Estimating the Attack Surface from Residual Vulnerabilities in Open Source Software Supply Chain

Dapeng Yan
Nanjing University of Aeronautics and Astronautics
dapeng.yan@nuaa.edu.cn

Yuqing Niu
Nanjing University of Aeronautics and Astronautics
977012358@qq.com

Kui Liu
Nanjing University of Aeronautics and Astronautics
kui.liu@nuaa.edu.cn

Zhe Liu
Nanjing University of Aeronautics and Astronautics
zhe.liu@nuaa.edu.cn

Zhiming Liu
Southwest University
zliu@nwpu.edu.cn

Tegawendé F. Bissyandé
SnT, University of Luxembourg
tegawende.bissyande@uni.lu

***Abstract*—Software supply chain security has now become a critical concern in the software industry (and beyond) following the large impact of recent attacks: hackers injected malicious code into Solarwinds components and Octopus scanner, which eventually infected a wide range of downstream dependencies, affecting a massive number of users. Since supply chain vulnerabilities are a well-known concern, especially with open source systems, approaches in the literature mainly focus on identifying and patching such vulnerability. Frequently, however, a vulnerability patch is not immediately propagated to earlier releases that have been inherited by dependents, leaving residual vulnerabilities in supply chains. Our work addresses this challenge and develops a simple approach to iteratively explore the attack surface of supply chain residual vulnerabilities in open source projects. We have assessed our search scheme on 50 GitHub-hosted projects having high stars and forks: we mine their bug fix commits and identify buggy package versions to track the affected dependents and estimate the potential attack surface. We find that many projects fix their vulnerable issues by update their dependency versions, and version inheritance is a significant cause of supply chain attacks for open source projects.**

***Index Terms*—Software supply chain, Vulnerability, Empirical assessment.**

I. Introduction

Software engineering is now being increasingly challenged by the need to meet time-to-market requirements. The building and delivery of software thus required industrialization of processes, notably in its supply chain. The Software supply chain was first introduced in the literature in the ’90s to describe cooperative relationships in software development [1]. As Kaczorowski[1] recalls in the GitHub security series, a supply chain is anything that the developer needs to deliver a product to users—including all the components the developer uses. In practice, this includes “everything that touches the code from development, through the CI/CD pipeline, until it gets deployed into production”. Given this definition, the software supply chain has become critical from a security standpoint due to the pervasive integration of software dependencies. The current 2020 state of the Octoverse [2] report by Github underlines that, on average, it’s common for projects to use hundreds of open source dependencies. These dependencies represent functionality whose code is written by third parties. Figure 1 shows an example of the software supply chain, where a given project (middle) relies on *dependencies* upstream (left) and also have *dependents* downstream (right).

While this software reuse scheme has been publicized in the open-source community [3], recent data unveil that proprietary software is massively reliant on open source dependencies. For example, the 2020 open source security and risk analysis [4] report by Synopsys even suggests that over 80% of enterprise code-bases contain open source code and packages. Consequently, if one of the dependencies is vulnerable, the delivered software product will likely be susceptible to attacks. The major challenge is that even when developers have made due diligence before integrating a dependency, their code or dependency may change over time to introduce a vulnerability or make them susceptible to a previously unexploitable exposure. Therefore, by abusing developers’ trust in the authenticity and integrity of third-party packages hosted on commonly-used servers (along with the adoption of automated build systems that encourage them to rely on package repositories [5]), attackers could compromise dependent systems. The objective is to tamper with the product of a given vendor and make it available to end-users through trusted channels, e.g., download, refer package or update sites [6], hence framing supply chain attacks. Such attacks are performed by injecting

[*]Kui Liu is the corresponding author.
[1]https://cutt.ly/dkrQnOc

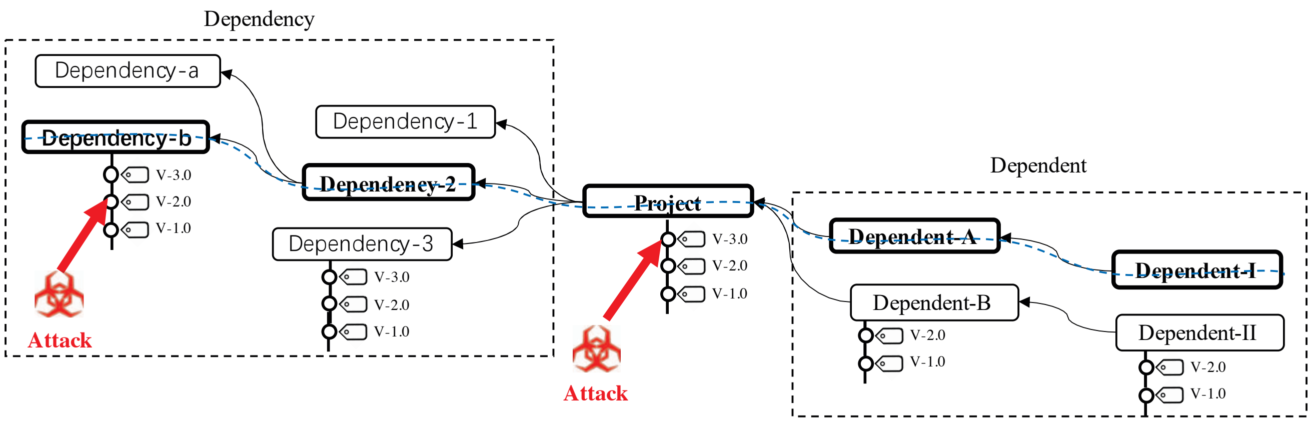

Fig. 1: Supply chain process and its attack.

malicious code into a software product, typically in the form of a vulnerability in the code, a Trojan horse, or a back door. Given the pervasive use of software dependents, supply chain attacks have increasingly become an acute problem in the industry [5], [7]–[16].

