Detecting Unsafe Updates in Software Ecosystems

Yao-Wen Chang
Student ID: 1346258

Supervisor: Christoph Treude
Industry Mentor: Behnaz Hassanshahi (Oracle Labs, Australia)

Word Count: 5410 words

Submission Date: September 17, 2023

Contents

ABSTRACT

The popularity of open-source packages has experienced a significant surge, attracting an ever-growing number of projects that rely on the deployment of third-party artifacts. Simultaneously, the rising prevalence of automatic Continuous Integration/Continuous Delivery (CI/CD) systems has empowered developers to accelerate their project development processes dramatically. However, this convenience, coupled with the continuous addition of new features, has expanded the attack surface, offering malicious actors additional avenues to target downstream users.

This literature review that we are going to discuss in the related work section serves as an introduction to the world of Continuous Integration/Continuous Delivery (CI/CD). It delves into the potential attack surfaces that exist within CI/CD pipelines and explores how malicious attackers exploit these vulnerabilities to compromise these critical systems. Throughout this review, we will introduce various methods and frameworks designed to mitigate these threats. Some of these methods target specific risks at particular stages in the pipeline, while others provide comprehensive coverage across the entire CI/CD process.

In our research, we will contribute to Macaron framework and further to adopt it to analyse the vulnerabilities in the repositories during the software supply chain. This framework is built upon the principles of the Supply Chain Level Security Artifacts (SLSA) framework. We will design our research methodology to involve the retrieval of third-party repositories from GitHub as our primary source of research data, analysis through Macaron, and scanning source code through downstream automation scanner. At the end of the research, we will evaluate the results through the metric we defined in this research proposal.

Our research aims to contributor a framework for scrutinizing artifacts and ensuring their safe passage through the supply chain. We will also provide the scanning results and engage in discussions regarding the implementation of best practices and the development of a secure software supply process.

1 INTRODUCTION

CI/CD is a development method used to efficiently build and test code updates. It helps organizations keep their software consistent and allows them to smoothly incorporate new changes. However, CI/CD systems can be tempting targets for cyberattackers. These attackers may try to insert malicious code into CI/CD processes, steal valuable credentials, or disrupt the original functioning of applications[16].

Recent incidence like the infection of SolarWind's Orion platform [8, 14] which is used to monitor and manage the network is downloaded by thousands of customers, including U.S. government agencies, critical infrastructure providers, and private companies. The malicious actors use supply chain attack to insert malicious code into their system, then SolarWind begin to send out the update with malicious code.

Another malicious attack target the credentials from a contributor of the esline-scope package. The attackers updated the malicious code by merging the code directly into the code base, which will end up stealing multiple credentials from the downstream users [3].

Sometimes, the risks are due to the situation that the consumers are not aware of the importance of frequent update of the dependencies which are consumed in their packages. In this situation, the attackers are able to discover the unsafe dependencies' version on the Git repositories, then looking for other packages depend on these unsafe outdated dependencies. It is simple to find out how the bugs being fixed on the original dependencies' repositories, and directly compromise the downstream users with the bugs. Therefore, a smart alert system [18] to detect if the older version contains bugs and should be updated. Then, the system will automatically update the dependencies. This system can be deployed in different stages to automatically detect unsafe version, and update the dependencies. The developers are sometimes not aware of using the older dependencies will result in tremendous attack. With this automatic tool, it will support developers to solve the issues.

The framework adopted in our research possesses the capability to mitigate the aforementioned exploitation.

In Section 2, we will explore the realm of Software Supply Chain Security, which is the prerequisite for understanding the key components of the Macaron framework implemented in our research. We dissect the Software Supply Chain, emphasizing the critical importance of provenance. Also, we will explain the nuanced difference between Software Bill of Materials (SBOM) and the SLSA Provenance. Finally, we investigate the intricacies of the Build Model and the Security Levels specified by the SLSA framework.

In Section 3, we delve into the related work on the CI/CD pipeline security. Three main attack surfaces within CI/CD pipeline and the counter measures are introduced here. We will introduce the

in-toto framework, and explain why it is related to our framework. These will provide a comprehensive foundation for our research.

In Section 4, Our research questions are defined in this section. The reason behind these questions will be described. Also, we provide the method of data collection and how we are going to evaluate our framework with our defined metric. We delineate our research aims and objectives. Our overarching aim is to fortify the security aspects of CI/CD pipelines. And our objectives include how we are going to achieve our aims. Furthermore, the current research progress and the research gap we find out in related works will be improved by our contribution.

Section 5 provides a detailed timetable and plan for the research project. We outline the key milestones, tasks, and deadlines necessary for successful project completion.

In Section 6, we offer a concise conclusion to the research proposal, summarizing our intentions and expectations.

2 BACKGROUND

SLSA is a security framework provides checklist to ensure the integrity of the supply chain. Artifacts that fulfill SLSA requirements endorsed a traceable source of the software provided by trusted providers. SLSA provides trustworthiness of the artifacts to the developers, downstream users [15]. Customers can examine the dependencies they plan to use in their artifacts through checking all the requirements in SLSA.

