# Vulnerability Detection Across Different CI/CD Platforms

**Vasco Manuel Oliveira**
Checkmarx, Braga, Portugal
ALGORITMI Research Centre/LASI, Dep. of Informatics, University of Minho, Braga, Portugal

**Alberto Simões**
Checkmarx, Braga, Portugal
2Ai – School of Technology, IPCA, Barcelos, Portugal
LASI – Associate Laboratory of Intelligent Systems, Guimarães, Portugal

**Pedro Rangel Henriques**
ALGORITMI Research Centre / LASI, Dep. of Informatics, University of Minho, Braga, Portugal

## Abstract

In recent years, more attention has been paid to the security of the software supply chain (SSC). While at first SSC was just seen as the dependency of other libraries, today SSC is broader, considering all the environment where a software application is developed, from the actors, hardware and auxiliary tools.

This work focuses on a specific part of software supply chain security: the tools used for continuous integration and continuous deployment (CI/CD) and their vulnerabilities and risks. These tools are widely used by organizations to accelerate their software development, testing, and delivery, making any security issue present in these tools problematic for the organization. This is especially true given that most tools are open-source, making these tools the primary targets for exploits.

We will present a quick introduction to SSCS and CI/CD and provide a practical solution to detect risks and vulnerabilities in CI/CD tools, emphasizing the modular approach, allowing the system to easily scale to detect new risks and vulnerabilities, as well as to support new CI/CD tools.

**2012 ACM Subject Classification** Software and its engineering → Software verification and validation

**Keywords and phrases** Software Supply Chain, Software Supply Chain Security, CI/CD Platforms

**Digital Object Identifier** 10.4230/OASIcs.SLATE.2025.4

**Funding** *Alberto Simões*: This work has been partly supported by Fundação para a Ciência e Tecnologia, under framework of the Strategic Fundings UIDB/05549/2020, UIDP/05549/2020 and LASI-LA/P/0104/2020.
*Pedro Rangel Henriques*: This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Unit Project Scope UID/00319/Centro ALGORITMI (ALGORITMI/UM)

## 1 Introduction

The Software Supply Chain (SSC) includes the environment in which the software is developed. Focuses on the actors, the hardware, and the software used. This complex environment is susceptible to security risks, vulnerabilities, and attacks. Although detecting vulnerabilities in actors and hardware is almost impossible, new work has been done to protect the software used during the software development cycle. This is where this work fits, and all references to SSC Security (SSCS) during the remainder of this article are specific to the security of the software used during the development process.

The SSC consists of four different blocks [12]: the Source, the Build, the Dependencies, and the Deployment or Packaging phase. Continuous Integration and Continuous Deployent (CI/CD) tools can be used on most of them [3]: during the Source phase for code analysis,


during the Build phase for automating the compiling and unit testing, and during the Deployment or Packaging, to create deliverables and deploy the software for production. Examples of CI/CD tools are Github Actions [4] or Jenkins [7].

## 1.1 Motivation

CI/CD tools enable faster, more flexible, and diverse software delivery. However, they have also expanded the attack surface, giving attackers more ways to exploit vulnerabilities. Consequently, the number of attacks abusing flaws in the CI/CD ecosystem in the industry has been significantly increasing [13]. It is therefore essential to develop effective methods for identifying, understanding, and managing the risks inherent in these environments.

## 1.2 Objectives

This article seeks to investigate and document theoretical knowledge and present a practical solution in the field of CI/CD security, with an emphasis on enhancing their security in the context of Software Supply Chain Security.

Since this is a very recent topic and it still needs a lot of investigation, our first goal was to learn how to create a detection software that scans CI/CD tools configurations for risks and vulnerabilities, with an emphasis on acting with modularity, by seamlessly being able to implement rules for existing and newly found risks and vulnerabilities, and being easy to extend for new and different CI/CD tools.

# 2 Background and State of the Art

This section introduces the concept of Software Supply Chain and the relevance of its security. Follows a discussion on what are Continuious Integration/Continuous Deployment tools and their risks and vulnerabilities in the context of SSC.

## 2.1 Software Supply Chain

In general, a Supply Chain is a network of interconnected entities that work together to transform raw materials into finished products for consumers. This chain begins with some raw material extraction or production, proceeds through the manufacturing, transportation and distribution processes and concludes with retail sales to many consumers [11].

In computing, Software Supply Chain refers to all components, tools and processes involved in creating and deploying software products. These chains include code, configurations, open-source libraries, proprietary binaries, plugins, and dependencies, all coordinated through build tools and repositories. Each part of this interconnected chain brings functionality but also introduces vulnerabilities, as dependencies can propagate risks from one element to the entire system. The complexity and interdependencies inherent in these Supply Chains make them particularly vulnerable to attacks, with bad actors exploiting gaps in these chains, such as misconfigurations or outdated software, to infiltrate and compromise systems.