Given that open source heavily relies on dependencies to implement software reuse (e.g., libraries), we propose to focus on this ecosystem to check the extent to which even patched vulnerabilities in the dependencies have not adequately propagated to dependents. In particular, we focus on the question of dependency release updates within a project build process. We further shed light on the phenomenon of transitive inheritance where a dependent does not immediately view that its dependency is vulnerable due to its use of other vulnerable dependencies upstream.

In this work, we first empirically overview an example attack on the open-source supply chain. Then we present our methodology for tracking residual vulnerable dependencies used by projects through obsolete releases. Finally, we conduct an empirical study on 50 open source projects to explore to what extent the vulnerable software packages can impact the OSS supply chain.

In summary, this paper makes the following contributions:

- We summarize the key features of commits, such as hunks, modified file types, insertions, and deletions, that help identify projects that are likely affected by supply chain attacks.
- We develop an analysis methodology that explores the inheritance of software dependencies to identify potential supply chain attacks in open source projects.
- We provide a prototype automated tool that implements our analysis. We evaluated it on a sample set of open-source projects. We identified actual dependents and some sub-dependents of selected projects that could be attacked due to their continuous use of vulnerable versions.

II. Octopus Scanner Malware in the Open Source Supply Chain

In this section, to better motivate our work, we detail a concrete example of a supply chain attack.

Octopus Scanner Malware Attack: Figure 2 presents an overview of the Octopus attack process. When the program with the octopus scanner malware is downloaded to the user's terminal, the malware will first identify the NetBeans IDE and enumerate all projects in the NetBeans directory. Then, the disguised malware (i.e., the `ocs.txt` file) drop to the "`octopus.dat`" malicious payload that will be copied to `nbproject/cache.dat`. The `nbproject/build-impl.xml` file will be modified to ensure the malicious payload is executed whenever a NetBeans project is built. If the malicious payload is an instance of the Octopus Scanner itself, the newly built JAR file will be infected as well. Eventually, the infected projects under the NetBeans project will be backdoored by the malware and spread over open source again.

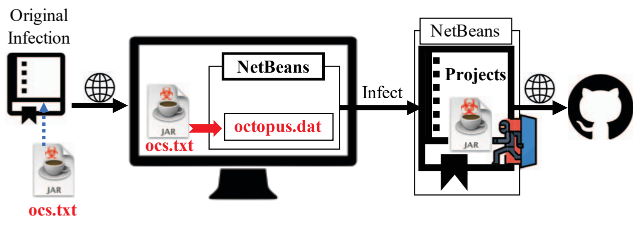

Fig. 2: Process of octopus scanner malware attack.

Backdoored Programs: A security researcher reported to GitHub that some projects (at that time 12 as presented in Table I) were infected by the octopus scanner malware and were available in GitHub [17]. The developers of three projects (i.e., SuperMario-FR, JavaPacman, and pacman-java-ia) replied and resolved the reported octopus malware attack issue. The developers of the remaining nine projects haven't taken action to fix the issue. According to the reply of developers, the fundamental problem is that the developers were utterly unaware that they were committing backdoored code into their repositories [17].

TABLE I: Projects Backdoored by Octopus Scanner Malware.

Project	issue status	Project	issue status
SuperMario-FR	✓	GuessTheAnimal	✗
JavaPacman	✓	SnakeCenterBox4	✗
pacman-java-ia	✓	PacmanGame	✗
ProyectoGerundio	✗	CallCenter	✗
Secuencia-Numerica	✗	Punto-de-venta	✗
Kosim-Framework	✗	V2Mp3Player	✗

*✓ and ✗ mean the issue was resolved and unresolved by developers, respectively.

After looking into those 12 projects (the versions with reported issues), all of them contain a folder "`../nbprojecct/`" that includes two files (i.e, "`cache.dat`" and "`build-imp.xml`"). The octopus scanner malware attack released "`cache.dat`" in the infected programs to hide the backdoor of remote control for the infecting device. The file "`build-imp.xml`" is maliciously added with two new code statements (`<target name = "-pre-jar">` and `<target name = "-post-jar">` to cooperate with "`cache.dat`" to further attack the new projects under NetBeans.

To resolve the attack problem, various ad-hoc solutions have been found by developers. In the case of the `SuperMario-FR-` project, the fix consisted of removing the two files "`cache.dat`" and "`build -imp.xml`", and then rebuilding the new NetBeans environment to remove the infected NetBeans.

To sum up, to fix the whole supply chain attack, according to the studied case, it is necessary to remove the related malicious code and infected dependency files. Unfortunately, it is possible that some of these files are not updated (e.g., with non-infect Jar) in the dependent projects. We refer to them as residual vulnerabilities. We will try to identify them to estimate the attack surface of residual vulnerabilities, even across inherited (also called transitive) dependencies, in the open-source software supply chain.

III. Study Design

In this section, we introduce the research questions for our study. We then present the design details, notably w.r.t. the study subjects and the methodology that we propose to identify vulnerable release versions of open source programs.

A. Research Questions

To strengthen supply chains, it is of high importance to identify and analyze their weak points and identify the potential attack surface a supply chain offers to malicious code injection. Also, taking a lesson from the problems that occurred, we should judge its influence scope to avoid repeating it on different projects in the software supply chain. Overall, our investigation of the vulnerabilities in the open-source software supply chain seeks answers for the following research questions (RQs):

- **RQ1.** *To what extent project dependents are impacted by the vulnerable programs in the open-source software supply chain?*
  In the open-source software supply chain, the vulnerabilities of supplier programs can propagate to dependent programs relying on them. Normally, if the vulnerabilities in supplier programs are fixed, their infected dependents could simultaneously resolve their vulnerabilities. This research question is to estimate to what extent real-world programs remain affected by some residual vulnerabilities in the open-source software supply chain.
- **RQ2.** *In the open-source software supply chain, to what extent the supplier programs are affected by vulnerabilities?*
  In practice, when a vulnerability arises in a project and is identified, the developers will fix it and a related patch will be committed into the code repository. Our research question aims to investigate how many versions of supplier programs in the open-source software supply chain have been infected by vulnerabilities and how long these vulnerabilities lasted. Such duration will serve as an important indicator to measure the risk vulnerabilities and the extent of their impact on the project.
- **RQ3.** *Where are the vulnerabilities located?* Since it is impossible to avoid attacks in the open-source software supply chain, it is critical to enable early detection and rapid fixing of the introduced vulnerability to limit the attacks. Locating vulnerabilities when they are known is the initial step for an analysis of the attack surface and for applying a fix. We propose to explore and deepen the characteristics of vulnerabilities in the supply chain by investigating the location of such vulnerabilities.