2.1 What is Software Supply Chain?

The software supply chain encompasses a complex network of multiple components, including both first-party and third-party libraries, as well as various processes integral to the development, build, testing, and publication of a software artifact. It serves as the backbone for delivering software products to end-users [12].

2.2 Provenance

SLSA provenance clearly provides the transparent information about the artifacts or the packages. Information such as, who builds this artifact and how the artifact was built from the source. The information will be verified by the package registry or even the customers. The provenance is an attestation in SLSA.

2.3 SLSA Provenance versus SBOM (Software Bill of Materials)

Provenance and SBOM are somehow similar, so they are easily confused. Provenance is used to assess the trustworthiness and security of the processes used to build and deliver the software artifacts. An example shows in Listing 1. By contrast, SBOM focus on listing software components and their versions. SBOM format is similar to the Listing 2.

Most software is distributed with pre-compiled package, so inspecting the source code is somehow impossible. SBOM provides a mapping between binary code and the materials of a software. For example, the version of the software will be documented in SBOM. However, the developers cannot trust the pre-compiled artifact is actually derived from the authentic source code. Usually, the current trend will tend to build the reproducible artifact which means the binary code is able to rebuild from the source code if using the given hardware. Then, the developers can verify the integration through comparing the hash between pre-compiled package and rebuilt artifact [4]. SLSA Level 4 does not strongly require the user-provided build script to declare whether the build is reproducible, but it is a best practice.

Listing 1: SLSA Provenance

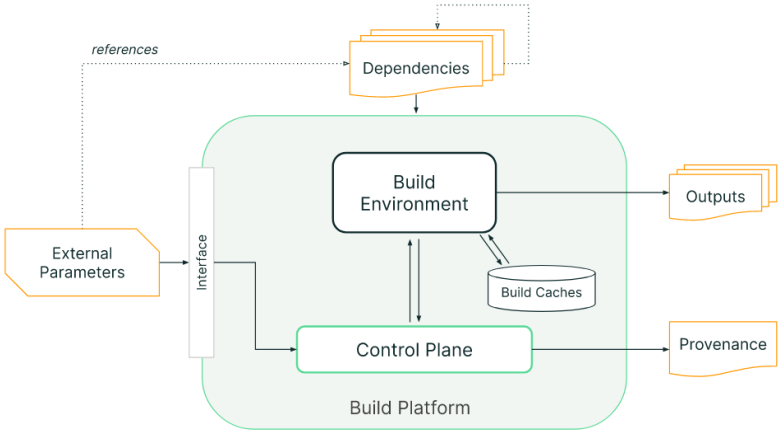
```
[Software Build Provenance]
Build Date: 2023-09-01
Build Environment: Secure, Isolated Environment
Signing Authority: Trusted Certificate Authority (CA)
Signature Verification: Passed
[Supply Chain Processes]
Code Review: Multi-stage code review by security experts
Dependency Scanning: Automated scanning for known vulnerabilities
Build Automation: Continuous Integration/Continuous Deployment (CI/CD) pipeline
```

```
Deployment: Automated deployment to secure servers
[Organizations Involved]
Development Team: Responsible for code development
Security Team: Responsible for security reviews and scanning
Operations Team: Responsible for deployment
```

Listing 2: SBOM

```
[MyApp (v1.0)]
Frontend Framework (v2.3)
Database Connector (v1.1)
Authentication Library (v3.0)
Logging Utility (v1.2)
```

2.4 Build Model



1. The tenant (developers) provide specific external parameters, like the version of the application and the reference to the dependency.
2. The control plane receives the external parameters, then fetch the necessary build scripts, configuration, and dependencies based on the external parameters.
3. The control system sets up the isolated environment for the build.
4. Finally, the model outputs the artifacts. If the build platform follows SLSA build level 2+, then the provenance will also be generated from the control systems.

2.5 SLSA Security Level

Currently, SLSA have defined 4 levels (LV.0 - LV.3), which will be briefly described in Table 1, and working on level 4.

2.6 Policy Engines

Policy is a logic code, which will decide whether the action can be operated. There could be nested policies within the policy. For example, the consumer want to check if they are allowed to access the cloud and download the data from the cloud. The policy will check if their provided password is valid and whether the source IP is not within the banned IP list. Then, the policy to check whether the password is valid. The validity is based on the correct password format and the password is stored in database.

Policy Engine is able to adopt the results generated from other tools, like the checker in the framework of our research or the malicious code injection scanner. The Trusted Wrappers provide and interface to convert the results from the original format into the logic. If the policy is verified, it will become a fact and store in the Transparency log. Everyone can directly monitor the facts store within the

Table 1: SLSA Security Levels

Level	Requirements	Focus
Build L0	None	No security practices are in place.
Build L1	With provenance	Basic security practices are followed, such as code review and basic dependency scanning.
Build L2	Signed provenance, generated by a hosted build platform	More comprehensive security practices are implemented, including in-depth code review, vulnerability scanning, and build verification.
Build L3	Hardened build platform	The highest level of security is maintained, with strict adherence to security practices, automatic testing, and supply chain integrity checks.

transparency log, which prevent the logic from being compromised. The transparency log is untrusted to the consumers, which means the consumers will not trust the maintainers of the log [4].

