency** and **Relation** entities and then inserted into

the Neo4j graph. The communication between python and the graph is possible with the use of the py2neo library.

Neo4j

After having received all the information needed from the scanned repository, we iterate through the data, adding the appropriate nodes and relations when needed. We do not allow duplicates in the graph, so if a certain node already exists, we do not add it, and instead only add the potential new relations, if any exist. Therefore all repositories which are scanned are interconnected when needed and the end result is a large graph with nodes for all dependencies parsed and edges for all relationships. Py2neo is not only used for adding new nodes and relationships to the graph, but also for getting the appropriate data when needed. Through an API it is possible to fetch certain data, in order to find the remediating version of a dependency. Based on the fact that NVD often defines vulnerabilities through a vulnerable interval, we have created a main API endpoint to be used for the remediation. This takes the name of the direct dependency and the vulnerable transitive dependency, as well as the vulnerable version interval as input. This is used when the vulnerability is within a certain interval, and the graph is queried to find a version of a direct dependency where that specific transitive dependency is either outside of that interval, or is not interconnected with it.

Graph database

We want the graph to be as simple as possible and therefore only include the minimum required to accomplish the task of remediating vulnerabilities. A node contains two properties, a *name* and a *version* and nodes relate to each other through a relation called "TRANSITIVE_OF" without any properties, which points upwards in the graph. This is done so that we can traverse the graph from a safe transitive dependency version up towards the direct dependencies which depend on it and that remediates the vulnerability. The format of a CVE on NVD often contains a vulnerable interval or specific vulnerable version, as mentioned in Section 4.3.1. Due to this fact, we have two main queries for getting a root fix from the graph database:

1. A query which takes the name of the vulnerable transitive and the direct dependency, along with the vulnerable interval. This is used to find all direct dependency versions connected to versions of the transitive dependency that are outside of the interval. The first step of the query is to search for transitive dependencies that have a version outside of the vulnerable interval. Step two is then to traverse up the tree of each of these dependencies until the end or until it finds a dependency whose name matches the name sent in as the direct dependency name. In step three it then returns that root node. An illustration of these three steps can be seen in Figure 5.4

2. A query which takes the name of the vulnerable transitive and the direct dependency and finds all direct dependency versions where the transitive does not exist. This works well in conjunction with query 1 if the transitive

was added or removed at a certain point in the direct dependency's version history.

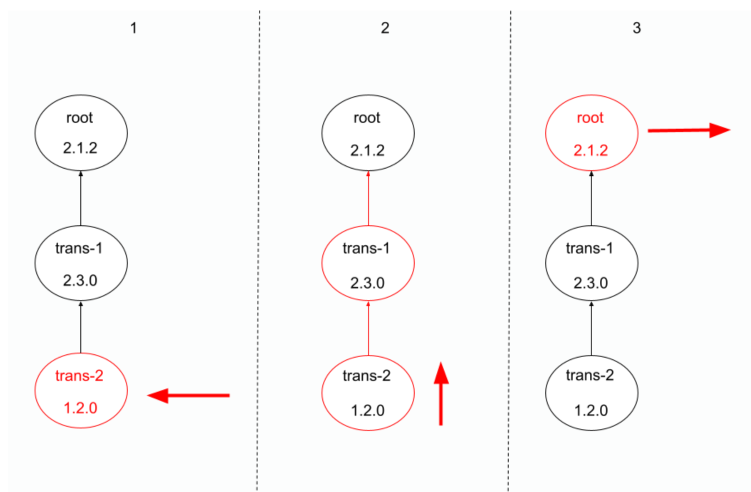

Figure 5.4: Illustration of a root fix in the graph

5.3 Simulations

As mentioned in Section 3.3, we decided to compare the two solutions using time efficiency with regards to the depth of the vulnerable transitive dependency to be remediated.

5.3.1 Maven registry

We considered it a risk that if the two strategy simulations did not share the same prerequisites, the results could not be compared due to different dependency setups. In order to create a shared defined dependency pool we created our own Maven registry on the platform GitHub. We published in total eight packages, the first being *mvn1* and the last *mvn8*. For each package we published two versions, *1.0* and *2.0*. We also made *mvn1* the leaf dependency in the transitive chain. *mvn2* then used *mvn1* as a direct dependency, while *mvn3* used *mvn2* and so on until *mvn8* lastly imported *mvn7*. We created two such transitive branches. One version *1.0* branch and one version *2.0* branch. This setup is illustrated in Figure 5.5.

5.3.2 Remediation using a brute force strategy

Since we wanted to use our newly created Maven registry with published packages we first of all were forced to make changes to the Maven configuration inside the Debricked service to make use of our registry instead of the Maven central repository.

