NASA/TM—20240006136

Freddie Software Security Patching

Chok Fung Lai
Ames Research Center, Moffett Field, California

June 2024

NASA/TM—20240006136

Freddie Software Security Patching

Chok Fung Lai
Ames Research Center, Moffett Field, California

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, CA 94035-1000

June 2024

Acknowledgments

The author would like to thank Jinn-hwei Cheng and Weston Mathews for setting up the build, scan, and deployment infrastructures, as well as the Freddie Platform Services team members for their coding reviews and technical feedback. In addition, the author would also like to thank Fu-tai Shih, Leonard Bagasol, Dr. Aditya Das, and Dr. William Chan for reviewing this technical memorandum.

Abstract

Software applications become more complicated over time as they depend on many third-party, open-source libraries. The Freddie Platform Services team actively improves software security by addressing software bugs and vulnerabilities that negatively impact software applications, especially those providing real-time operations and services for the federal partners and industries. In order to detect bugs and patch vulnerabilities in software development and maintenance cycles, an automated and systematic approach is needed. This document describes what bugs and vulnerabilities are, and how they can be detected by using static code analyzers and software composition analysis tools. Once vulnerabilities are detected, the patching approaches, such as upgrading direct and transitive dependencies and loading custom classes first, are presented together with their strengths and weaknesses. In addition, patching walkthrough, example code, lessons learned throughout the vulnerability patching process and the recommended practices are discussed.

Practitioner Notes

What is already know about this topic

- Java-based projects leverage third-party, open-source libraries to save development time and effort.
- Software bugs and library vulnerabilities must be detected and addressed throughout the software development and maintenance cycles.
- Though automation tools are used to scan and report software bugs and library vulnerabilities, addressing the issues for development and release branches are time consuming and error prone.

What this document adds

- Lessons learned from the vulnerability patching process.
- Security patching techniques for software development teams to improve software code quality and security.
- A script to help query all the dependency libraries and their versions for projects using Gradle build tool to verify the security patching.

Implications for practice and/or policy

- Avoid using libraries that are constantly having vulnerability issues.
- Enforce coding practices to ease the maintenance effort and improve productivity due to fewer software bugs and conflict-free code merges.
- Libraries with vulnerabilities need to be patched weekly or monthly. Those with critical vulnerabilities need to be patched instantly.

This page intentionally left blank.

Table of Contents

List of Figures

List of Listings

List of Tables

This page intentionally left blank.

1. Introduction

Software developed by National Aeronautics and Space Administration (NASA) must adhere to the NASA Software Engineering Requirements [1] for class E (Design Concept, Research, Technology, and General Purpose Software) and above regarding cybersecurity and code quality. With an increasing complexity of the code base and number of library dependencies, using automation to detect and fix potential software problems is a common approach in the software industry [2, 3, 4]. Like the software industry, multiple air traffic management projects [5, 6, 7] developed at NASA Ames Research Center use Agile software development [8] and leverage automation to improve code quality and security.

Unmanned Aircraft System (UAS) Traffic Management (UTM) has been an ongoing research and development effort among the Federal Aviation System, NASA, other federal partner agencies, and industries [9]. Federated Airspace Management Framework, or *Freddie*, is a NASA implementation for various projects, including the UAS Service Supplier, the Provider of Services to Urban Air Mobility, the Upper E Traffic Management Service Supplier, and other services for research and operations of federated airspace management. The Freddie Platform Services project uses a microservice architecture instead of a monolithic architecture. The project currently consists of twenty-six Java [10] microservice modules and newer modules will be added to support additional capabilities. These modules can be built, deployed, and run separately as needed. Since each microservice is developed using the Spring Framework [11] that provides a rich set of supporting libraries for service-oriented applications, the team focuses on the business-logic implementation and deployment configurations. Freddie's source code is hosted on *GitHub* with organizational access [12]. Freddie software development follows the industry standard by having the main branch for development and release branches for production. Like other Java-based projects, each Freddie microservice uses third-party, open-source libraries to save time and effort. On the other hand, whenever a programming bug is found or a third-party library has a vulnerability, the bug and vulnerability must be addressed in the main and the release branches.

The rest of this document is structured as follows: Section 2 briefly describes what software bugs and vulnerabilities are and how they are detected. The patching approaches used by the Freddie Platform Services team are detailed in Section 3. After presenting the lessons learned in Section 4, concluding remarks are provided in Section 5.

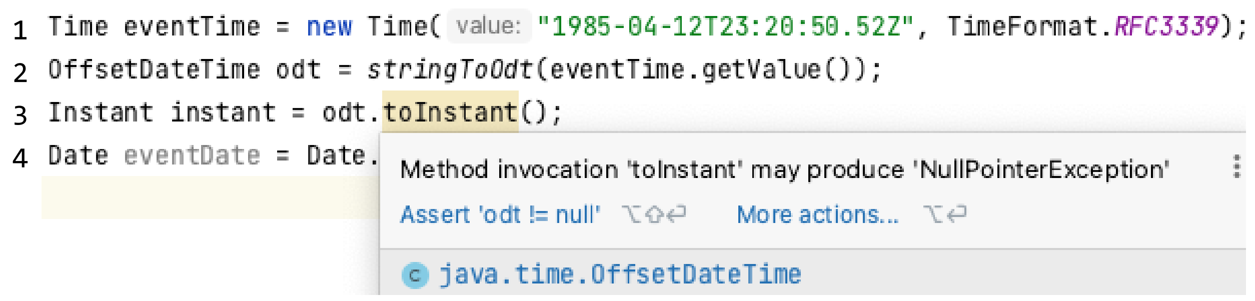
2. Software Bugs and Vulnerabilities

Software bugs and vulnerabilities must be detected and addressed throughout the software development and maintenance cycles. The software security patching, using Freddie project as a use case, is discussed in this document. The Freddie Platform Services team is working on integrating GitHub and two automation tools to streamline the bug detection and vulnerability reporting tasks:

1. *SonarSource's SonarQube* helps organizations detect potential bugs via source code static analysis [13]. The static code analyzers find coding issues based on predefined patterns and rules.
2. *Mend Software Composition Analysis* (SCA) helps organizations find and fix vulnerable open-source dependencies [14]. The SCA tool can also provide recommendations using vulnerability databases.