Our framework implements the policy engines combined with the result from the check in order to provide a clear view and easy way to understand the security requirement of the consumers.

3 RELATED WORK

In the beginning of the literature review, prerequisite definition of the CI/CD technique will be described. Then, a variety of risks and attacks are categorised based on the three attack surface (source code repository, build and test, and deploy) within the CI/CD pipeline. Some defense methods introduced in the related work will be aligned with the attacks in this section.

3.1 Definition of CI/CD

CI/CD is a development and deployment process for automatic building and testing code changes that support organizations maintain a consistent code base, then automatically shipping the artifact to the target customers. 'CI' involves developers frequently merging code changes into a central repository where automatic builds and tests run. 'Build' is the process of converting source code into executable code. Then, running the automatic tests which usually combined unit test and integration test against unexpected results during the build process. These process will avoid integration challenges that can happen when waiting for release day to merge changes into the release branch. 'CD' defined the process of the releases happened automatically [12].

Nowadays, nearly all software applications were built on top of others works which are the third-party packages. However, the convenience and capabilities of the third-party source code usually cause cybersecurity risks [10]. Software supply chain attacks aim at injecting code into software components including source code base and artifacts, like docker image or executable software, to compromise downstream users [8, 5]. Some risks and counter method will be introduced in the following sections.

3.2 Three Main Stage in CI/CD

In the following three subsections, we will categorise some attack events and potential attack methods into the high-level representation of three main categories within the CI/CD pipeline. This categorisation allows us to gain a deeper understanding of the security challenges and risks associated with each phase of the software development and delivery process. There are many strategies provided by the researchers to grapple with the tough issues presented in the software supply chain.

3.2.1 Source Code Repository

Java Virtual Machine (JVM) executes Java bytecode and provides strong safety guarantees. However, the unsafe API, "sun.misc.Unsafe", will cause serious security issue if it is misused by the developers. The research [10] studied a large repository, Maven, and analyzed the compiled Java code. The security issues include violating type safety, crashing the virtual machine (VM), uninitialized objects and so on.

These misuse might impact third-party package management service. Without a manual code review from the maintainer and security inspection through automatic testing on Git repository provider, the misused practice will not be discovered and will end up being merged in the release artifacts. Our **Macaron** framework provide some checks to ensure the artifacts update and release is working through a proper inspection process. The merged code should be reviewed by at least one other authenticated reviewers. Thus, customers can trust the artifact they consume. However, the SLSA checklist does not make sure the reviewers are capable of detecting the unsafe function call.

The research [6] proposed anomaly detection method to cluster and detect the suspicious update JavaScript code. It shows that this unsupervised machine learning clustering method is able to reduce the manual review effort by 89 percent considering the worse case that all updates are not malicious. However, the data is skewed since more unsuspicious updates existed in the dataset. Also, the researchers choose the features based on a static method by reviewing some packages and identifying the malicious code. This method is less cost-efficient and only specific to the programming languages where there are models already trained.

Nevertheless, the author in [2] provide an attack approach to bypass the manual code review. With the Trojan source attacks, the attacker can embed seemingly unsuspicious Unicode into the comments and indirectly modify the order of the code causing unexpected results. Well-designed Unicode payload would bypass the method provided by [6], since this feature is not recognised by the model. The author recommends the developers to include Regex detection method in the 'Build' stage, since this malicious code is triggered by the compiled code. The fundamental way to avoid being compromised by the Trojan source attack is through banning the Unicode when it is not required in the projects. Even though recommend defense strategies are able to counteract this attack, but the method only deal with the specific attack against the CI/CD pipeline. The downstream method followed by **Macaron** is designed based on the Regex concept. Our approach will further diagnosis the health of the code base with the automation scanning tools.

How if the update is stealthy and in the guise of the valid patch, could the manual code review by the professional repository maintainer and the powerful automation tools be capable of filtering out the malicious pull requests? Probably not. According to the research [20], there are multiple immature vulnerabilities existed in the repositories, which are usually dismissed by the maintainers, since they are not harmful to the software execution. However, if the remaining conditions are introduced by the contributors intentionally or unintentionally, the performance of the artifacts will not live up to the maintainers' expectations. The author provides some effective mitigation including advanced static and dynamic analysis of the artifacts or committer liability which is what the SLSA provenance provides. Based on quantifying the catch rate, the author is able to understand how their attack method is possible to bypass the code review of the maintainer. In our research, we will also implement similar method to detect the false positive rate of our detector, the manual code review will be conducted by ourselves.

3.2.2 Build/Test

Malicious code injection can occur during the build of the pre-built components, especially in compiled languages. This attack method is favored by the attackers, since the detection of the pre-built components and compiled code is typically more difficult [9]. Opting for building package directly from source code is a measure to avoid this risk.