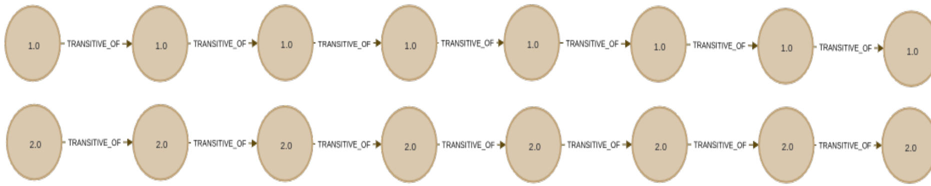

Figure 5.5: Maven dependency node structure in Neo4j

We wanted to simulate the time needed to remediate a vulnerable dependency on different dependency depths. Since the simulation focused on transitive dependency depths, it was set up to only require one version bump in order to resolve a vulnerability. If we were to add more versions that would scale linear and so we did not see any value in adding more. With the published Maven packages we could now explicitly say what depth was to be tested by always pointing out *mvn1* as the vulnerable dependency but for each new depth change the direct dependency from the initial *mvn8* down to *mvn1*. By doing so we could simulate depth 7 to 0.

Each remediation attempt started with inputting the relevant direct dependency along with the vulnerable one to the brute force logic. It then returned the remediating version of the direct dependency. When this was finished, the simulation saved the time spent on remediating and lastly cleared the cache, so that each attempt would be unaffected by the others. For each depth, 100 remediation iterations were run. After this was done, a mean time value was calculated and saved as the time result for the particular depth. The resulting data set was then translated into a bar graph together with the standard deviation for each level.

5.3.3 Remediation using a graph database structure strategy

Minimal graph

This simulation was similar to the brute force strategy simulation setup but instead of using the brute force remediation logic we made use of the graph system. The big difference was that the two package branches that were published to our own Maven registry were now pushed to our Neo4j instance, as shown in Figure 5.5. With that done we could run the simulation by inputting the direct dependency and the vulnerable one to the flask REST API endpoint, that in turn made queries to the Neo4j instance, instead of calling the brute force logic. Just like for the brute force simulation, each depth was run 100 times and the mean time per depth and standard deviation was saved and translated into a bar graph.

Larger graphs

The number of nodes in the graph database was expected to affect the query times since it determines how much it needs to search through. Because of this, we thought that it would be interesting to do several simulations on the graph solution, each time expanding the graph with a lot of other dependency nodes and relations outside of the two relevant dependency branches. The graph was filled

in stages, adding 10, 100, 1000, 10000 and 100000 nodes with the same names, but different versions than the relevant branches. The same simulation procedure was then run again for each of the stages, attempting to find out if the size of the graph mattered for the end result.

5.3.4 Most common dependency depths for Maven packages

In order to evaluate which solution was the best in a real life scenario, we also wanted to evaluate common dependency depths in Maven repositories. By dependency depth we mean what level of transitivity a certain dependency has. If we consider Maven packages as a whole, what depths are the most common? This is relevant for the simulation, since we can cross-reference it to the graphs from the simulations made in the different solution and see which solution fares best in the most common dependency depths.

Since Debricked has a lot of users with Maven setups in their projects we felt that it was suitable to use that data for this analysis. Due to the fact that the same project could occur several times in the set we made sure to only use the latest release of each project. If we had not made this filtering we felt that there was a big risk of the same samples reoccurring, thereby corrupting the results.

With the filtering done we wrote code that fetched each latest Maven dependency scan per project from Debricked. On each scan we got raw dependency relation data. From these we traversed all branches of the dependency tree and saved each dependency depth as a sample to our result. An important note here is that the same dependency could occur twice as a sample. This was due to that a dependency could share another, thus enabling different sub branches for one dependency. The counted depths were then plotted as a bar diagram by using the python library matplotlib [32].

5.3.5 Package version count in the Maven registry

As mentioned in Section 5.3.2, we only require one version bump for each simulation to solve a vulnerability. It is still, however, interesting to look at how many version bumps that could be expected in the general case. Therefore, we decided to scrape the Maven central repository [13] in order to find out how many versions a given Maven package has in average.

Chapter 6

Results

6.1 Remediation using a brute force strategy

The results from the brute force strategy simulation can be seen in the bar graph in Figure 6.1, with the bars representing the time it took to find a remediating version and error bars represented by lines in the center of each bar. With a depth of 0, meaning a vulnerable direct dependency without any transitive dependencies, it took 5.25 seconds (std: 0.19s). When the depth is increased to 1 and the imported direct dependency has a vulnerable transitive dependency directly under it, the elapsed time was increased to 6.30 seconds (std: 0.41s). Furthermore, for depths 2, 3, 4, 5, 6 and 7, the dependency remediation took 6.88 (std:0.29s), 8.29 (std: 0.42s), 8.60 (std: 0.31s), 10.12 (std: 1.14s), 10.99 (std: 0.33s) and 12.20 (std: 0.44s) seconds respectively.