2.1. Software Bugs

Common programming bugs introduced by software developers can be detected by static code analyzers. Static code analyzers help improve the code quality by finding bugs in the early software development stage, thereby reducing issues at a later development stage as well as after deployment. Integrated Development Environments (IDEs), such as *JetBrains' IntelliJ IDEA* [15] and open-source *Eclipse IDE* [16], provide a direct or a plug-in support of static code analysis to improve code quality even before a project is built. Whenever developers write code that would potentially introduce bugs, the IDEs will not only report the issues in real time but also provide suggested solutions. Figure 2.1 shows a canned example code illustrating a conversion of a Freddie's *Time* instance into a Java's *Date* instance. IntelliJ IDEA detects that the utility method `stringToOdt(String)` (line 2) may return a `null` value, thus, a *Null Pointer Exception* will be produced when calling the statement `odt.toInstant()` (line 3) whenever the variable `odt` points to a `null` value. In addition, bugs can also be reported by SonarQube from a Continuous Integration and Continuous Delivery (CI/CD) pipeline or from nightly builds.



```
1 Time eventTime = new Time( value: "1985-04-12T23:20:50.52Z", TimeFormat.RFC3339);
2 OffsetDateTime odt = stringToOdt(eventTime.getValue());
3 Instant instant = odt.toInstant();
4 Date eventDate = Date.
```

```
Method invocation 'toInstant' may produce 'NullPointerException'
Assert 'odt != null'  ⌥⇧↩      More actions...  ⌥↩
java.time.OffsetDateTime
```

Figure 2.1 A potential Null Pointer Exception bug reported by IntelliJ IDEA

In the software development lifecycle, fixing the source code created by the team requires less effort, especially the static code analyzers usually provide suggested solutions, and the same team has complete access and control of the source code.

2.2. Vulnerabilities

According to the National Institute of Standards and Technology (NIST), a vulnerability is a weakness in the computational logic found in software and hardware components that can result in a negative impact to confidentiality, integrity, or availability [17]. Vulnerabilities introduced by third-party dependencies need to be patched to keep the application secure. Vulnerability patching improves the code quality by addressing vulnerable libraries in the maintenance stage, even though the process requires much effort. The Common Vulnerabilities and Exposures (CVE) Program assigns a unique, alphanumeric identifier called CVE Identifier, or CVE ID, to each vulnerability. CVE IDs have the following format:

```
CVE prefix + Year + Arbitrary Digits
```

where `Year` is the four-digit year when the vulnerability was made public, and `Arbitrary Digits` is a number with four or more digits [18]. NIST maintains a database of known vulnerabilities. For example, the following NIST web page lists the detail of a vulnerability discovered in 2021 that, when certain criteria are met, an attacker can execute arbitrary code via a library called `log4j-core`: https://nvd.nist.gov/vuln/detail/CVE-2021-44228

Projects written in Java programming language leverage artifact and dependency repositories using the *Apache Maven Project* format [19]. Commonly used libraries, especially open source, are included in the Maven repositories. *MvnRepository* [20] is a web site for searching both the Maven repositories and the Java open-source libraries. As of August 1, 2023, there were 1,921 indexed repositories, 34.8 million indexed packages, and 120 terabytes of disk space of the public repositories. Most Maven repositories [20, 21, 22] also list the known library vulnerabilities discovered. For example, the following web page lists all the available versions of the `log4j-core` library among multiple repositories:
https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core

and the library version 2.14.1 has four vulnerabilities. Clicking the version number on the page will redirect the browser to the version-specific view:
`https://mvnrepository.com/artifact/org.apache.logging.log4j/log4j-core/2.14.1`
where the vulnerability `CVE-2021-44228` is listed in the vulnerabilities section.

There are two kinds of library dependencies: direct and transitive. Given three example libraries, A, B, and C, as shown in Figure 2.2(a). Each directed link represents a "depends on" relationship. Library A depends on library B whenever a function defined in A calls one or more functions defined in B. Library B is a *direct* dependency of library A. Similarly, library C is a direct dependency of library B. In this example, library C is a *transitive* dependency of library A. Even though none of the functions defined in library C is used by library A, the omission of C in the application will indirectly impact the functions in A. Thus, a project utilizing library A must include both libraries B and C.