In [13], an overview of the attack methods and how the attacks are triggered are being summarised. The most frequent way implemented to inject the bad dependencies is 'Typosquatting'. It is a simple method to modify the packages' name with a similar one. If the consumers did not carefully check the installed packages, the malicious dependencies will being triggered based on the trigger point designed by the attackers. Also, this research find out the popular trigger point of the malicious dependencies is at the installation step. At the installation step, the malicious code is usually embedded in the installation script. When the customers install the packages, the systems are directly being compromised. This research paper implements the taxonomy method to deduce possible trigger points and the possible attacks occurring within each CI/CD stage. This method provides developers a clear and completed checklist to prevent the potential vulnerabilities. However, this paper does not provide the completed counter measure against the attacks. **SLSA** indirectly addresses 'Typosquatting' issue by providing the provenance to check the origin of the artifacts.

3.2.3 Deploy

In the extremely severe attack event [19], the unknown entity gain access to the GitHub repository with the higher permission from the controlled account of the repository maintainer. The accident causes malicious code to be committed and the attacker even trying to remove various repositories. Some good practices are implemented in this repository. For example, the audit logs function provided by GitHub enables the Gentoo organization to respond quickly to prevent from the extra impact. However, the Gentoo GitHub organization does not implement 2FA to prevent the accounts from being fully compromised. Also, it does not manage the identities correctly. **SLSA LEVEL 4** ensure the contributors are authenticated through 2FA, and the pull requests must be reviewed by another authenticated reviewers.

[17] presents a framework to detect the discrepancies of the packages on between source repositories and package managers. This framework can be cooperated with other tools to improve efficient and effective. However, the research does not consider the situation that there are no discrepancies between the source code and the published packages, but the code have already been maliciously injected. In the framework we deploy in this research, the provenance can ensure there are no suspicious contributors contributing to the code base, because the detail of the artifact from building to deployment are all be recorded in the provenance.

3.3 in-toto Framework

in-toto is an end-to-end security framework aim to protect the software supply chain. It provides a variety of mechanism for each steps in the CI/CD to check the integrity of the previous steps. Cryptography and hash function are implemented in in-toto.

Layout is a file format to document the actors, timestamp and actions with each step of the supply chain. It provides the information for downstream steps to verify. This information is the cryptographic hash of action, actors and some other security related data. Therefore, the downstream steps can verify the integrity based on the hash. This is the high level structure compared to Link Metadata.

Link MetaData is a lower level structure compared to **Layout**, which provides much detailed information about each action.

SLSA framework adopted in our research is based on the in-toto framework.

3.4 Conclusion

Despite the previously introduced methods seems to address all the security issues existed in the code base and within the CI/CD, some of them may overemphasize one particular approach to address software supply chain security without considering compounding factors that impact risk. Some projects aim at providing single solution that conflates multiple objectives [11]. For instance, SAP's Risk Explorer is designed to cultivate a community of users and contributors to the taxonomy. Users can explore the taxonomy by collapsing and expanding nodes in an attack tree. Detailed information about attack vectors, references, and safeguards is presented beneath the tree. This tool addresses all potential risks associated with an entire CI/CD pipeline, as referenced in [9].

Similarly, the in-toto framework is dedicated to ensuring the integrity of the supply chain pipeline while harmonizing multiple objectives.

The framework employed in this research is based on Supply Chain Level Security Artifacts (SLSA). In the following section, we will introduce and discuss Supply Chain Level Security Artifacts (SLSA) in more detail.

4 METHODOLOGY

According to previous works from other researchers, most of the works mainly focus on detecting the malicious code within the source code base. Despite this traditional method might be able to discover the suspicious code through scanning, it is an inefficient process if the code base is frequently updated. Also, the method can only promise the malicious code within the whitelist to be detected. However, the attackers usually figure out possible payloads or injection methods to bypass the automatic scanning, like the Trojan source attacks [2] we introduced before. Some works recommend to use machine learning method to detect malicious code as an outlier [6]. How if the attacker try to bypass the training model with obfuscation method? Finding out proper features for training a model is time-consuming. The research [21] discover a method to modify the forward function in deep learning model, which will

indirectly poison the downstream developers who build their model based on the incorrect pretrained weight.

Therefore, we are thinking of whether developing a method to grapple with malicious code injection is possible. If our framework can make sure the contributors are not being compromised and the CI/CD configuration follows the safe requirements suggested by security experts, the developers can trust the dependencies that are used in their project. In order to efficiently check the safety of the artifacts, our research will focus on contributing to the Macaron Framework which is based on SLSA requirements.

Macaron provides trusted artifact information which is collected from the source code build to the deployment stage. However, we are also interested in the malicious contents provided by the attackers. Also, how these updates would affect the artifacts consumers.