6.2 Remediation using a graph data structure strategy

6.2.1 Minimal graph

Shown in Figure 6.2 are the results of the simulation made on the minimal graph, which contained just the nodes required to simulate a remediation. In this simulation, the remediation of a vulnerable direct dependency (depth 0) took a total of 0.0098 seconds with a standard deviation of 0.0009 seconds. For depth 1, the mean time decreased to 0.0090 seconds, with a standard deviation of 0.0008 and depths 2, 3, 4, 5, 6 and 7 took 0.095 (std: 0.0009s), 0.0088 (std: 0.0010s), 0.0088 (std: 0.0009s), 0.0092 (std: 0.0009s), 0.0095 (std: 0.0009s) and 0.0083 (std: 0.0008s) seconds respectively.

6.2.2 Big graph

The results for a graph with 100,000 nodes can be seen in Figure 6.3. For depths 0, 1, 2, 3, 4, 5, 6 and 7 it took 1.373 (std: 0.059s), 0.143 (std: 0.013s), 0.168 (std: 0.010s), 0.204 (s 0.242 (std: 0.017s), 0.270 (std: 0.018s), 0.310 (std: 0.018s) and 0.331 (std: 0.015s) seconds respectively. The results for when the graph had 10, 100, 1000 and 10,000 nodes can be seen in Figures A.5, A.6, A.7 and A.8 respectively, in Appendix A.

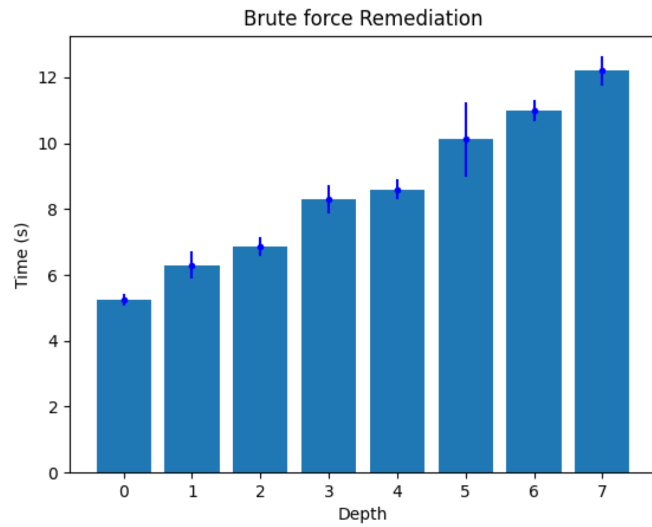

Figure 6.1: Remediation using a brute force strategy

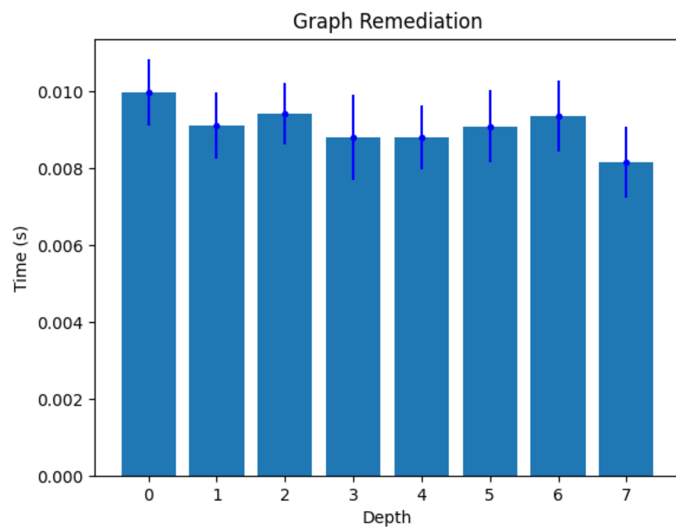

Figure 6.2: Remediation using a graph data structure strategy - minimal graph

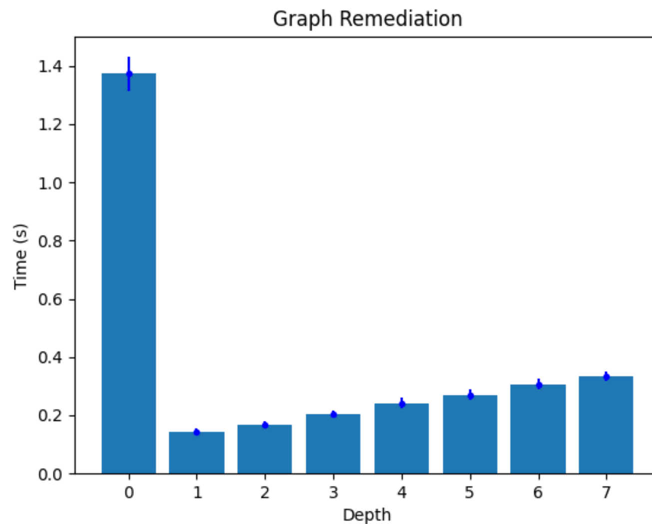

Figure 6.3: Remediation using a graph data structure strategy - big graph

6.3 Most common dependency depths for Maven packages

From the dependency depth simulation it was discovered that the vast majority of all packages in Maven projects scanned by Debricked were on depth 1 and 2. Direct dependencies and dependencies on depth 3 were the second most common and the deepest dependencies found were on depth 7 which were also the least common. The complete bar graph is found in Figure 6.4.