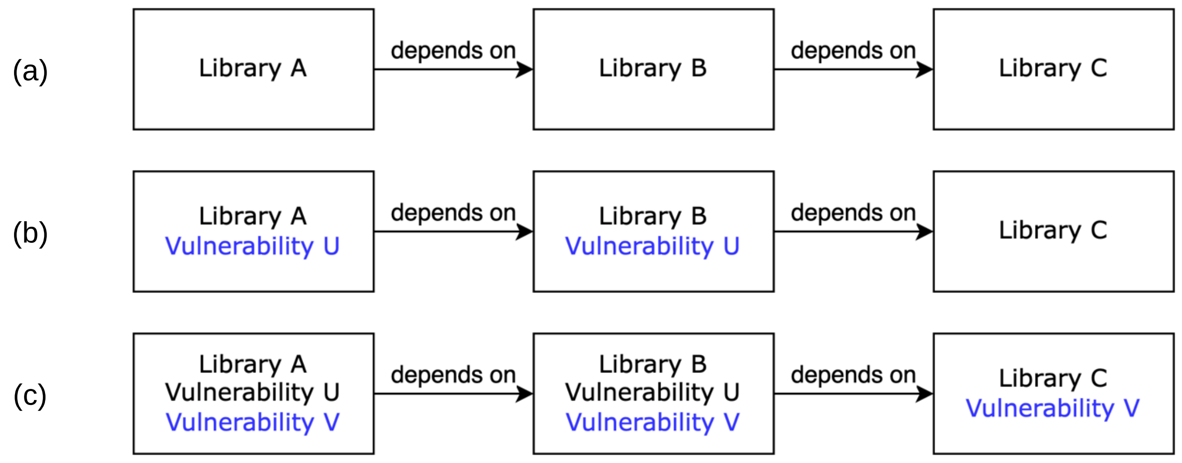


Figure 2.2 Library Dependencies and Vulnerabilities

Vulnerabilities are chained. Figure 2.2(b) depicts that if library B has a vulnerability U, library A will have the vulnerability U too. In addition, Figure 2.2(c) depicts that if library C has a vulnerability V, both libraries A and B will have the vulnerability V as well. Whenever a dependent library has a vulnerability, the vulnerability must be addressed through one of the patching approaches presented in the next section.

3. Patching Approaches

The Freddie Platform Services project uses Gradle Build Tool [23] for configuration, module and library dependency management, as well as source code generation. In Gradle build files, dependency libraries are commonly declared using one of the following two formats:

```
1. implementation group: 'xxx', name: 'yyy', version: 'zzz'
2. implementation 'xxx:yyy:zzz'
```

where `xxx` represents a group name, `yyy` a library name, and `zzz` a version string. The version strings commonly follow the semantic versioning [24]. In this section, a use case is presented to demonstrate the procedure to upgrade a library named `avro`, a data serialization system [25], that contains a vulnerable dependency named `jackson-databind`, a general data-binding package [26]. Gradle has a built-in task to show the dependencies tree of the direct and transitive libraries. Listing 3.1 shows the *Linux*-based shell commands to print out the library dependencies used by the Conformance Monitoring module in the Freddie Platform Services. The `$` symbol indicates the command prompt. Line 1 changes the current working directory to the Conformance Monitoring module. Line 2 executes the Gradle dependencies command.

Listing 3.1 Running Gradle dependencies command

```
$ cd freddie-platform-services/modules/conformance-monitoring/
$ ./gradlew dependencies
```

The output of the Gradle dependencies task is comprehensive and informative. The output produced by the Gradle command has nearly 3,000 lines. Listing 3.2 shows a snippet of the Gradle dependency tree for visualizing the hierarchy of dependencies. The snippet shows that two direct dependency libraries are used when compiling the Conformance Monitoring module: `lambok` version 1.18.24 (line 2) and `avro` version 1.11.0 (line 3). In addition, the `avro` library has four direct dependency libraries, namely, `jackson-core` version 2.12.5 (line 4), `jackson-databind` version 2.12.5 (line 5), `commons-compress` version 1.21 (line 6), and `slf4j-api` version 1.7.32 (line 7). Thus, these four dependency libraries are transitive to the module.

Listing 3.2 Snippet of "`gradlew dependencies`" output

```
1  compileClasspath - Compile classpath for source set 'main'.
2  +--- org.projectlombok:lombok:1.18.24
3  +--- org.apache.avro:avro:1.11.0
4  |    +--- com.fasterxml.jackson.core:jackson-core:2.12.5
5  |    +--- com.fasterxml.jackson.core:jackson-databind:2.12.5
6  |    +--- org.apache.commons:commons-compress:1.21
7  |    \--- org.slf4j:slf4j-api:1.7.32
8  ⋮
```

Vulnerable libraries can be investigated by using the Mend SCA tool (recommended), performing a keyword search in the NIST's National Vulnerability Database[1], or inspecting the "vulnerabilities" column on the MvnRepository website[2]. According to the MvnRepository, the `jackson-databind` version 2.12.5 has the following four direct vulnerabilities:

1. `CVE-2022-42004`
2. `CVE-2022-42003`
3. `CVE-2021-46877`
4. `CVE-2022-36518`

Automatically detecting and reporting the vulnerable libraries is recommended since the scanning process can be a step in the CI/CD pipeline. The software scanning process can also be scheduled during the nightly builds so that developers are notified whenever a vulnerability library is detected. Manual process is useful for understanding the detail of the vulnerabilities, especially when patching the project libraries. The following subsections present three commonly used approaches to address vulnerable libraries including steps, strengths, and weaknesses:

3.1. Upgrading Direct Dependencies
3.2. Upgrading Transitive Dependencies
3.3. Loading Custom Classes First

---

[1] See https://nvd.nist.gov/vuln/search/results?query=jackson-databind
[2] See https://mvnrepository.com/artifact/com.fasterxml.jackson.core/jackson-databind

3.1. Upgrading Direct Dependencies

Whenever a specific version of the library is vulnerable, the library needs to be either upgraded to a stable, i.e., invulnerable, version or replaced with an alternate library. A stable version, if available, can be used by specifying the version in the build file. Listing 3.3 shows a snippet of the dependencies code block of a Gradle build file specifying the `avro` library to use the exact version 1.11.2 (line 3). By specifying the library version, any older `avro` library versions (e.g., 1.11.0) will be promoted to 1.11.2. Similarly, any newer `avro` library versions (e.g., 1.11.3) will be demoted to 1.11.2.