B. Subjects

In this work, we select 50 Java open-source projects as subjects which have been shown in Table II to estimate the attack surface from the residual vulnerabilities in the open-source software supply chain. We rely on the Maven dependency management[2] to find the dependents of a supplier program in the open-source software supply chain. Therefore, all our subjects have a '`pom.xml`" file. For selecting the subjects, we first consider the Java open-source projects in two popular organizations, Apache[3] and Google[4], where we randomly select 25 projects. Then, we further randomly select 25 Java open-source projects that have obtained wide attention with at least 1,000 stars on the GitHub platform.

[2]https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html\#Dependency_Management
[3]https://www.apache.org/
[4]https://opensource.google

TABLE II: Selected subjects.

Organization	Project	Organization	Project	Organization	Project
apache	struts	google	jimfs	alluxio	alluxio
apache	tomee	google	gson	eclipse	che
apache	opennlp	google	truth	flyway	flyway
apache	pulsar	google	guava	immutables	immutables
apache	storm	google	error-prone	jdbi	jdbi
apache	drill	google	caliper	graphhopper	graphhopper
apache	kylin	google	compile-testing	liquibase	liquibase
apache	karaf	google	google-java-format	pmd	pmd
apache	archiva	google	closure-templates	rest-assured	rest-assured
apache	druid	google	gwtmockito	mockserver	mock-server
apache	camel	google	re2j	hazelcast	hazelcast
apache	hbase	google	tink	dropwizard	dropwizard
apache	curator	aws	aws-sdk-java-v2	janusgraph	janusgraph
undertow-io	undertow	wildfly	wildfly	jersey	jersey
keycloak	keycloak	checkstyle	checkstyle	hawtio	hawtio
hcoles	pitest	ebean-orm	ebean	graphhopper	jsprit
querydsl	querydsl	openvidu	openvidu		

C. Data Collection

To conduct our experiments, we consider three categories of data: (1) the fix commits for vulnerabilities, (2) the released versions of programs with vulnerabilities, and (3) the dependent programs impacted by the residual vulnerabilities in the supply chain.

a) Collecting fix commits for vulnerabilities: Fix commits are explored to figure out the code fragments and code files related to exposures and to identify the last version of a subject program impacted by the corresponding vulnerability. To collect fix commits, we use the keyword matching method in commit messages that were proposed by Mocks and Votta [18] and have been applied in several studies [19]–[22]. In this study, we consider four keywords (i.e., CVE - The Common Vulnerability Exposures, vulnerability, vulnerable, and backdoor) to search the commits concerning fixing vulnerabilities from the commit histories of the 50 subject projects. In this paper, we summarize the collected data into three categories: **vul** (i.e., "vulnerability" and "vulnerable"), **cve** and **backdoor**.

b) Identifying released versions of programs with vulnerabilities: Identifying the released versions of a program with the related exposure needs to determine the initial commit that inserts the vulnerable code and the fixing commit that resolves the vulnerability. In each collected commit of fixing vulnerabilities, we consider the deleted code (e.g., the code started with the symbol "−" and highlighted in red shown in Figure 3) as the vulnerable code to retrieve the initial commit that adds the vulnerable code. All released versions between the fixing commit (e.g., f-commit shown in Figure 4) and the initial commit (e.g., init-commit in Figure 4) are identified as the versions involving the vulnerability.

c) Identifying the dependents impacted by vulnerable suppliers: For the open-source software, when it is released, it is always made publicly available online (e.g., the popular maven repository[5]), the same for the versions of software released with vulnerabilities. As shown in Figure 5, with the identified versions released with vulnerabilities, we can figure out the dependents in the software supply chain are impacted by the vulnerable versions of suppliers.

Dependents of main subjects can be written in different development languages, and our approach is to match artifact

```
diff --git a/core/src/main/java/org/apache/
    struts2/interceptor/
    StrutsConversionErrorInterceptor.java b/
    core/src/main/java/org/apache/struts2/
    interceptor/
    StrutsConversionErrorInterceptor.java
index 2a19432f0..799848937 100644
--- a/core/src/main/java/org/apache/struts2/
    interceptor/
    StrutsConversionErrorInterceptor.java
+++ b/core/src/main/java/org/apache/struts2/
    interceptor/
    StrutsConversionErrorInterceptor.java
@@ -80,7 +80,7 @@ public class
    StrutsConversionErrorInter- ceptor extends
     ConversionErrorInterceptor
  try {
    stack.push(value);

-   return "'" + stack.findValue("top", String
    .class) + "'";
+   return escape(stack.findString("top"));
  } finally {
    stack.pop();
  }
```

Fig. 3: The patch diff for fixing a vulnerability in `struts`.

[5]https://mvnrepository.com/

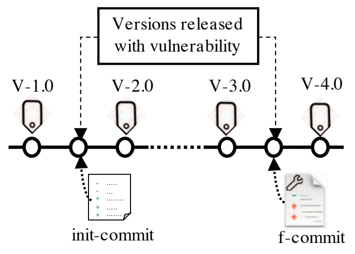

Fig. 4: Identifying the versions released with vulnerabilities.

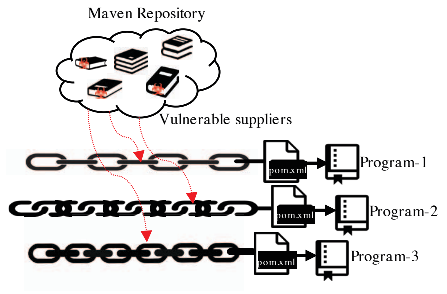

Fig. 5: Identifying the dependents impacted by vulnerable suppliers.