6.4 Package version count in the Maven registry

According to the results from scraping the Maven central repository it contained 394,095 unique packages which together contained a total of 6,624,270 versions. By dividing the two, we get a mean value of ≈ 16.81 versions per package. The amount of versions released for a given package varied from 1 to 4609.

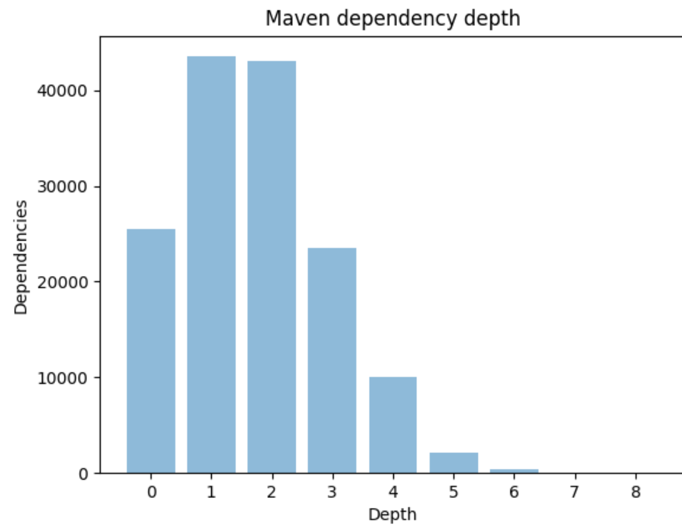

Figure 6.4: Maven dependency depths

Chapter 7

Analysis

7.1 Remediation using a brute force strategy

7.1.1 Dependency depths

When observing the results from the brute force strategy simulations, it is found that the time spent on remediating increases from one depth to the next with an average of 0,99 seconds, with the lowest value being 0,58 seconds and the highest 1,52 seconds. This gives an idea of the time it takes to fetch a dependency from our registry and use it in a build. For depth 0, it only has to fetch a single dependency, but it also builds the prerequisites for a skeleton project. By subtracting the mean time it takes to fetch a dependency (0,99 seconds) from the time it took to remediate in depth 0 ($\approx$ 5.254 seconds), it can be assumed that creating the needed parts for a Maven skeleton projects to build, takes around 4,26 seconds. This is already a lot longer than any graph remediation result.

7.1.2 Number of versions

Another aspect to consider is the average number of versions that exists in a given Maven package. It would most likely be very difficult to find an average of versions bumps required to remediate a vulnerability, requiring us to collect data of remediations in our brute force solution over a very long time. The brute force strategy is also already so much slower than the graph database when just requiring a single version bump, leading us to that it was not worth the time. Regardless, looking at the average number of versions published for a given Maven package, the worst case scenario for the brute force strategy would in average be a remediation which required 17 version bumps, drastically increasing the time of remediation. Scraping the Maven registry also revealed that there are package which have up to 4609 versions published. Attempting a brute force remediation on such a package would most likely be an incredibly time consuming task.

7.1.3 Private Maven registry

By creating our own Maven registry on GitHub, we were able to quickly create a chain of dependencies that were relevant for the simulations. It does, however, leave the question of how much it affects the build time of the project. Could it

be that using the Maven Central Repository is more efficient when it comes to fetching packages, thereby making the brute force solution faster than it was in our simulations? Since we wanted to isolate depth as a factor when bench-marking the different solutions, we felt that the best way of doing this was by creating our own Maven packages where size or width of each package did not skew the results. The Maven Central Repository has quite a rigorous process required to publish a package, however, making us opt for the simpler solution of just using GitHub to publish them, saving us a lot of time. The fact that Maven allows the use of repositories other than the Maven Central Repository also supports us not feeling forced to use it for the simulations. To be completely sure about the validity of the our own Maven registry it could have been interesting to actually perform the same simulation when the packages were published on Maven Central Repository to exclude the risk of the setup not being representative for the simulation.

7.2 Remediation using a graph data structure strategy

The remediation time for simulations on the minimal graph proved, unlike the brute force strategy, not to be as clearly dependent on the depth of the remediation. The fairly high standard deviation made it difficult to draw any clear connections between depth and time. The simulations did, however, show that a remediation in the minimal graph is a lot more efficient than the brute force strategy. We were also interested in seeing how well the graph solution scaled. Since the Maven Central Repository according to our scraping contained 6,624,270 versions, the graph in its final form would require as many nodes in order to create the full Maven dependency tree for use in production. Due to time constraints and the time it took to create all dependency nodes and relationship, we were unable to test adding 1,000,000 and 10,000,000 dependency nodes to the graph database.