Listing 3.3 Snippet of `build.gradle` file content using upgraded direct library

```
1  dependencies {
2      ...
3      implementation 'org.apache.avro:avro:1.11.2'
4      ...
5  }
```

Listing 3.4 shows a partial output after rerunning the `gradlew dependencies` task. The `avro` library is upgraded from the previous version 1.11.0 to the upgraded version 1.11.2 (line 3). In addition, the upgraded `avro` version contains newer dependency library versions, as indicated by lines 4 to 8: the `jackson`-related libraries are promoted from the version 2.12.5 to 2.14.2 (lines 4 and 5), the `commons-compress` library is promoted from the version 1.21 to 1.22 (line 7), and the `slf4j-api` library is promoted from the version 1.7.32 to 1.7.36 (line 8).

Listing 3.4 Snippet of "`gradlew dependencies`" output after upgrading direct library

```
1  compileClasspath - Compile classpath for source set 'main'.
2  +--- org.projectlombok:lombok:1.18.24
3  +--- org.apache.avro:avro:1.11.2
4  |    +--- com.fasterxml.jackson.core:jackson-core:2.14.2
5  |    +--- com.fasterxml.jackson.core:jackson-databind:2.14.2
6  |    ⋮
7  |    +--- org.apache.commons:commons-compress:1.22
8  |    \--- org.slf4j:slf4j-api:1.7.36
9  ⋮
```

Upgrading the version of a third-party, vulnerable library to a stable one—either recommended by the Mend SCA tool or manually investigated on a Maven repository web site—by specifying the stable version in the dependencies block of the project's Gradle build file is straightforward. The upgrade can be verified by running the `gradlew dependencies` task, which shows the hierarchy of the direct and transitive dependencies. The technique described in this subsection has two limitations: (a) when a stable library is not available, the vulnerability will not be addressed; and (b) when a stable library is not compatible with the other existing libraries used in the project due to build or test failures, further actions described in the following subsections will be needed.

3.2. Upgrading Transitive Dependencies

In case a direct library does not have a stable version, upgrading the library's transitive dependencies can be performed. For example, the version 1.11.2 of the `avro` library was a stable release on Monday, July 3, 2023[3]. Thus, Listing 3.3 will not work prior to this date. To address the vulnerabilities, instead of upgrading the *direct* library version, the *transitive* libraries are upgraded to their stable versions. The vulnerabilities are addressed by defining the transitive libraries with the stable versions in the dependencies code block of the Gradle build file. Put simply, the technique described in Section 3.1 can be applied to upgrade versions of both directed and transitive libraries. Listing 3.5 shows that the two `jackson`-related libraries are upgraded to use the stable version 2.13.5 (lines 3 and 4) from the original version 2.12.5 used by the `avro` library version 1.11.0 (see lines 3-5 in Listing 3.2).

Listing 3.5 Snippet of `build.gradle` file content using upgraded transitive libraries

```
1  dependencies {
2      ...
3      implementation 'com.fasterxml.jackson.core:jackson-core:2.13.5'
4      implementation 'com.fasterxml.jackson.core:jackson-databind:2.13.5'
5      ...
6  }
```

Listing 3.6 shows a snippet of the Gradle dependency tree that the two transitive libraries are upgraded from the previous version 2.12.5 to the upgraded version 2.13.5, as indicated by the lines ending with the "`(*)`" marker on both lines 4 and 5.

Listing 3.6 Snippet of "`gradlew dependencies`" output after upgrading transitive libraries

```
1  compileClasspath - Compile classpath for source set 'main'.
2  +--- org.projectlombok:lombok:1.18.24
3  +--- org.apache.avro:avro:1.11.0
4  |    +--- com.fasterxml.jackson.core:jackson-core:2.12.5 -> 2.13.5 (*)
5  |    +--- com.fasterxml.jackson.core:jackson-databind:2.12.5 -> 2.13.5 (*)
6  |    +--- org.apache.commons:commons-compress:1.21
7  |    \--- org.slf4j:slf4j-api:1.7.32
8  ⋮
```

Since the Freddie Platform Services has twenty-six microservice modules and more modules will be added in the future, verifying that all the libraries are upgraded to stable versions requires time-consuming and error-prone effort. The team developed a shell script file called `list-dependencies.sh` (see Appendix 7.1) to aid the verification task. The script file has been utilized for finding the versions of specific libraries used among the modules. The file output can help developers make sure stable versions are correctly included. To illustrate the usages of the script file, Listing 3.7 captured the commands and outputs the team relied on to check for the versions of the library `jackson-databind` among the direct and transitive dependencies, as well as their upgraded version 2.13.4.2:

1. Line 1 changes the current working directory to the project's `scripts` folder.
2. Line 3 checks out the release branch named `v3.13.x`.
3. Line 7 generates log files for version querying from all the modules.
4. Line 10 lists all the `jackson-databind` library versions using the generated log files.
5. Lines 12 to 28 are the outputs.