To manage those risks, organizations usually request a Software Bill of Materials, a detailed list of all third-party components in a project, enabling transparency and compliance with industry standards. However, using a Software Bill of Materials is not enough, with those vulnerabilities continuing to exist across infrastructure, software and codebase levels, as well as actors and processes involved. To strengthen the Software Supply Chain Security, organizations are increasingly adopting CI/CD pipelines, integrating automated scanning

and testing tools to detect misconfigurations, exposed secrets, and dependency vulnerabilities early in the development cycle. By implementing these security checks at every stage of the pipeline, teams can reduce exposure and respond swiftly to threats, helping to secure the software ecosystem against evolving attacks [6].

There are already some tools that provide security to the Software Supply Chain, such as Legit Security [9], a platform that protects the integrity of the SSC by automatically discovering and securing pipelines, infrastructure and other elements of the SSC. Another example is the OX Security platform [14], which offers an end-to-end approach to securing the software development lifecycle by providing visibility, risk analysis, and active protection across the entire pipeline. However, these tools are not open-source, which limits their scalability and adaptability, especially as new threats and attacks against the Software Supply Chain continue to emerge on a daily basis.

## 2.2 Continuous Integration/Continuous Deployment

Continuous Integration/Continuous Deployment, also known as CI/CD, is a software development methodology that, aligned with agile principles[1], aims to accelerate the software development lifecycle. Originating from the eXtreme Programming methodology[2] [1], CI/CD has become central to more agile software development.

### Continuous Integration (CI)

CI/CD consists of two core practices, with the first practice being Continuous Integration or CI, which consists of frequent integrations along with automated testing of the code pushed by developers to a shared repository. It ensures that the committed code changes do not include any bugs, guaranteeing the stability and functionality of the codebase. This process includes various elements such as:

- **Automated Build Process:** Automating the application's build process, ensuring that the latest code changes are always in a deployable state.
- **Instant Feedback:** Developers get prompt feedback on their code changes, enabling them to quickly address any issues.
- **Enhanced Code Quality:** CI/CD pipelines often include static code analysis and security scanning to identify vulnerabilities.
- **Performance Monitoring:** Performance monitoring is implemented to ensure that recent code changes do not negatively impact the application's performance.

### Continuous Delivery/Deployment (CD)

The second practice can be approached in two distinct ways, depending on whether the process is partially or fully automated, respectively:

- **Continuous Delivery:** involves ensuring that the application is always in a deployable state after passing the integration stage. The final deployment to production requires manual approval.
- **Continuous Deployment:** builds upon this by fully automating the process. Code changes that pass automated tests are deployed directly to production without human intervention. This approach enables faster and more frequent delivery of updates, ensuring consistent and reliable deployments.

[1] The agile principles are a set of guidelines that promote flexible and efficient software development.
[2] XP is a software development methodology intended to improve software quality and responsiveness to changing customer requirements.

These two practices together form a CI/CD pipeline, accelerating software delivery by automating and optimizing the processes of integrating, testing, and deploying code [17].[3]

### CI/CD Tools

Thus, CI/CD tools allow for these two practices. This automation minimizes human intervention, reduces errors, and ensures faster delivery of high-quality software. CI/CD tools, such as GitHub Actions [4] or Jenkins [7], provide the infrastructure necessary to define and execute this kind of pipelines, allowing for the automation of these workflows [16, 17]. A CI/CD pipeline, or CI/CD workflow, is a sequence of automated steps designed to streamline the process of delivering new software versions [5].

## 2.3 CI/CD Risks and Vulnerabilities

To properly illustrate how CI/CD tools are susceptible to attacks, this subsection discusses a few examples of CI/CD security risks, based on the top-ten CI/CD security risks, published by OWASP [13]. Despite not being the most recent publication, it remains one of the most widely referenced resources for understanding possible vulnerabilities and risks in CI/CD ecosystems.

The main goal of this list is to help defenders identify focus areas for securing their CI/CD ecosystem, by providing risks that cover the full CI/CD pipeline lifecycle, from source code to production, and are based on real-world attacks and incidents. All of these risks include a clear description, real examples or use cases, consequences of exploitation and recommendations/mitigations. This was only made possible through extensive research into attack vectors associated with CI/CD, as well as the analysis of high-profile breaches and security flaws. As the information in this list is very extensive, only a few selected risks will be displayed in this article, but all of them can be viewed on the OWASP website[4].

### Insufficient Flow Control Mechanisms

Attackers with access to CI/CD components (SCM[5], CI, artifact repository[6], etc.) can push malicious code or artifacts due to weak enforcement of approvals and reviews. CI/CD pipelines prioritize speed, often relying on automation with minimal human intervention.