When conducting simulations for the different graph sizes, it was found that increasing the size of the graph had a profound effect on the time it took to find a remediation. This is believed to be both due to the fact that the database is larger, but even more so because we used the same dependency names, but with unique versions, for all dependencies we added. This means that the query found a matching name (but not a matching version) in all dependency chains. This is an unrealistic scenario since a Maven package has on average ≈ 17 published versions, but we decided to do so, since the graph proved to be so much faster in regards to remediation time and we wanted to simulate if it was possible to reach a point where the best case brute force remediation would be better, in regards to time, than the graph solution. This was also useful due to the fact that we did not test adding 1,000,000 and 10,000,000 nodes to the graph.

The simulations used a local instance of a Neo4j database and flask application when querying for root fixes. This might have lessened the time it took to send an API request to our flask application, compared to if we would have actually had it deployed. The decision to not deploy was made due to a lack of time, but it would have been interesting to see how that would have affected the simulations.

7.3 Most common dependency depths for Maven packages

The simulations on dependency depths showed us that depths 1 and 2, followed by 0 and 3 were the most common depths of dependencies, at least in Debricked's database. Therefore these depths are of most interest when deciding which solution is preferred for our use case. This is positive for the brute force strategy, since it proved to be more affected by deeper dependency trees than the graph database. However it still does not change the fact that the graph was more effective in all depths, however. In depth 1, for example, the graph with 100,000 nodes took 0.14 seconds to find a remediation in average, while the brute force remediation took 6.30 seconds.

7.4 Other observations

7.4.1 Remediation of direct dependencies

When it comes to the results for depth 0 in the graph simulation, it turned out not to be as interesting as one might think. The reason for why it is so high compared to the other bars and that it keeps growing when adding more nodes, is because our graph queries are not designed to find a vulnerable direct dependency. It first attempts to find a dependency with the name *mvn1* that has a transitive dependency with the name *mvn1* that is not within the vulnerable interval, but does not find any. After that it tries the other query and finds all dependencies with the name *mvn1* that does not have a transitive dependency with the name *mvn1*, returning all nodes with the name *mvn1*. The situation is not realistic, since if the direct dependency is vulnerable there is no need to perform a root fix. The reason for this is that you simply need to adjust the version of the root to one that is no longer vulnerable in those cases. The same goes for the brute force solution, but it is still interesting to see how long that took, since it allowed us to find out how long it takes to set up the project.

7.4.2 Dependency width and size

Worth noting for the simulations is that the environment that was created for the simulations was made to be as simple as possible and enable us to isolate dependency depth as a variable. By creating our own Maven registry on GitHub, as mentioned in Section 5.3.1, we were able to create long dependency chains that had a width of 1. It is not, however, uncommon that real life Maven packages depend on several other packages and that each level in a dependency chain has a width larger than 1. This is expected to greatly increase the number of dependencies that has to be fetched in each brute force attempt. Since our graph query moves upwards in the dependency tree in a straight line from a transitive to a direct dependency, it should in theory not affect the graph solution, however, outside of the fact that there would be more nodes in total in the database. There are other variables that may affect the brute force remediation method, such as the size of each package in the dependency tree. In the end, we deemed the depth being the most interesting for our use-case and comparison and felt that since the graph

database proved to be so much faster in all cases in our simulation, impairing the situation for the brute force solution was not of interest when answering our main questions.

Chapter 8
Conclusions

The focus of this Master's project was to find a solution for remediating vulnerabilities in transitive dependencies in open source. The Master's project was conducted at a company called Debricked and the solution was hoped to assist them in the remediation of their customers vulnerabilities. To accomplish that, we devised two different strategies for resolving such vulnerabilities. The first strategy was the brute force strategy. This attempts to upgrade the direct dependency that depended on the vulnerable transitive dependency iteratively in as small version upgrades as possible. The iteration was to be done until a version of the direct dependency was discovered that no longer depended on the transitive dependency which contained the vulnerability. The other strategy was to create a graph database over all available Maven packages and versions. It could then be queried in order to find versions of direct dependencies, which no longer used a particular transitive dependency. For the sake of the Master's project, we decided to create a proof of concept of the graph, which contained the packages and versions we deemed suitable in order to test the strategy.

After implementation, we compared the two solutions in order to find which of them proved more time-efficient. The comparison was made through multiple simulations, which focused on how the depth of a transitive dependency would affect the time it took to find a remediating version. It became clear from the results of the simulations that the graph database was the preferred solution when it comes to time-efficiency in finding a remediating version independent of vulnerability dependency depth. The graph database proved to be a lot less affected by the depth of the vulnerable transitive dependency than the brute force strategy and was overall a lot faster. It was also of importance that the graph solution scaled well, so that it would be usable even when containing all Maven packages in the future. With a total of 100,000 nodes, the graph database was still able to outperform the brute force strategy by a large margin. We believe that the results provided by this Master's thesis will be of great use for Debricked when further developing their solution to the problem.