[3] See https://mvnrepository.com/artifact/org.apache.avro/avro

Listing 3.7 Running `list-dependencies.sh` script to check for `jackson-databind` versions

```
1  $ cd freddie-platform-services/modules/scripts/
2  
3  $ git checkout v3.13.x
4  Already on 'v3.13.x'
5  Your branch is up to date with 'origin/v3.13.x'.
6  
7  $ ./list-dependencies.sh -f
8  Dependencies log files are generated
9  
10 $ ./list-dependencies.sh jackson-databind
11 Library: jackson-databind
12 com.fasterxml.jackson.core:jackson-databind
13 com.fasterxml.jackson.core:jackson-databind -> 2.13.4.2
14 com.fasterxml.jackson.core:jackson-databind:2.10.5.1 -> 2.13.4.2
15 com.fasterxml.jackson.core:jackson-databind:2.11.0 -> 2.13.4.2
16 com.fasterxml.jackson.core:jackson-databind:2.11.1 -> 2.13.4.2
17 com.fasterxml.jackson.core:jackson-databind:2.12.2 -> 2.13.4.2
18 com.fasterxml.jackson.core:jackson-databind:2.12.5 -> 2.13.4.2
19 com.fasterxml.jackson.core:jackson-databind:2.12.6 -> 2.13.4.2
20 com.fasterxml.jackson.core:jackson-databind:2.12.6.1 -> 2.13.4.2
21 com.fasterxml.jackson.core:jackson-databind:2.13.2 -> 2.13.4.2
22 com.fasterxml.jackson.core:jackson-databind:2.13.2.1 -> 2.13.4.2
23 com.fasterxml.jackson.core:jackson-databind:2.13.3 -> 2.13.4.2
24 com.fasterxml.jackson.core:jackson-databind:2.13.4 -> 2.13.4.2
25 com.fasterxml.jackson.core:jackson-databind:2.13.4 -> 2.13.4.2 (c)
26 com.fasterxml.jackson.core:jackson-databind:2.13.4.2
27 com.fasterxml.jackson.core:jackson-databind:2.6.6 -> 2.13.4.2
28 com.fasterxml.jackson.core:jackson-databind:2.9.6 -> 2.13.4.2
```

The dependencies log files only need to be generated once (line 7) unless code changes have been made after the last file generation. Once the log files are generated, command line 10 can be reused to list the versions of additional libraries. If there is a line in the output section (lines 12 to 28) indicating a wrong version is used, a developer can find the parent libraries in the log file located at `freddie-platform-services/logs/list-dependencies.log` by searching for the specific library and version. The log file contains the output of the `gradlew dependencies` command (see Listing 3.2) from all the microservice modules.

This subsection demonstrated the technique to upgrade the version of a vulnerable transitive library to a stable one by specifying the stable version in the dependencies block. In addition, a script file is created to help find and verify whether all the library versions are upgraded to the stable ones among all the modules. Like Section 3.1, the technique described in this subsection has the same two limitations: (a) a stable transitive library is not available; and (b) a stable transitive library is not compatible with the other existing libraries used in the project due to build or test failures. The limitations can be addressed by applying the technique presented in the following subsection.

3.3. Loading Custom Classes First

As mentioned in the subsections above, addressing a vulnerable library, whether direct or transitive, by upgrading its version to a stable one works if the stable version is available and the library of the stable version is compatible with the existing libraries used in the project. In case a vulnerable library cannot be upgraded directly, it is possible to patch the library by writing custom classes with the fully-qualified class and method names, and making sure the custom classes are loaded before the vulnerable classes by the Java Virtual Machine. This approach requires effort to understand the vulnerable library, its source code, and the detail of the vulnerability.

Listing 3.8 Canned vulnerable method in a third-party library

```
 1  package com.example.library;
 2
 3  public class Calc {
 4    public static float divide(int numerator, int denominator) {
 5      return numerator / denominator;
 6    }
 7  }
```

To illustrate the technique, assume a canned third-party library class, `Calc`, having a vulnerable method named `divide(int numerator, int denominator)` that performs an integer division in the floating-point context (see Listing 3.8). For example, calling `divide(1,2)` returns 0.0 instead of 0.5 because Java returns the integer division rounds towards zero (line 5) [27], even though the method signature indicates that the method returns a floating-point value (line 4).

Listing 3.9 Custom class that fixes the canned vulnerability

```
 1  package com.example.library;
 2
 3  public class Calc {
 4    public static float divide(int numerator, int denominator) {
 5      return (float) numerator / denominator;
 6    }
 7
 8   static {
 9    LoggerFactory.getLogger(Calc.class).warn("Using custom Calc class.");
10   }
11  }
```

A fix for this canned vulnerable method would be modifying the line 5 to ensure the numerator is casted to a *float* data type, as shown in Listing 3.9 (line 5). To ensure the method `divide(int, int)` in the custom class is used, the custom class must be loaded before the vulnerable class in the third-party library. This can be achieved by using the Java's `-classpath` environment variable, as shown in Listing 3.10, where the Java archive (jar) file `calc.jar` contains the custom class `Calc` and the jar file `app.jar` contains the third-party class `Calc`. In addition, a logging statement may be added to the custom class to indicate that the custom class is actually used (see lines 8-10).

Listing 3.10 Loading custom class first with -classpath environment

```
 1  java -classpath calc.jar:app.jar gov.nasa.app.Main
```

As mentioned earlier, Freddie microservices use Spring Framework to ease development effort. The Spring Framework has the `spring-web` library to provide web-related features. The

Mend SCA tool suggested upgrading the library version to 6.0.0 because all the versions 5.3.x are vulnerable[4]. However, the version 6 of the `spring-web` library requires Java Development Kit (JDK) 17 [28] and the Freddie Platform Services project was using JDK 11 at that moment. Thus, upgrading the `spring-web` library requires upgrading the Java version but this was not a feasible option. As a result, the vulnerability in the `spring-web` library was patched by disabling the features via custom classes and such features were not allowed to be used in the code base. To achieve these objectives, the team defined a list of custom classes extending a base class named `Cve_2016_1000027` (see Listing 3.11) to prevent a Java deserialization of untrusted data from calling in the Freddie Platform Services code base. Whenever a custom class was accessed, an *Unsupported Operation Exception* would be thrown from the base class (lines 6 and 7). To further ensure the custom classes were used, a unit test was written to assert that the exception was thrown whenever calling the Spring remoting with Hypertext Transfer Protocol (HTTP) invokers.