Three major questions will be addressed in this research:

RQ1: *What is the scope of the impact of the risks that exist within our target Python and Java repositories?*

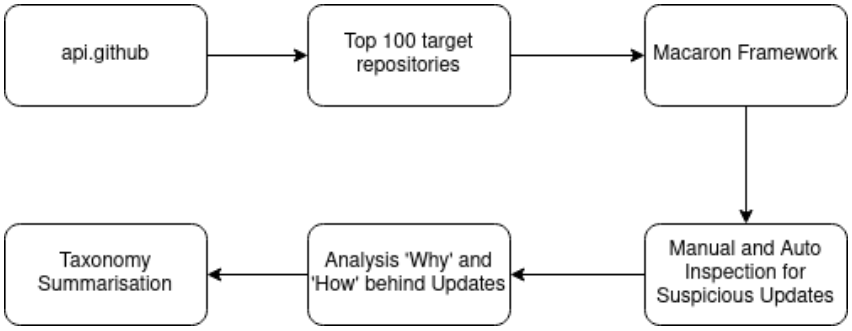
Software supply chain attack is known for its capability to impact downstream customers, so our research will investigate the attack impact scope. Manual code review is required to understand which stage in CI/CD pipeline will be influenced upon the code is triggered intentionally or unintentionally. Finally, the result table will be presented to demonstrate each code injection and its potential victims.

RQ2: *What are the results if these suspicious updates from contributors in open source projects compromise the target?*

From the stakeholders' point of view, the results are of significance, since they care more on whether the attack will result in significant loss. Therefore, in this research, we will analyze the effect when the malicious code is being executed.

RQ3: *To what extent does this work enhance the security of CI/CD pipelines based on the findings and recommendations from our research?*

Scanning the repository is time-consuming, also, precisely pinpointing the malicious code is even harder. We will adopt quantitative measurement to calculate the false positive rate and the time duration. Also, we will assess the severity through CVE severity assessment.



4.1 Data

We use the self-defined script to fetch the URLs of the top 100 Python and Java repositories from GitHub. The URLs will then be fed into the Macaron framework to collect the potential suspicious repositories. We assume that most of the target repositories will not follow the requirements defined in SLSA. Afterwards, we will pass the suspicious pull requests to the automation scanner. Then, we will adopt the automation scanner to scan those repositories.

4.2 Metrics

To quantify the false positive, we decide to use the **precision** concept, which is defined as follows.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Additionally, time efficiency holds great importance in our evaluation. Therefore, we will consider time-consuming, and the severity of the malicious code referring to CVE. Finally, we introduce the security

metric as follows. The metric will be implemented to compare our framework to another static scanners. We will use static scanner to scan all the repositories. On the other hand, we will employ Macaron to filter out repositories deemed unsuspicious, and then subject these selected repositories to the same set of static scanners. In the final analysis, we will evaluate and compare the effectiveness of these two approaches using our defined metrics.

$$SM = (W_p \cdot P) * (W_{tf} \cdot TF) * (\frac{W_{tc}}{TC}) * (W_s \cdot S) \tag{1}$$

Where:

$$\begin{aligned} SM &= \text{Security Scanner Metric} \\ P &= \text{Precision (as a decimal)} \\ W_p &= \text{Weight for Precision} \\ TF &= \text{Total Findings (TP + FP)} \\ W_{tf} &= \text{Weight for Total Findings} \\ TC &= \text{Time Cost} \\ W_{tc} &= \text{Weight for Time Cost} \\ S &= \text{Normalized Severity Score} \\ W_s &= \text{Weight for Severity} \end{aligned}$$

4.3 Research Aims

This research is aim to contribute the Macaron framework, then examine top 100 Python and Java Git repositories with this framework. The statistical findings will be further concluded through the further process. Also, the results will provide the developers and maintainers with a good understanding of the vulnerabilities existed in their CI/CD pipelines configuration. Furthermore, the reason behind these unsafe update will be deeply investigated.

4.4 Research Objectives

1. *First step: Fetch the repositories build from Python and Java with the top 100 most stars.*
2. *Second step: Input the repositories name into the Macaron framework to analysis.*
3. *Third step: Summarize the outputs generated by Macaron and visualize the results with graphs.*

   On this step, we will collect the outputs generated from Macaron and to present statistical frequency of vulnerabilities across various CI/CD stages with graph. Through viewing the graph, it will be more understandable for what is the popular attack surface in the famous repositories.
4. *Fourth step: Manually and automatically inspecting the code base from the potentially problematic repositories due to not comply with the requirements from the SLSA.*

   On this step, we are going to incorporate automation tools, like **Bandit** for Python repositories and **Find Security Bugs** for Java related repositories. Even though automation tools will sometimes generate false positive/negative results, it will save much time for our research. Manual code review on the input passed by **Macaron** framework. As shown in Figure 1, the figure provide the overview of the downstream auto-scanner. To further investigate the scope of the attack caused by the detected code, we will manually inspect to some extent the malicious code impact the down stream consumers. For example, in Python, the **exec** may execute the Python code and directly exfiltrate the user information of the consumers. If the code is injected into the workflow, the build process might be modified.
5. *Fifth step: Document and investigate the reason for this suspicious updates.*

   We will use **taxonomy** approach to classify the aim and possible methods to achieve it. Tree based taxonomy is able to provide a clear view for the developers to understand how to avoid the different attacks.