Chapter 9

Future work

9.1 Develop simpler strategies for remediation

The current brute force remediation strategy will, as shown by the results, be quite time consuming, and scales fairly poorly with larger packages. The fact that Maven does not accept several versions of a certain package, as mentioned in Section 4.1.2, opens up a possibility of being able to set the vulnerable transitive as a direct dependency and force its version to a secure one. This makes the remediation process a lot simpler, but it does not evade the fact that breaking changes may appear and it would be of interest to look into how this solution could mature.

9.2 Scaling the graph database solution for production

Our current implementation of the graph database is made as a proof of concept for comparing between the brute force solution and the graph database. Since the graph database solution has proven to be very effective, it is of great interest to scale the solution so that the graph includes all existing Maven packages. This way, it can be used live in Debricked's remediation tool. It would also be interesting to conduct similar simulations to the ones made in this thesis with the complete graph, since the size of the database should affect query times.

9.3 Keeping the graph representation up to date

When scaling the graph database solution to be used in Debricked's tool, new issues that have not been discussed in this Master's thesis arise. One of them is how to keep the graph representation up to date. The goal of such a database would be to have an accurate representation of all Maven packages so that it can be used for remediation of vulnerabilities. Because of this, the graph needs to contain the latest available information regarding all packages to ensure that users receive up-to-date updates for their packages. This is not an easy feat due to the size of the graph itself and that it needs to be updated regularly and be fairly fast when it does.

9.4 Widening the scope to more languages

Implementing a solution for Maven was of high priority to Debricked, but looking forward, there could be a lot of value in making similar implementations and conduct the same kind of tests with graph data from other package managers, such as npm, pip, composer and NuGet. Since package managers often differ a lot in terms of how both packages and versions are handled in a project, it would be interesting to see how it affects the parameters that were discussed in the Master's thesis and if similar results would still be obtained.

9.5 The economic aspect

Even though it was clear that the graph solution outmatched the brute force solution in all simulations in regards to time we have not made any comparisons in regards to cost. Say the graph solution would be a lot more expensive to keep updated and to use than the brute force strategy. An interesting aspect would then be to find out at what point the hefty price would not match the gain in speed for the graph solution.

References

[1] Debricked AB. Debricked. https://debricked.com.

[2] Opensource.com. What is open source? https://opensource.com/resources/what-open-source.

[3] Apache Maven Project. Pom reference. https://maven.apache.org/index.html.

[4] Apache Maven Project. Introduction to the dependency mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[5] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. A modular package manager architecture. *Information and Software Technology*, 55(2):459 – 474, 2013.

[6] Composer community. Documentation - basic usage. https://getcomposer.org/doc/01-basic-usage.md.

[7] npm community. package-lock.json. https://docs.npmjs.com/cli/v7/configuring-npm/package-lock-json.

[8] Laerte Xavier, Aline Brito, Andre Hora, and Marco Tulio Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 138–147, 2017.

[9] Tom Preston-Werner. Semantic versioning 2.0.0. https://semver.org/.

[10] Maven community. Available plugins. https://maven.apache.org/plugins/index.html.

[11] Welcome to Apache Maven. package-lock.json. https://maven.apache.org/.

[12] Maven repositories. https://maven.apache.org/guides/introduction/introduction-to-repositories.html.

[13] Maven central repository. https://repo1.maven.org/maven2/.

[14] Oracle. Setting the class path. https://docs.oracle.com/javase/7/docs/technotes/tools/windows/classpath.html.

[15] NIST. General information. https://nvd.nist.gov/general.

[16] MITRE. Cve requirements. https://cve.mitre.org/cve/cna/rules.html#section_8-1_cve_entry_information_requirements.

[17] NIST. Official common platform enumeration (cpe) dictionary. https://csrc.nist.gov/projects/security-content-automation-protocol/specifications/cpe.

[18] NIST. Official common platform enumeration (cpe) dictionary. https://nvd.nist.gov/products/cpe#:~:text=CPE%20is%20a%20structured%20naming,and%20tests%20to%20a%20name.

[19] NIST. Cve-2018-16487 detail. https://nvd.nist.gov/vuln/detail/cve-2018-16487#vulnConfigurationsArea.

[20] Surajit Medhi and Hemanta K. Baruah. Relational database and graph database: A comparative analysis. volume 5, pages 21–25, 2017.

[21] George F. Hurlburt, George K. Thiruvathukal, and Maria R. Lee. The graph database: Jack of all trades or just not sql? *IT Professional*, 19(6):21–25, 2017.

[22] Petr Filip and Lukáš Čegan. Comparison of mysql and mongodb with focus on performance. In *2020 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, pages 184–187, 2020.

[23] H.R. Vyawahare, P.P. Karde, and V.M. Thakare. A hybrid database approach using graph and relational database. In *2018 International Conference on Research in Intelligent and Computing in Engineering (RICE)*, pages 1–4, 2018.