Listing 3.11 Custom `spring-web` classes

```
 1  package org.springframework.remoting.httpinvoker;
 2
 3  class Cve_2016_1000027 {
 4    static {
 5      if (true) {
 6        throw new UnsupportedOperationException(
 7          "HTTP Invoker Feature should not be used due to CVE-2016-1000027!");
 8      }
 9    }
10  }
11
12  public class AbstractHttpInvokerRequestExecutor extends Cve_2016_1000027 { }
13  public class HttpComponentsHttpInvokerRequestExecutor extends
14  Cve_2016_1000027 { }
15  public class HttpInvokerClientConfiguration extends Cve_2016_1000027 { }
16  public class HttpInvokerClientInterceptor extends Cve_2016_1000027 { }
17  public class HttpInvokerProxyFactoryBean extends Cve_2016_1000027 { }
18  public class HttpInvokerRequestExecutor extends Cve_2016_1000027 { }
19  public class HttpInvokerServiceExporter extends Cve_2016_1000027 { }
20  public class SimpleHttpInvokerRequestExecutor extends Cve_2016_1000027 { }
21  public class SimpleHttpInvokerServiceExporter extends Cve_2016_1000027 { }
```

This subsection presented a way to address a vulnerable library that does not have a stable or compatible version. Defining custom classes and ensuring these classes are loaded first, so that the vulnerable methods would not be called. This technique has two limitations: (a) developers need to fully understand the feature implementation of the third-party libraries; and (b) the custom classes need to be removed whenever a stable and compatible version is available.

[4] See https://mvnrepository.com/artifact/org.springframework/spring-web

4. Lessons Learned

Throughout the vulnerability patching process, the team realized that certain coding practices would ease the maintenance effort and improve the productivity. The practices described in this section can be used to speed up the security patching and dependency library verification processes. The author assumed that software developers in the industry would have already used similar techniques to help fix the vulnerable libraries.

Each microservice used to have its own Gradle build file, thereby increasing the effort to patch vulnerabilities. In addition, some microservices used to depend on the same library but with different versions, thereby increasing the effort to maintain the code base throughout the software development lifecycle. To make the patching process more effective and efficient, a base module named `freddie-core` is created. The base module depends on the common dependencies used by all the microservices such as Apache Avro and Spring Framework. Common dependencies defined in the base module are removed from the individual microservice modules. As a result, patching the dependencies becomes less error prone as their newer, stable versions can be specified in the base module (a single build file) rather than in all the microservice modules (multiple build files).

When patching vulnerabilities across the main and the release branches, sorting the libraries defined in the Gradle build files by their group names and then package names would make *diff* and *merge*, two commonly used operations, easier among the branches. A diff operation compares changes between two file sets, while a merge operation applies changes from one file set to another file set. A merge conflict occurs whenever a merge operation cannot be performed completely. When a merge conflict happens, developers need to spend time and effort to manually resolve merge conflicts. The manual process not only is error prone but also affects productivity.

Table 4.1 Duration to Patch Vulnerable Libraries

Freddie Version	Supported Branches	Vulnerable Libraries	Supported Branches × Vulnerable Libraries	Duration to Patch Vulnerable Libraries	
				Total	**Average per Library per Branch**
(a)	(b)	(c)	(d) = (b)×(c)	(e)	(f) = (e)÷(d)
3.5.x	1	1	1	3 h 11 m	3 h 11 m
3.7.x A	1	1	1	2 h 17 m	2 h 17 m
3.7.x B	1	1	1	5 h 51 m	5 h 51 m
3.9.x	4	2	8	29 h 25 m	3 h 33 m
3.10.x	2	1	2	26 h 26 m	13 h 13 m
3.11.x A	3	2	6	22 h 52 m	3 h 49 m
3.11.x B	1	3	3	15 h 8 m	5 h 3 m
3.11.x C	**1**	**3**	**3**	**4 h 33 m**	**1 h 11 m**
3.12.x	**1**	**10**	**10**	**8 h 6 m**	**49 m**
3.13.x	**2**	**8**	**16**	**5 h 18 m**	**20 m**
4.0.x	**1**	**13**	**13**	**14 h 53 m**	**1 h 9 m**
4.1.x	**4**	**11**	**44**	**49 h 36 m**	**1 h 8 m**
4.2.x A	**3**	**7**	**21**	**17 h 10 m**	**49 m**
4.2.x B	**3**	**3**	**9**	**19 h 37 m**	**2 h 11 m**

Table 4.1 lists (a) individual Freddie release versions, (b) number of supported release branches, (c) number of vulnerable libraries to be patched, (d) total number of vulnerable libraries among the supported branches, (e) total duration to patch the vulnerable libraries among the supported branches, and (f) average duration to patch a vulnerable library in a branch. The information was obtained by querying Freddie's issue tracking system between the release versions 3.5.x and 4.2.x. Whenever a release version (i.e., 3.7.x, 3.11.x, and 4.2.x) required different patching due to weekly software scans, a letter suffix is appended to each release version to differentiate the patched releases. The patching techniques documented by this technical memorandum were applied since the release version 3.11.x C (the embolden rows). Prior to this release version, on average, it would take about 2,217 minutes ÷ 7 releases = 5 hours 17 minutes to patch a vulnerable library. After applying the patching techniques, the average duration to patch a vulnerable library took 457 minutes ÷ 7 releases = 1 hour 5 minutes. This is a 4 hours 12 minutes (or 79.4%) time saving per vulnerability library patch.

When the vulnerability scanning is performed frequently such as weekly or monthly, most updates involve increasing the library's version patching number. A vulnerability found in an old version of a third-party library may no longer be addressed by the library creators. Solutions, presented in the following subsections, include upgrading the library to a newer version, finding an alternative library with the same or similar functions, or investigating into a workaround.