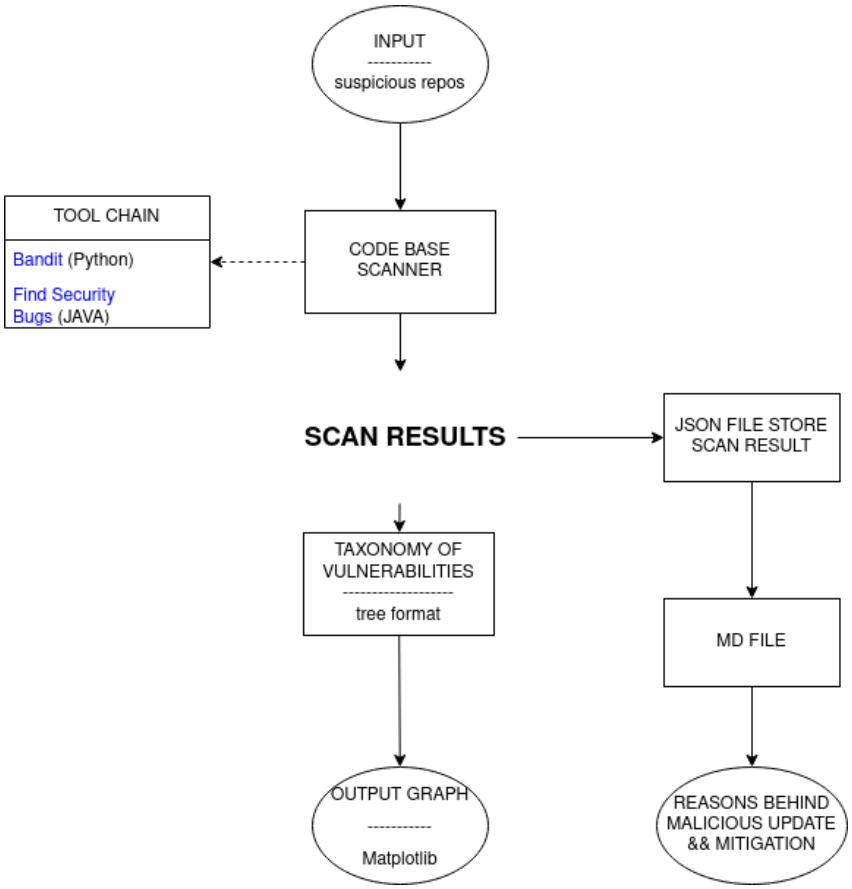


Figure 1: Downstream Scanner

5 TIMETABLE AND PLAN

We split the research plan into three phases, also, providing a timetable for understanding the research schedule.

5.1 Phase One

Defining Unsafe Updates: Building on similar research conducted in the JavaScript ecosystem (https://ieeexplore.ieee.org/document/8805698), the project's first phase involves defining what an unsafe update means within the context of Python and/or Java. Typically, an unsafe update could be one that introduces breaking changes, negatively affects performance, opens up security vulnerabilities, or adds incompatible API changes.

In order to discover different type of unsafe updates and the target victim, we will review related work, articles, and CI/CD attack events to support the understanding of the CI/CD attacks existed nowadays.

5.2 Phase Two

Implementation of Safety Checks: Next, we will extend the Macaron framework's functionality by implementing additional safety checks for these unsafe updates. Macaron is an extensible checker framework for supply chain security and CI/CD services, such as GitHub Actions. It allows adding new checks as Python modules and provides intermediate representations specifically designed for CI/CD services to facilitate verifying new properties. We will begin from implementing SLSA Level 4 check including Two Person Review, Verified History, and Retained indefinitely.

A malicious and suspicious event on GitHub is conducted by one user account "pastramahodu" which forked nearly 2000 repositories. The researcher find out the attacker try to use script to automatically fetch and detect the target file type, such as package.json, sh, and so on. Then, the script creating pull request against each victim repositories to compromise the repositories that are not configured properly. The malicious code injection will be triggered in CI pipelines to exfiltrate repository name and hostname back to webhook server [7]. Therefore, we decide to implement the following three checks to avoid the malicious code being merged into the branch.

The Two Person Review check ensures each pull request is reviewed by at least one authentic reviewer. This check deal with the situation that someone wants to bypass the review and directly merge their code into the branch. Our method will fetch all the pull requests from the branch specified by the users of Macaron framework.

The Verified History ensures at least one strong authenticated participant of the revision's history. The identities should be authenticated through two-step verification. The first step usually verified the password, and second step might be verified through SMS or Email. This check grapple with the situation when the identities' account are being compromised.

Retained indefinitely will check if the commits are preserved for 18 months; therefore, the consumers can trust the artifacts they are going to use in their applications are not being modified by suspicious contributors.

The remaining session, we will keep contributing to the framework, but only implementing a specific version on the phase three.