IDs and versions between the subjects and their dependents, so there is no matching data for dependents that are not Maven projects. For each dependent program, with its "`pom.xml`" file, the versions for all of its dependencies in the supply chain can be precisely identified. The dependents impacted by vulnerabilities in the supply chain can be identified by matching the vulnerable versions of suppliers and the interpretations of their dependencies. The replicable package with source code and all collected data for this study are publicly available at:

https://github.com/1993ryan/sc-scan.

IV. Study Results

In this study, we assess the extent of the attack surface related to the residual vulnerabilities of suppliers in the open-source software supply chain. We also investigate the residual vulnerabilities characteristics in 50 open-source Java projects involved in the supply chain as suppliers. The study results answer the research questions (RQs) listed in Section III-A and present vital insights. Note that the commits and dependency relationships are changing day by day. All data used to perform this study are collected from GitHub until 3rd December in 2020.

A. RQ1: Distribution and Location of Vulnerabilities

Our first research question concerns the vulnerabilities that existed in the commit history of suppliers in the open-source software supply chain. We want to understand how the vulnerabilities are distributed and located in the open-source software supply chain suppliers.

Complexity of fixing vulnerabilities: We dissect the vulnerabilities in terms of the modified files, code hunks, and code lines.

As illustrated in Figure 6, the vulnerable code is mainly located in a small number of files as most fixing commits modified a few code files (e.g., 48%=120/250 for single files, 20%=50/250 for two files, and 9.2% for three files). The modified code hunks have a similar distribution as well, shown in Figure 7. However, some vulnerabilities involve multiple code files or multiple code hunks, which could increase the difficulty of detecting and resolving the vulnerabilities in the suppliers.

Figure 8 further illustrates the distribution of vulnerabilities in line granularity. The **cve** vulnerabilities have fewer code lines than the other two kinds of vulnerabilities. The number of **backdoor** vulnerabilities is much fewer than the other two kinds of vulnerabilities, but its deleted and inserted code lines in its fixes are much higher than the fixes of the other two kinds of vulnerabilities. Overall, for fixing most vulnerabilities, they always need to insert more code lines than deleted code lines, and the total number of inserted code lines is almost three times bigger than deleted code lines.

Location of vulnerabilities: We further check the file types associated with the vulnerability fixing commits. Overall, the changes of 250 commits are related to 30 kinds of files (shown in Figures 9) after excluding the descriptive files (e.g., LICENSE).

Figure 9 presents the distribution of the modified file types concerning the referred supplier projects. **vul** type vulnerabilities are retrieved from 38 projects. Thirty-one of them modified the `xml` files in the related fixing commits. For **cve** type vulnerabilities, 23 out 31 projects modified `xml`

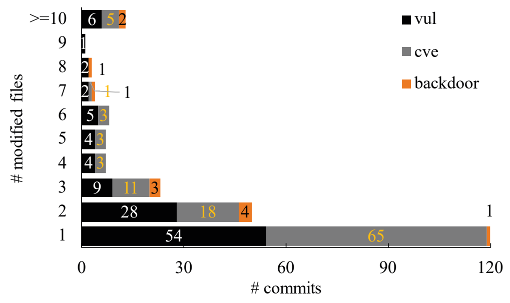

Fig. 6: Distribution of modified files with respect to vulnerability fixing commits.

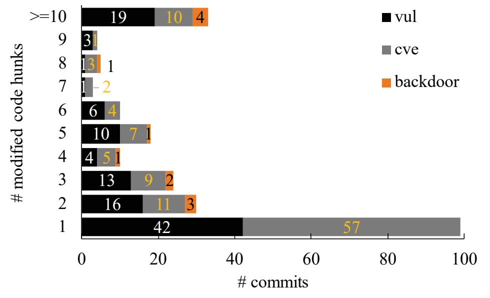

Fig. 7: Distribution of modified code hunks with respect to vulnerability fixing commits.

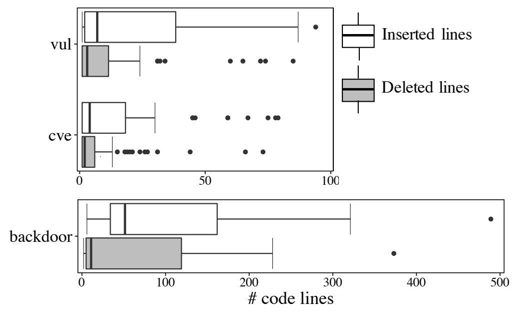

Fig. 8: Distribution of modified code lines with respect to vulnerability fixing commits.

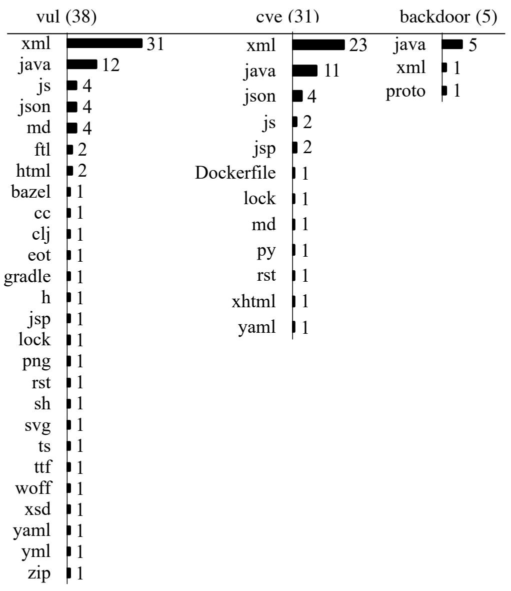

Fig. 9: Referred projects of modified file types.

files for fixing the related vulnerabilities. In the two types of vulnerabilities, `java` code files are the second widely modified files. For fixing **backdoor** vulnerabilities, all of the five projects modified the `java` code files. From the aspect of file types in projects, `xml` and `java` files are the top-2 most frequently modified files for fixing vulnerabilities in the 50 supplier projects. Most of the other file types are specific to different projects.