4.1. Library Upgrade

Upgrading the library will address the issue most of the time. However, the upgrade may not work when a newer version introduces software conflicts, such as requiring a newer Java version to run. For example, the library `spring-web` version 5.3.19 has a vulnerability and its version needs to be upgraded to 6.0.0. However, the version 6.0.0 requires Java 17. In this case, the team created a release plan for the major library or Java version upgrade, meanwhile, applied the other solutions for the current vulnerability.

4.2. Finding Alternate Library

Open-source project communities can come and go due to technology change and volunteer availability. If a library is old or no longer maintained, consider finding an alternative, recent, and active library to ensure any vulnerabilities found in the future will be addressed in a timely manner. However, finding an alternative library may introduce additional workload and sometimes an alternate library is not an option especially when the current library is a part of a famous framework such as the Spring Framework.

4.3. Workaround

When both upgrading and finding an alternate library do not work, the last resort is to find a workaround to ensure the vulnerability will not be accessible to the function callers or application users. Investigate the functions used by the library and determine whether the functions can be reimplemented or not. When reimplemented correctly, the application becomes less vulnerable, especially as most of the unused functions will no longer be included.

5. Concluding Remarks

A software application is vulnerable whenever its direct or transitive dependency library has a vulnerability. NIST maintains a database of known vulnerabilities and the database is utilized and referenced by many Maven repositories. Patching vulnerabilities is important but becomes more difficult, especially when the application over time becomes more complicated and depends on more libraries. Static code analyzers can detect common programming bugs introduced by software developers during software development. Software composition analysis tools help detect vulnerable open-source dependencies and suggest fixes during software maintenance. Automatic software scanning can be a part of the CI/CD pipeline and nightly build processes to ensure any vulnerable library is reported to the developers in a timely manner.

Utilizing commonly used third-party, open-source libraries especially those developed by software communities are recommended so that more resources would be put on maintaining the libraries by the communities, hence reducing our software development and maintenance effort. In addition, the patching effort can be reduced when a project standardizes the library versions used among the modules. Creating a common build file for all the modules will also ease the patching effort.

This document presented three approaches to patch vulnerable libraries in the Freddie Platform Services project: 1) upgrading direct dependencies, 2) upgrading transitive dependencies, and 3) loading custom class first. Strengths and weaknesses of each of these three approaches were discussed. Upgrading direct and transitive dependencies is straightforward but only works if stable and compatible libraries are available; loading custom classes involve much effort that requires a full understanding of the implementation of the vulnerable libraries. Whenever a library upgrade is not possible, consider finding an alternate library or reimplementing the vulnerable functions. Finally, a shell script file is provided to aim the verification of the patched library versions.

The patching techniques can be applied to any Java-based projects, especially using the Gradle build tool. The author hopes that this document will help other software development teams to apply for security patching techniques to improve their software code quality and security in the software development and maintenance lifecycles.

6. References

1. “NASA Procedural Requirements 7150.2D” (Online), https://nodis3.gsfc.nasa.gov/displayDir.cfm?t=NPR&c=7150&s=2B. Retrieved 2024/02/01.
2. Hovemeyer, D., and Pugh, W., “Finding bugs is easy.” Acm sigplan notices 39.12 (2004): 92-106.
3. Morgenthaler, J.D., Gridnev, M., Sauciuc, R., and Bhansali, S., “Searching for build debt: Experiences managing technical debt at Google,” 2012 Third International Workshop on Managing Technical Debt (MTD), Zurich, Switzerland, 2012, pp. 1-6, doi: 10.1109/MTD.2012.6225994.
4. Oyetoyan, T.D., Milosheska, B., Grini, M. and Soares Cruzes, D., “Myths and facts about static application security testing tools: an action research at Telenor digital,” In Agile Processes in Software Engineering and Extreme Programming: 19th International Conference, XP 2018, Porto, Portugal, May 21–25, 2018, Proceedings 19 (pp. 86-103). Springer International Publishing.
5. Robinson, J.E., III, Lee, A., and Lai, C.F., “Development of a High-Fidelity Simulation Environment for Shadow-Mode Assessments of Air Traffic Concepts,” RAeS and AIAA Modelling and Simulation in Air Traffic Management Conference, London, 14-15 November, 2017.
6. Rios, J.L., Smith, I.S., Venkatesen, P., Homola, J.R., Johnson, M.A., and Jung, J., “UAS Service Supplier Specification,” NASA Technical Memorandum, 2020, https://ntrs.nasa.gov/citations/20200000512
7. Verma, S.A, Monheim, S.C., Moolchandani, K.A., Pradeep, P., Cheng, A.W., Thipphavong, D.P., et al., “Lessons Learned: Using UTM Paradigm for Urban Air Mobility Operations,” 2020 AIAA/IEEE 39th Digital Avionics Systems Conference (DASC), San Antonio, TX, USA, 2020, pp. 1-10, doi: 10.1109/DASC50938.2020.9256650.
8. “Agile 101” (Online), https://www.agilealliance.org/agile101/. Retrieved 2023/07/31.