5.3 Phase Three

Empirical Analysis of Real-World Projects: With the safety checks in place, the final phase of the project is an empirical study conducted on GitHub to ascertain the frequency of unsafe updates occurring in Python and/or Java projects. By understanding the 'how' and 'why' behind these updates, developers can adopt more informed, proactive strategies in their coding practices. We will design a script to fetch our data, which will be designed to perform the Python script Listing 3.

Based on the suspicious commits detected by the Macaron framework, we will further check the code base through static checking tools, then dismissing the false positive case through manual inspection. Considering the Python malicious package, typo-squatting is the most frequent case to compromise the consumers who do not carefully check the packages they download. Usually, the attackers' main goal is using the injected code to connect to their webhook server or build up a reverse shell, which will cause sensitive data exfiltration. And the malicious code are prone to be injected in the setup.py which is

triggered when running the installation script. Therefore, our static analysis method can include this rule [1].

Furthermore, we will build graph and even attack tree to classify our finding and visualize the result. In this way, the developers can easily understand what they should focus and improve in their current configuration of the CI/CD pipeline or the future projects.

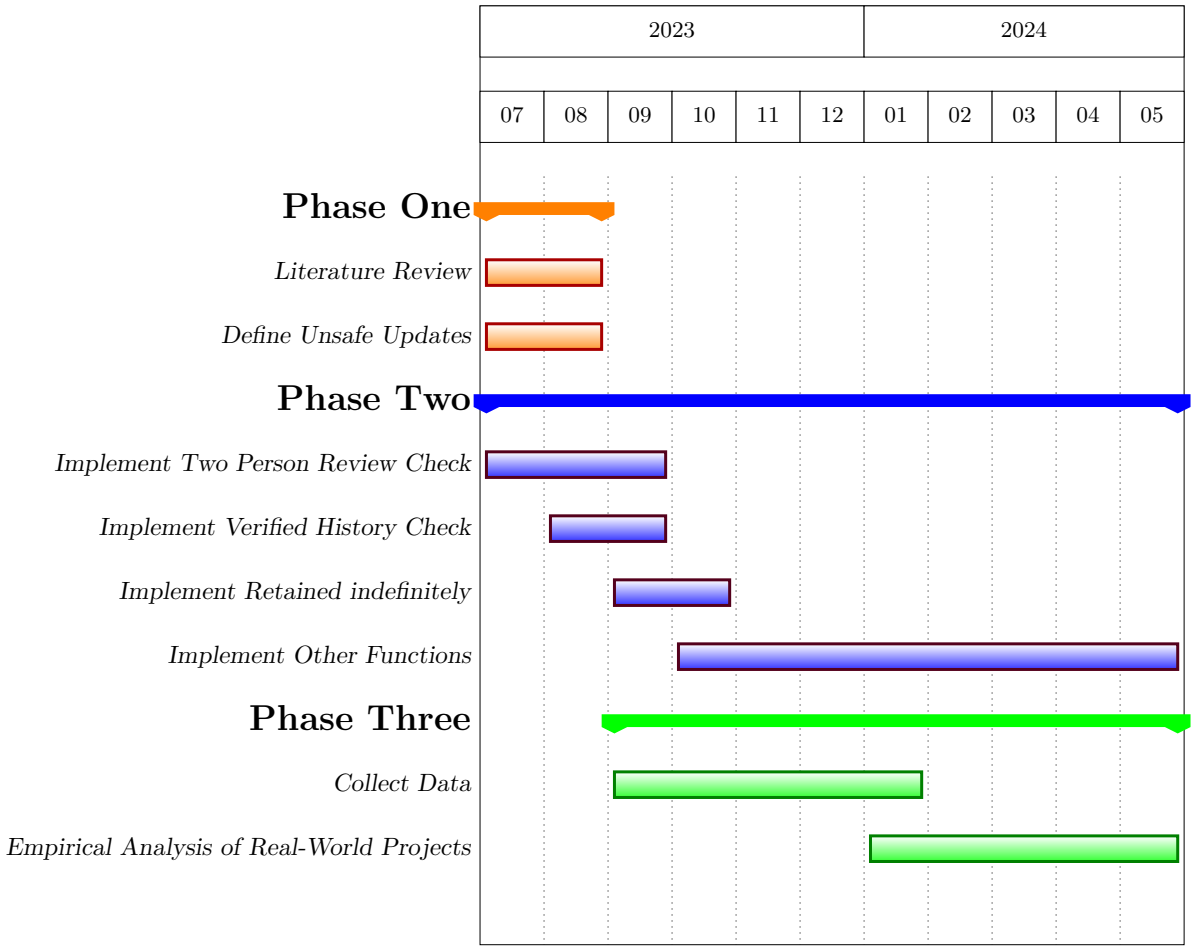
Listing 3: Fetch Top 100 Java and Python Repositories

```
import requests
import os
import sys
access_token = os.getenv("GITHUB_TOKEN")
if access_token is None:
  print(f"Warning: {access_token} environment variable is not set.")
  sys.exit(1)
url = "https://api.github.com/search/repositories"
language = ['java', 'python']
params = {
    'q': f'language:{language}',  # Filter for repositories with more than 0
        stars
    'sort': 'stars',  # Sort by stars
    'order': 'desc',  # Sort in descending order (most stars first)
    'per_page': 100,  # Number of results per page (max is 100)
    'page': 1,  # Page number (start with 1)
}
headers = {
    'Authorization': f'token {access_token}',
}
# Send the GET request to the GitHub API
response = requests.get(url, params=params, headers=headers)

# Check if the request was successful
if response.status_code == 200:
    # Parse the JSON response
    data = response.json()

    # Print information about the top repositories
    for idx, repo in enumerate(data['items'], start=1):
        print(f"{idx}. {repo['name']} - {repo['html_url']} 
            ({repo['stargazers_count']} stars)")
else:
    print(f"Error: {response.status_code} - {response.text}")
```