Looking at the distribution on the modified file types concerning the number of fixing commits and the distribution on the modified files types concerning the number of modified files, `xml` and `java` files are the top-2 most frequently modified files for fixing vulnerabilities in the 50 supplier projects as well. All of the 50 supplier projects are Java projects, and it is customary to modify the `java` code files to fixing the related vulnerabilities.

In general, we totally found 91 commits related to **vulnerability**. Meanwhile, there are 22, 127 and 12 commit records related to **vulnerable**, **cve** and **backdoor**. Statistically, we found that 88%≈ $\frac{80}{91}$ **vulnerability** commits modified `java` or `xml` files. For **vulnerable** and **cve**, it accounts for 86%≈ $\frac{19}{22}$ and 91%≈ $\frac{115}{127}$. Evenmore, 100%≈ $\frac{12}{12}$ related commits of **backdoor** modified these two file types.

Furthermore, We manually checked all the changes in `xml` files and observed that all the related changes are associated with modifying the versions of the dependencies of the suppliers. Moreover, the changes of 11 `json` files are also related to modifying the dependency versions, and the same goes for `md` files. It means that such vulnerabilities are not from the source code of the suppliers but their suppliers. This further highlights that the importance of vulnerabilities in the open-source software supply chain.

B. RQ2: The Extension of Vulnerabilities in the Supplier Lifecycle

Distribution of commits for fixing vulnerabilities: Firstly, Table III presents the number of commits that fix the vulnerabilities that existed in the history of each subject.

TABLE III: Number of commits related to fixing vulnerabilities in the 50 subjects.

Subject	**vul**	**cve**	**backdoor**	Total
struts	12	0	0	12
tomee	2	5	0	6
opennlp	1	0	0	1
pulsar	4	8	0	10
storm	3	0	0	3
drill	2	2	0	3
kylin	6	0	8	14
karaf	2	7	0	8
archiva	1	2	0	3
druid	12	17	0	27
camel	1	14	0	15
hbase	3	5	1	8
curator	0	0	1	1
jimfs	1	0	0	1
gson	0	0	1	1
truth	1	0	0	1
guava	3	1	0	3
error-prone	2	0	0	2
caliper	0	0	0	0
compile-testing	1	1	0	2
google-java-format	0	0	0	0
closure-templates	1	1	0	1
gwtmockito	0	0	0	0
re2j	0	0	0	0
tink	1	1	0	2
alluxio	6	7	0	12
che	3	2	0	5
flyway	4	1	0	4
immutables	1	1	0	2
jdbi	0	3	0	3
graphhopper	0	1	0	1
liquibase	2	3	0	3
pmd	1	4	0	5
rest-assured	3	0	0	3
mockserver	3	2	0	5
hazelcast	0	2	0	2
dropwizard	5	11	0	14
aws-sdk-java-v2	3	2	0	5
undertow	1	1	0	2
wildfly	0	12	0	12
jersey	2	0	0	2
janusgraph	3	4	0	7
keycloak	5	0	0	5
checkstyle	2	2	0	2
hawtio	2	1	0	3
pitest	3	0	0	3
ebean	0	4	1	5
jsprit	0	1	0	1
querydsl	1	0	0	1
openvidu	1	0	0	1
Total	110	128	12	235

In total, 235 (= 110 + 128 + 12 - 15) fixing commits are collected from the 50 subjects. **The 15 reasons for subtracting the total here are that part of the project fixes different bugs with the same commit**, such as `druid` fixed the "**vul**" and "**cve**") issues by the same commit whose ID is "958764908f96bcd1e9b119466a7b2c0f070b3db3". Except for Google projects (i.e., `caliper`, `google-java-format`, `gwtmockito` and `re2j`), each subject contains at least one commit to resolve the related vulnerability described with one of the four keywords (i.e., vulnerability, vulnerable, cve and backdoor). It is expected that the number is high since the vulnerabilities are not widely hidden in programs.

We observe that, in the 50 subjects, most vulnerability-fixing commits are described with two categories ("**vul**" (44%) and "**cve**") (51.2%). Only five projects involve 12 (4.8%) fixing commits concerning the keyword "**backdoor**", and 8 out of them are from the project `kylin`. The project `druid` has the most significant number of fixing commits than other projects. Project `druid` is an Apache project with 406 contributors who offer strong support for this project. This number is higher than the contributors of other projects. ***The more contributors of a project, the higher possibility to resolve the vulnerabilities involved in the program. However, more contributors could increase the risk of injecting malicious code into open-source supplier projects because of their unintentional or bad-intentional operation.***

Impacted time interval of vulnerabilities: Figure 10 presents the distribution of the time interval of each vulnerability that is estimated by computing the time of inserting and fixing the related vulnerable code fragments. Developers fixed only a tiny part of the vulnerabilities in a month. 30% vulnerabilities existed in the supplier projects over one year. eleven vulnerabilities lasted for over five years. The long time interval of fixing the related vulnerabilities reflects that the small number of residual vulnerabilities affect a higher number of dependents.

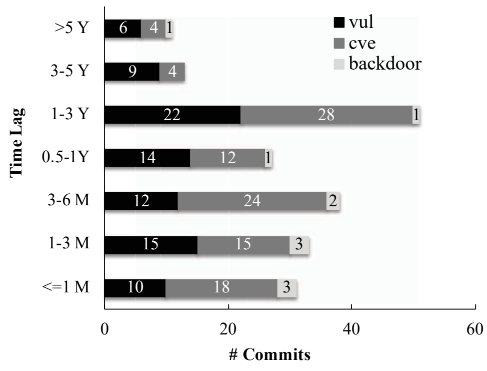

Fig. 10: Projects Distributions Based on Duration Days. • **M** and **Y** represent "month" and "year", respectively.