9. “Unmanned Aircraft System Traffic Management (UTM) | Federal Aviation Administration” (Online), https://www.faa.gov/uas/research_development/traffic_management. Retrieved 2023/07/24.
10. “Java SE | Oracle Technology Network | Oracle” (Online), https://www.oracle.com/java/technologies/java-se-glance.html. Retrieved 2023/08/01.
11. “Spring Framework” (Online), https://spring.io/projects/spring-framework. Retrieved 2023/07/21.
12. “NASA GitHub External Low” (Online), https://github.com/NASA-Github-Low. Retrieved 2023/07/20.
13. “Code Quality Tool & Secure Analysis with SonarQube | Sonar” (Online), https://www.sonarsource.com/products/sonarqube/. Retrieved 2023/07/20.
14. “Mend SCA: Open Source Software Management Made Simple” (Online), https://www.mend.io/sca/. Retrieved 2023/07/20.
15. “IntelliJ IDEA – the Leading Java and Kotlin IDE” (Online), https://www.jetbrains.com/idea/. Retrieved 2023/07/24.
16. “Eclipse IDE | The Eclipse Foundation” (Online), https://eclipseide.org/. Retrieved 2023/07/24.
17. “NVD - Vulnerabilities” (Online), https://nvd.nist.gov/vuln. Retrieved 2023/08/01.
18. “cve-website” (Online), https://www.cve.org/About/Process. Retrieved 2023/08/01.
19. “Maven – Welcome to Apache Maven” (Online), https://maven.apache.org/. Retrieved 2023/08/01.
20. “Maven Repository: Search/Browse/Explore” (Online), https://mvnrepository.com/ . Retrieved 2023/07/21.
21. “Maven Central” (Online), https://central.sonatype.com/. Retrieved 2023/08/01.
22. “Open Source Insights” (Online), https://deps.dev/. Retrieved 2023/08/01.
23. “Gradle Build Tool” (Online), https://gradle.org/. Retrieved 2023/07/21.
24. “Semantic Versioning 2.0.0” (Online), https://semver.org/. Retrieved 2024/01/03.
25. “Apache Avro” (Online), https://avro.apache.org/. Retrieved 2023/08/04.
26. “FasterXML/jackson-databind: General data-binding package for Jackson (2.x): works on streaming API (core) implementation(s)” (Online), https://github.com/FasterXML/jackson-databind. Retrieved 2023/08/04.
27. “Java Language Specification Chapter 15.17.2 Division Operator /” (Online), https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html#jls-15.17.2. Retrieved 2024/01/03.
28. “Spring Framework Versions” (Online), https://github.com/spring-projects/spring-framework/wiki/Spring-Framework-Versions. Retrieved 2023/10/06.

7. Appendix

7.1. list-dependencies.sh

The script file lists the dependency libraries and their versions for projects using the Gradle build tool to verify the security patching. The script uses the *Bash* command processor and four commands: `echo`, `grep`, `sed`, and `sort`. The script assumes the Gradle’s Java project directory structure is used, and the “`gradlew dependencies`” task is called to generate dependency graphs.

```
#!/bin/bash
set -e
set -o errexit
set -o pipefail
```

```
usage() {
  echo "This script will list all the dependencies used by models and
modules."
  echo "This script should be run in the freddie-platform-
services/modules/scripts/ directory."
  echo ""
  echo "In order to find dependency library versions, log files need to be
generated once by using -f parameter."
  echo "Subsequent operations do not need -f parameter. Note that if any
library version has changed in any *.gradle"
  echo "file, the log files need to be regenerated."
  echo ""
  echo "Usage: $0 [-f] [-h] [-v version] [lib1 [lib2 ...]]"
  echo "  -f           : Generate dependencies log files"
  echo "  -h           : Print help usage"
  echo "  -v version   : Version string to be appended to the log files;
useful for working on multiple branches"
  echo "  lib1         : First library name, e.g., spring-cloud-function-
context"
  echo "  lib2         : Second library name, e.g., spring-integration-core"
  echo ""
  echo "Examples:"
  echo "  # To generate log files once for the current code base"
  echo "  $0 -f"
  echo ""
  echo "  # To list versions of spring-cloud-function-context library"
  echo "  $0 spring-cloud-function-context"
  echo ""
  exit 1
}

GENERATE_FILES=false
VERSION=

# process arguments
while getopts "fhv:" options; do
  case "${options}" in
    f)
      GENERATE_FILES=true
      ;;
    h)
      usage
      ;;
    v)
      VERSION="-${OPTARG}"
      ;;
    *)
      usage
      ;;
  esac
done
```

```
LOG_DIR=./logs
LOG_FILE=$LOG_DIR/list-dependencies${VERSION}.log
LOG_FILE_SORTED=$LOG_DIR/list-dependencies-sorted${VERSION}.log

if [[ "$GENERATE_FILES" = "true" ]]; then
  mkdir -p $LOG_DIR

  echo "Generating $LOG_FILE ..."
  for file in $(find .. -name build.gradle | sort); do
    dir=$(dirname $file)
    echo $dir
    cd $dir
    ./gradlew dependencies
    cd -
  done > $LOG_FILE

  echo "Sorting $LOG_FILE ..."
  sort -u $LOG_FILE > $LOG_FILE_SORTED

  echo "Dependencies log files are generated"
fi

if [ ! -f "$LOG_FILE_SORTED" ]; then
  echo "File $LOG_FILE_SORTED does not exist!"
  echo "Please run with -f parameter to generate one."
  echo ""
  usage
  echo ""
  exit 1
fi

# shift to the first argument
shift $(expr $OPTIND - 1)

# find library versions
for lib in $*; do
  echo "Library: $lib"

  # Regex explanations:
  # 1. Remove the literals (*) and (n) that appear at the end of some lines
  # 2. Remove tree connector symbols something like | , \--- , and +---  at
the beginning of some lines
  # 3. Remove spaces appear in the beginning of some lines
  # 4. Remove spaces appear in the end of some lines
  # 5. Sort the lines with duplicates removed
  #
  #    1            2               3       4                            5
  sed 's/([*n])//g;s/[|\\+]-*//g;s/^ *//;s/ *$//' $LOG_FILE_SORTED | sort -u
| grep "$lib"
```

```
    echo ""
done
```