5.4 Timeline



6 CONCLUSION AND EXPECTED OUTCOMES

6.1 Conclusion

In this research proposal, we analyse different methods developed in the previously related works. And we investigate the pros and cons of these methods in order to figure out the research gap that we can improve in our research. Also, we provide the overview for three main attack surfaces in CI/CD pipeline, then discussing some potential vulnerabilities and counter measure on the security of the software supply chain. And we summarise the Top 10 risks in order to provide a clear view of which risks should be focused more and tackle with first. Also, the **in-toto** framework is introduced since **SLSA Provenance** is based on in-toto format. Since our research tool is developed based on the SLSA, we introduce key components and concepts of this framework as a prerequisite. In the final section, the method and the plan of our research are being discussed. Our method tries to improve the method to detect suspicious updates that is not achieved in previous research.

6.2 Expected Outcomes

The expected outcomes of this research will be contributing some checks properly to the Macaron Framework. Also, finding out vulnerabilities within the popular repositories. Our work expects to find out if the CI/CD pipeline configuration or the protection mechanism is strong enough to avoid unexpected attacks. Taxonomy technique mentioned in the previous section will be implemented in our research to classify the discovered vulnerabilities, and finding out the primary goal of the attack according to these vulnerabilities [13]. Finally, our research will expect to discover the reason behind these malicious updates.

References

[1] Bertus. Detecting cyber attacks in the python package index (pypi), October 13 2018.

[2] Nicholas Boucher and Ross Anderson. Trojan source: Invisible vulnerabilities, 2023.

[3] ESLint. Postmortem for malicious packages published on july 12th, 2018.

[4] Andrew Ferraiuolo, Razieh Behjati, Tiziano Santoro, and Ben Laurie. Policy transparency: Authorization logic meets general transparency to prove software supply chain integrity. In *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, pages 3–13, 2022.

[5] OWASP Foundation. Owasp top 10 ci/cd security risks, 2023.

[6] Kalil Garrett, Gabriel Ferreira, Limin Jia, Joshua Sunshine, and Christian Kästner. Detecting suspicious package updates. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 13–16. IEEE, 2019.

[7] Yehuda Gelb. Mass scanning of popular github repos for ci misconfiguration, May 2023. Published in *checkmarx-security*.

[8] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1509–1526. IEEE, 2023.

[9] Piergiorgio Ladisa, Serena Elisa Ponta, Antonino Sabetta, Matias Martinez, and Olivier Barais. Journey to the center of software supply chain attacks, 2023.

[10] Luis Mastrangelo, Luca Ponzanelli, Andrea Mocci, Michele Lanza, Matthias Hauswirth, and Nathaniel Nystrom. Use at your own risk: The java unsafe api in the wild. *ACM Sigplan Notices*, 50(10):695–710, 2015.

[11] Marcela S. Melara and Mic Bowman. What is software supply chain security?, 2022.

[12] U.S. Department of Defense. Defending continuous integration/continuous delivery (ci/cd) environments. PDF document, 2023.

[13] Marc Ohm, Henrik Plate, Arnold Sykosch, and Michael Meier. Backstabber's knife collection: A review of open source software supply chain attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment: 17th International Conference, DIMVA 2020, Lisbon, Portugal, June 24–26, 2020, Proceedings 17*, pages 23–43. Springer, 2020.

[14] Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Carl Landwehr, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Michael. Perspectives on the solarwinds incident. *IEEE Security & Privacy*, 19(2):7–13, 2021.

[15] SLSA Collaboration. Supply chain levels of software assurance (slsa). Accessed on September 6, 2023.

[16] Sonatype. 2020 state of the software supply chain, 2020.

[17] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. Lastpymile: identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 780–792, 2021.

[18] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45. IEEE, 2020.

[19] Gentoo Wiki. Gentoo wiki incident report, June 2018.

[20] Qiushi Wu and Kangjie Lu. On the feasibility of stealthily introducing vulnerabilities in open-source software via hypocrite commits. *Proc. Oakland*, 2021.

[21] Tianhang Zheng, Hao Lan, and Baochun Li. Be careful with pypi packages: You may unconsciously spread backdoor model weights. *Proceedings of Machine Learning and Systems*, 5, 2023.