Affected versions of suppliers: We further assess the released versions of the suppliers with their unfixed vulnerabilities, of which results are illustrated in Table IV. Comparing with the number of vulnerability fixing commits, the number of buggy versions is significantly higher. It seems that **vul** ($10 \approx \frac{1{,}113}{110}$) and **backdoor** ($11 \approx \frac{135}{12}$) vulnerabilities affect more released versions of suppliers than **cve** vulnerabilities ($3.5 \approx \frac{407}{128}$). Anyway, the number of affected versions of supplier projects is much higher than the number of related commits, it indicates that ***it will be a arduous task for maintaining the reliability of the open-source supply chain***.

During our research, we also observed that the vulnerable code from 12 fixing commits does not affect any released version of suppliers as they are fixed quickly after they are inserted into the supplier projects, so there are no dependents affected by this kind of vulnerability. It could benefit from that their contributors immediately resolve the exposed vulnerabilities. ***Fixing the vulnerability in supplier projects before releasing them could effectively impede spreading residual vulnerabilities in the open-source software supply chain.***

TABLE IV: Number of affected versions of suppliers.

Subject	vul	cve	backdoor	Total
struts	26	0	0	26
tomee	5	9	0	14
opennlp	8	0	0	8
pulsar	33	13	0	46
storm	2	0	0	2
drill	4	4	0	8
kylin	53	0	12	65
karaf	3	9	0	12
archiva	20	32	0	52
druid	45	92	0	137
camel	3	8	0	11
hbase	172	7	2	181
curator	0	0	29	29
jimfs	1	0	0	1
gson	0	0	3	3
truth	1	0	0	1
guava	8	5	0	13
error-prone	0	0	0	0
caliper	0	0	0	0
compile-testing	2	3	0	5
google-java-format	0	0	0	0
closure-templates	20	20	0	40
gwtmockito	0	0	0	0
re2j	0	0	0	0
tink	0	0	0	0
alluxio	40	2	0	42
che	69	6	0	75
flyway	45	45	0	90
immutables	3	3	0	6
jdbi	0	4	0	4
graphhopper	0	30	0	30
liquibase	4	7	0	11
pmd	5	37	0	42
rest-assured	39	0	0	39
mockserver	9	2	0	11
hazelcast	0	3	0	3
dropwizard	6	10	0	16
aws-sdk-java-v2	6	2	0	8
undertow	199	2	0	201
wildfly	0	35	0	35
jersey	3	0	0	3
janusgraph	12	6	0	18
keycloak	122	0	0	122
checkstyle	3	3	0	6
hawtio	72	3	0	75
pitest	27	0	0	27
ebean	0	3	89	92
jsprit	0	2	0	2
querydsl	42	0	0	42
openvidu	1	0	0	1
Total	1,113	407	135	1,655

C. RQ3: Attack Surface due to Residual Vulnerabilities

TABLE V: Number of dependents (un)affected by each subject.

Project	# dependents		Project	# dependents	
	all	affected		all	affected
struts	33,418		alluxio	682	
tomee	1,453		che	384	
opennlp	2,865		flyway	5,647	
pulsar	454		immutables	1,651	
storm	5,940	7	jdbi	1,609	
drill	130		graphhopper	525	
kylin	788		liquibase	4,348	
karaf	975	4	pmd	2,517	11
archiva	515		rest-assured	16,585	1
druid	259		mockserver	1,761	
camel	4,066		hazelcast	2,822	
hbase	26,250	2	dropwizard	21,535	14
curator	21,135	2	aws-sdk-java-v2	933	
jimfs	661		undertow	5,006	2
gson	1,049		wildfly	2,062	12
truth	1,777		jersey	5,174	37
guava	17,249	1	janusgraph	786	1
error-prone	2,684		keycloak	5,315	
caliper	1,152		checkstyle	7,551	2
compile-testing	542		hawtio	631	
google-java-format	406		pitest	616	
closure-templates	273		ebean	530	69
gwtmockito	252		jsprit	152	
re2j	237		querydsl	8,126	774
tink	194		openvidu	278	
Total dependents		221,950	Total affected dependents		939

*If a cell is empty, it means none of the dependents still uses the vulnerable of a project.

With the method described in Section III-C, we can identify the dependents of each subject. As shown in Table V, in total, 221,950 dependent projects are leveraging the 50 open source projects, and 939 of them are still using the residual vulnerable versions of 15 projects.

When comparing the number of affected dependents against all dependents, it seems that only a tiny part (0.4% = $\frac{939}{221,950}$) dependents are still affected by the residual vulnerabilities from the suppliers in the open-source software supply chain. Nevertheless, with only 15 supplier projects, 939 dependent projects are infected, multiplying by 62 the initial number of 15 supplier projects. Especially for the supplier project `querydsl`, its residual versions affect 774 dependents, and all of the 774 dependents are impacted by the residual versions of the same program `querydsl-apt` that is a sub-project in the supplier `querydsl`. Most of affected dependents are using version `4.2.1` and `4.1.3`. ***Thus, the security problem from the residual vulnerabilities in the open-source software supply chain can have a broad impact on their dependents, which should be captured attention by the community.***

Table VI details the number of dependents that are affected by each of three vulnerable categories. Majority (88% = $\frac{823}{939}$) of the dependents are affected by the vulnerabilities of **vul** category. Only 11 dependents are impacted by the residual **backdoor**-category vulnerabilities in two suppliers. Surprisingly, most of the dependents are mainly affected by the residual vulnerabilities of three suppliers (`querydsl`, `ebean` and `jersey`). However, the three suppliers do not have the highest number of dependents comparing to other suppliers. In particular, `ebean` only has 530 dependents, but the number of its affected dependents is 60, which accounts for 13% of its dependents, and it is the highest number in column **cve**. Meanwhile, the 774 affected dependents for `querydsl` is the highest number in the table, and it is much larger than the other results in the table. ***To improve the reliability of suppliers in the supply chain, practitioners should pay more effort on the projects of which residual vulnerabilities still have the surface attack on their dependents.***

TABLE VI: Number of dependents affected by residual vulnerabilities.