[24] I. Fosić and K. Šolić. Graph database approach for data storing, presentation and manipulation. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1548–1552, 2019.

[25] The PHP Development Team. Php: Hypertext preprocessor. https://www.php.net.

[26] Oracle Corporation. Mysql. https://www.mysql.com.

[27] Python Software Foundation. Welcome to python. https://www.python.org.

[28] Flask. Flask foreword. https://flask.palletsprojects.com/en/2.0.x/foreword.

[29] Redis labs. Redis. https://redis.io.

[30] Inc. Docker. Docker. https://www.docker.com.

[31] Inc. Docker. Docker compose. https://docs.docker.com/compose.

[32] The Matplotlib development team. Matplotlib. https://matplotlib.org/.

Appendix A

Figures

```
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          48 bits physical, 48 bits
     virtual
  Byte Order:             Little Endian
CPU(s):                   16
  On-line CPU(s) list:    0-15
Vendor ID:                AuthenticAMD
  Model name:             AMD Ryzen 9 5900HS with Radeon
     Graphics
    CPU family:           25
    Model:                80
    Thread(s) per core:   2
    Core(s) per socket:   8
    Socket(s):            1
    Stepping:             0
    Frequency boost:      enabled
    CPU max MHz:          3300.0000
    CPU min MHz:          1200.0000
    BogoMIPS:             6590.02
Virtualization features:
  Virtualization:         AMD-V
Caches (sum of all):
  L1d:                    256 KiB (8 instances)
  L1i:                    256 KiB (8 instances)
  L2:                     4 MiB (8 instances)
  L3:                     16 MiB (1 instance)
```

Figure A.1: CPU

```
Handle 0x0013, DMI type 16, 23 bytes
Physical Memory Array
        Location: System Board Or Motherboard
        Use: System Memory
        Error Correction Type: None
        Maximum Capacity: 32 GB
        Error Information Handle: 0x0012
        Number Of Devices: 2
```

Figure A.2: Memory - Handle 0x0013

```
Handle 0x001A, DMI type 17, 92 bytes
Memory Device
        Array Handle: 0x0013
        Error Information Handle: 0x0019
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 16 GB
        Form Factor: SODIMM
        Set: None
        Locator: DIMM 0
        Bank Locator: P0 CHANNEL A
        Type: DDR4
        Type Detail: Synchronous Unbuffered (
           Unregistered)
        Speed: 3200 MT/s
        Manufacturer: Hynix
        Serial Number: 00000000
        Asset Tag: Not Specified
        Part Number: HMAB2GS6AMR6N-XN
        Rank: 1
        Configured Memory Speed: 3200 MT/s
        Minimum Voltage: 1.2 V
        Maximum Voltage: 1.2 V
        Configured Voltage: 1.2 V
        Memory Technology: DRAM
        Memory Operating Mode Capability: Volatile
           memory
        Firmware Version: Unknown
        Module Manufacturer ID: Bank 1, Hex 0xAD
        Module Product ID: Unknown
        Memory Subsystem Controller Manufacturer ID:
           Unknown
        Memory Subsystem Controller Product ID: Unknown
        Non-Volatile Size: None
        Volatile Size: 16 GB
        Cache Size: None
        Logical Size: None
```

Figure A.3: Memory - Handle 0x001A

```
Handle 0x001D, DMI type 17, 92 bytes
Memory Device
        Array Handle: 0x0013
        Error Information Handle: 0x001C
        Total Width: 64 bits
        Data Width: 64 bits
        Size: 16 GB
        Form Factor: SODIMM
        Set: None
        Locator: DIMM 0
        Bank Locator: P0 CHANNEL B
        Type: DDR4
        Type Detail: Synchronous Unbuffered (
           Unregistered)
        Speed: 3200 MT/s
        Manufacturer: Samsung
        Serial Number: 17F12341
        Asset Tag: Not Specified
        Part Number: M471A2G43AB2-CWE
        Rank: 1
        Configured Memory Speed: 3200 MT/s
        Minimum Voltage: 1.2 V
        Maximum Voltage: 1.2 V
        Configured Voltage: 1.2 V
        Memory Technology: DRAM
        Memory Operating Mode Capability: Volatile
           memory
        Firmware Version: Unknown
        Module Manufacturer ID: Bank 1, Hex 0xCE
        Module Product ID: Unknown
        Memory Subsystem Controller Manufacturer ID:
           Unknown
        Memory Subsystem Controller Product ID: Unknown
        Non-Volatile Size: None
        Volatile Size: 16 GB
        Cache Size: None
        Logical Size: None
```