Subject	vul	cve	backdoor
storm	7	0	0
karaf	0	4	0
guava	1	0	0
janusgraph	1	0	0
checkstyle	1	1	0
pmd	0	11	0
rest-assured	1	0	0
querydsl	774	0	0
hbase	0	2	0
jersey	37	0	0
undertow	1	1	0
curator	0	0	2
wildfly	0	12	0
dropwizard	0	14	0
ebean	0	60	9
Total	823	105	11

We also assess the sub-dependents indirectly affected by the residual vulnerabilities of suppliers. In the beginning, we think that the more profound the project is, the number of sub-dependents will increase exponentially. However, in reality, as the project becomes more and more marginalized, the number of sub-dependents in the deeper layer will decrease to zero. The sub-dependents of a subject denote the projects that reference the dependents affected by the residual vulnerability of the issue. So the sub-dependents are indirectly affected by the subject. The related results are shown Table VII (i.e., # sub-dependents in the third column) As shown in Table VII, 14 out of 939 affected dependents are further referenced by other 399 projects (i.e., sub-dependents). The affected sub-dependents further support that ***the residual vulnerabilities in the open-source software supply chain should be attracted attention from practitioneres***.

We also conducted a simple investigation to check the historical versions of dependents that referenced the vulnerable versions of subjects by checking the related updates in the corresponding "`pom.xml`" file. Checking all dependents' historic commits will take too many resources. Thus, we randomly select 10 of them (one of the ten subjects is failed to retrieve the related commits). As shown in Table VIII, 80 dependents of 9 supplier projects used the vulnerable versions of the suppliers.

In the open-source software supply chain, the residual vulnerabilities of the suppliers are still directly af-

TABLE VII: Number of sub-dependents affected by the dependents of suppliers.

Subject	dependent	# sub-dependents
ebean	actframework/act-ebean	32
	avaje-common/avaje-agentloader	68
	ebean-orm-tools/finder-generator	10
	ebean-orm/ebean-agent	28
	hexagonframework/spring-data-ebean	1
	icode/ameba	10
	mrzhqiang/helper	8
querydsl	SpringCloud/spring-cloud-gray	48
	GUMGA/frameworkbackend	129
	svdev/marketcetera	7
	encircled/Joiner	2
storm	ptgoetz/storm-hbase	16
karaf-cve	apache/brooklyn-server	16
wildfly	wildfly-swarm-archive/wildfly-swarm-spi	26
Total		399

TABLE VIII: Dependents that fixed their vulnerable dependencies.

Subject	# dependents	Subject	# dependents
storm	11	guava	2
struts	1	querydsl	43
ebean	10	dropwizard	4
pmd	2	curator	1
wildfly	6	Total	80

fecting their dependents and indirectly influencing their sub-dependents. The affected dependents are not evenly distributed with suppliers and vulnerability categories. Practitioners should pay more attention to the (residual) vulnerabilities in the open-source software supply chain.

V. Threats to Validity

A threat to validity is the complexity of dependents. Some open-source projects did not maintain their dependency graph, which further leads to the breakpoint of their supply chain. Meanwhile, dependents are developed by different languages, and it is so hard to recognize all developing languages and locate their files used to write their dependency versions. To reduce this threat, we select Java projects only, and the data used for analysis are saved in a local git repository or database. In addition, we only analyze dependency versions of dependents with the pom.xml file to reduce the complexity of dependents' projects.

VI. Related Work

a) **Supply Chain Attack Research:** Gui et al. [23] studied the incident of XcodeGhost development tool infection, The attackers found it difficult in obtaining the official version of Xcode through official channels at that time, and they planted the virus into the Xcode distributed through unofficial channels. In 2017, Cochran analyzed the WireX Android botnet incident, the network propagation process of its token advantage of the software from the software developer to the users [24]. Ransomware is a virus that encrypts user data to hold users to ransom [25], NotPetya ransomware event is that attackers implant the virus in the process of software update phase, and a large number of computers were affected by it [26]. Khandelwal researched the malicious code implanting event of CCleaner, and the attacker tampered with the CRT library (C Runtime Lib) [8] and inserted malicious codes into it. However, these researches are all based on COTs, but the tendency of software development is increasingly towards using open-source software.

Pfretzschner and Othmane proposed a system to identify software supply chain attacks in npm packages by static code analysi [27]. The tool can find four categories of attacks: global variables leakage, management of global variables, operation of a local function, and dependency-tree manipulation, but they failed to recognize real-world instances of these attack types for assessment.

b) **Bug fixed commits Research:** Many researchers have analyzed the commits in software repositories [20], [28]–[30]. Purushothaman and Perry [31] investigated patch-related commits based on the types of repair action and sizes of bug fixed hunks to research the influence of slight source code variations. German [32] studies the features of amendment records, such as modifications of source code in the software version management system, the research analysis it from the following aspects: author, the number of files, and change coupling of files. Yin et al. [33] studied wrong bug-fixes, which are solved by tracking each patch-revision history and found that new bugs could be caused by the bug fixes process. Alali et al. [34] presented research to study the relationships from three aspects (number of files, hunks, and lines) of commits to deduce the features of commits based on a long time historical information.

VII. Conclusion

In summary, malicious code can exist in many kinds of project files. For example, our research finds that `backdoor` mainly existed in the `java` files of projects, and people pay great attention to such issues and often deal with them in a short time. However, sometimes they may be hidden in projects disguised as different file types, so we should also pay attention to the insignificant files, and they may become targets attacks that are difficult to detect.

Meanwhile, we find that projects mainly fixed their vulnerable issues by modifying `java` and `xml` files, and people inserted more codes than deletions to solve the problems. In particular, most files are related to updating their dependency versions, so it is essential to maintain dependency versions to avoid these issues caused by vulnerabilities or construct supply chain attacks for the projects.

Furthermore, the number of affected dependents is further expanded when looking at the next level of dependency, so any issue in one node of the supply chain will impact the chain, and their transmission force is strong. Therefore, people should avoid the risk of supply chain attacks by solving problems as early as possible.

ACKNOWLEDGEMENTS

This research was supported by the National Natural Science Foundation of China (Grant 62172214), the National Key R&D Program of China (No. 2020AAA0107704), the Natural Science Foundation of Jiangsu Province, China (Grant No. BK20210279), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (No. 2020A06) and the Open Project Program of the Key Laboratory of Safety-Critical Software (NUAA), as well as the funding from the European Research Council under the European Union's Horizon 2020 research and innovation programme (No. 949014).

REFERENCES

[1] Jacqueline Holdsworth. *Software Process Design*. McGraw-Hill, Inc., 1995.

[2] Github. The 2020 state of the octoverse. https://octoverse.github.com/, Accessed: 2021-1-28, 2020.

[3] Irina V. Kozlenkova, G. Tomas M. Hult, Donald J. Lund, Jeannette A. Mena, and Pinar Kekec. The role of marketing channels in supply chain management. *Journal of Retailing*, 91(4):586 – 609, 2015. Past, Present, and Future of Marketing Channels.

[4] Synopsys. 2020 open source security and risk analysis (ossra) report. https://www.synopsys.com/software-integrity/resources/analyst-reports/2020-open-source-security-risk-analysis.html, Accessed: 2021-1-28, 2020.

[5] Brian Chess, Fredrick DeQuan Lee, and Jacob West. Attacking the build through cross-build injection: How your build process can open the gates to a trojan horse. *Fortify Software*, pages 24–25, 2007.

[6] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Proceedings of the 17th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 12223 of *Lecture Notes in Computer Science*, pages 23–43. Springer, 2020.

[7] Catalin Cimpanu. Petya ransomware outbreak originated in ukraine via tainted accounting software. https://www.bleepingcomputer.com/news/security, Accessed: 2021-1-20, 2017.

[8] Swati Khandelwal. Ccleaner attack timeline-here's how hackers infected 2.3 million pcs. https://thehackernews.com/2018/04/ccleaner-malware-attack.html, Accessed: 2021-1-20, 2018.

[9] ESET. Winnti supply chain attack. https://securityaffairs.co/wordpress/92508/apt/winnti-supply-chain-attack.html, Accessed: 2021-1-20, 2019.

[10] Alfred Ng. Us: Russia's notpetya the most destructive cyberattack ever. https://www.cnet.com/news/uk-said-russia-is-behind-destructive-2017-cyberattack-in-ukraine/, Accessed: 2021-1-20, 2018.

[11] Elias Levy. Poisoning the software supply chain. *IEEE Security & Privacy*, 1(3):70–73, 2003.

[12] Alex Mullans. Keep all your packages up to date with dependabot. https://github.blog/changelog/2020-06-23-keep-all-your-packages-up-to-date-with-dependabot/, Accessed: 2021-1-20, 2020.

[13] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 181–191. ACM, 2018.

[14] Thomas Hunter II. Compromised npm package: event-stream. https://medium.com/intrinsic/compromised-npm-package-event-stream-d47d08605502, Accessed: 2021-1-20, 2018.

[15] Curtis Franklin Jr. Malware in pypi code shows supply chain risks. https://www.darkreading.com/application-security/malware-in-pypi-code-shows-supply-chain-risks/d/d-id/1335310, Accessed: 2021-1-20, 2019.

[16] Dan Goodin. Supply-chain attack hits rubygems repository with 725 malicious packages. https://arstechnica.com/information-technology/2020/04/725-bitcoin-stealing-apps-snuck-into-ruby-repository/, Accessed: 2021-1-20, 2020.

[17] Alvaro Muñoz. The octopus scanner malware: Attacking the open source supply chain. https://securitylab.github.com/research/octopus-scanner-malware-open-source-supply-chain, Accessed: 2021-1-20, 2020.

[18] Audris Mockus and Lawrence G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the 2000 International Conference on Software Maintenance*, pages 120–130. IEEE, 2000.

[19] Xuan Bach D Le, David Lo, and Claire Le Goues. History driven program repair. In *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 213–224. IEEE, 2016.

[20] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F. Bissyandé, and Yves Le Traon. A closer look at real-world patches. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*, pages 275–286. IEEE Computer Society, 2018.

[21] Kai Pan, Sunghun Kim, and E James Whitehead. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[22] Hao Zhong and Zhendong Su. An empirical study on real bug fixes. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 913–923. IEEE Computer Society, 2015.

[23] Xiaolin Gui, Jun Liu, Mucong Chi, Chenyu Li, and Zhenming Lei. Analysis of malware application based on massive network traffic. *China Communications*, 13(8):209–221, 2016.

[24] J. Cochran. The wirex botnet: how industry collaboration disrupted a ddos attack, 2017.

[25] Sandor Boyson, Thomas Corsi, and Hart Rossman. Building a cyber supply chain assurance reference model. *Science Applications International Corporation*, 2009.

[26] Mike McQuade. The untold story of notpetya, the most devastating cyberattack in history, 2018.

[27] Brian Pfretzschner and Lotfi Ben Othmane. Identification of dependency-based attacks on node.js. In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, pages 68:1–68:6. ACM, 2017.

[28] Li Li, Tegawendé F Bissyandé, Yves Le Traon, and Jacques Klein. Accessing inaccessible android apis: An empirical study. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 411–422. IEEE, 2016.

[29] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. Characterising deprecated android apis. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 254–264, 2018.

[30] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Shin Yoo, and Yves Le Traon. Mining fix patterns for findbugs violations. *IEEE Transactions on Software Engineering*, 47(1):165–188, 2021.

[31] Ranjith Purushothaman and Dewayne E Perry. Toward understanding the rhetoric of small source code changes. *IEEE Transactions on Software Engineering*, 31(6):511–526, 2005.

[32] Daniel M German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[33] Zuoning Yin, Ding Yuan, Yuanyuan Zhou, Shankar Pasupathy, and Lakshmi N. Bairavasundaram. How do fixes become bugs? In *Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and the 13th European Software Engineering Conference*, pages 26–36. ACM, 2011.

[34] Abdulkareem Alali, Huzefa H. Kagdi, and Jonathan I. Maletic. What's a typical commit? A characterization of open source software repositories. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 182–191. IEEE, 2008.