Figure A.4: Memory - Handle 0x001D

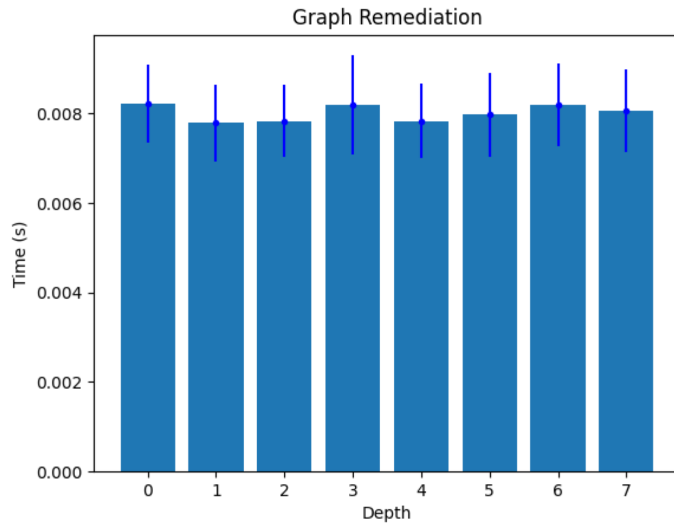

Figure A.5: Remediation using a graph data structure strategy - 10 node graph

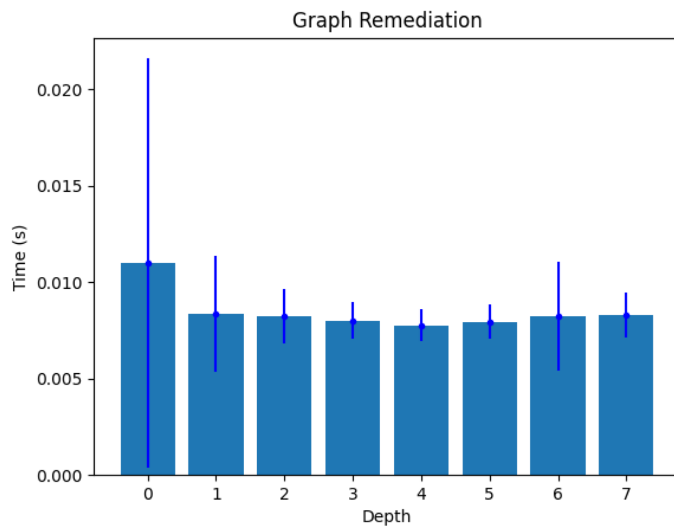

Figure A.6: Remediation using a graph data structure strategy - 100 node graph

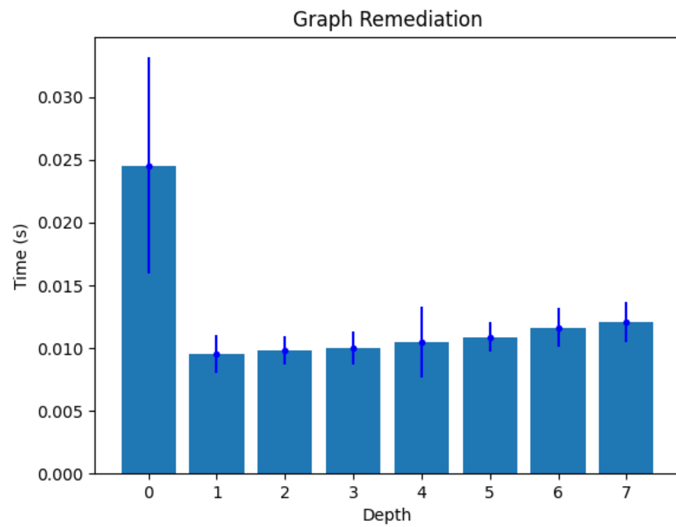

Figure A.7: Remediation using a graph data structure strategy - 1000 node graph

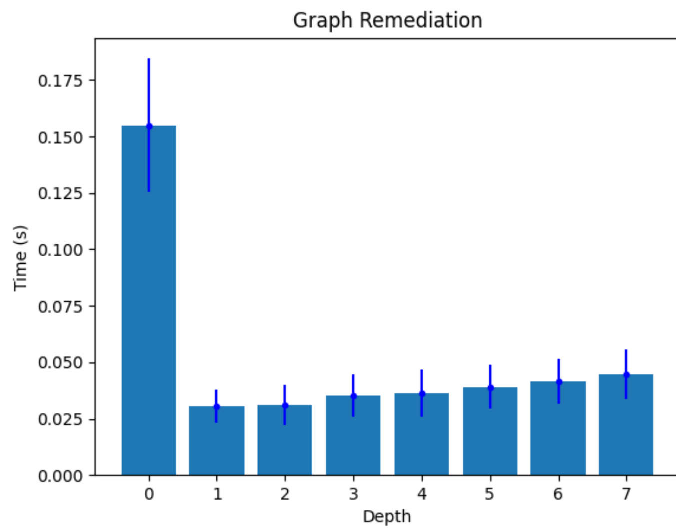

Figure A.8: Remediation using a graph data structure strategy - 10k node graph

Printed by Tryckeriet i E-huset, Lund 2021

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2021-848
http://www.eit.lth.se