Without proper flow control, an attacker could:

- Push malicious code that gets automatically deployed.
- Exploit weak branch protection or auto-merge rules.
- Upload tampered artifacts that are later deployed.
- Directly modify production code or infrastructure without verification.

To mitigate these vulnerabilities, the defenders should adopt the following strategies:

- Implement branch protection rules and restrict exclusions.
- Limit auto-merge usage and ensure strict validation.

---

[3] Besides this section, whenever the acronym CD is mentioned, it is referring to Continuous Deployment and not Continuous Delivery, unless it is stated otherwise.

[4] See `https://owasp.org/www-project-top-10-ci-cd-security-risks/`.

[5] tools and systems used to track and manage changes to source code, such as GitHub, GitLab and Bitbucket.

[6] An artifact repository is a system for storing, managing, and retrieving build artifacts (like compiled code, libraries, packages or container images), which are typically generated during the CI/CD pipeline process.

- Require additional approval before triggering production deployments.
- Prefer allowing artifacts that were created by a pre-approved CI service account to flow through the pipeline.
- Detect and correct drifts between production code and CI/CD sources.

### Poisoned Pipeline Execution (PPE)

PPE is an attack where an adversary with access to a source control system, but not the build environment, modifies CI/CD pipeline configurations to execute malicious code during the build process.

PPE exploits permissions in a source control repository to inject malicious commands into a CI/CD pipeline. Pipelines that execute unreviewed code, from pull requests or commits to arbitrary branches, are particularly vulnerable. Once executed, the attack runs within the pipeline's context, potentially leading to severe security breaches. There are three types of PPE:

- **Direct PPE (D-PPE):** The attacker modifies the CI configuration file, directly in the repository, by doing a direct push to an unprotected branch or through a pull request. Once the pipeline is triggered, the malicious commands run in the build node.
- **Indirect PPE (I-PPE):** Occurs when direct modification of the CI configuration file is restricted. The attacker injects malicious code into referenced files (like Makefiles, scripts, test cases, or tool configurations) that the pipeline executes, and when the pipeline runs, these files execute their malicious instructions.
- **Public PPE (3PE):** Targets public repositories where anyone can submit pull requests. If the pipeline executes unreviewed contributions, an anonymous attacker can inject malicious code. This can expose internal assets, especially if public and private pipelines share the same CI environment.

Once the malicious code is executed, it runs within the pipeline's context and can carry out a variety of harmful operations, such as compromising the build environment, stealing sensitive data, or deploying further malicious code into production. To mitigate PPE risks, the next strategies should be adopted:

- Restrict write access to CI/CD configuration files.
- Enforce rigorous code reviews for changes that affect pipeline configurations.
- Separate CI configuration from source code when possible.
- Limit the exposure of files that can be referenced by the pipeline to trusted sources only.

### Ungoverned Usage of 3rd Party Services

This risk associated with granting third-party services access to an organization's CI/CD systems without proper governance. It expands the attack surface due to the ease of providing third-party services access to sensitive resources.

Organizations often integrate third-party services into their CI/CD systems to increase functionality. However, the methods for connecting these services (like OAuth, SSH keys, or access tokens) are simple to implement, often without requiring additional permissions or approvals. This can result in third parties having extensive access, from reading code in a single repository to full administrative control. These services are also easily integrated into build pipelines, giving them access to resources that could expose the system to security risks.

Lack of governance and visibility can prevent organizations from maintaining proper Role-based access control and least privilege[7]. A single compromised third party could be used by attackers to manipulate code or trigger malicious actions in build systems, potentially affecting broader systems within the organization. To intercept those ungoverned third-party services, the following strategies should be adopted:

- Implement vetting procedures for third-party services before granting access to the environment, ensuring the principle of least privilege.
- Maintain visibility over third-party access, covering integration methods, granted permissions, and actual usage.
- Limit third-party access to only necessary resources and regularly review access controls.
- Periodically remove unused or unnecessary third-party services from the environment.

## 3 Security Frameworks

To effectively analyze and detect potential vulnerabilities in CI/CD components, the systems that will be developed rely on established security frameworks that facilitate pattern matching and standardized reporting. Two key technologies are used to support these capabilities: YARA-X, a high-performance pattern matching engine designed for security analysis, and SARIF (Static Analysis Results Interchange Format), a standardized format for reporting the results of static analysis tools.

### 3.1 YARA-X

YARA-X [18] is a widely used pattern matching tool designed for malware research and detection, and has the primary goal of enhancing the performance, security and usability compared with its predecessor, YARA, aiming to completely replace it in the future.

YARA-X enables users to define rules to identify malware families or other patterns of interest using textual and binary signatures. These rules consist of three main components:

- **Metadata:** Descriptive attributes of the rule, such as a description, threat level and status.
- **Strings:** A collection of textual or binary patterns used for detection.
- **Condition:** A boolean expression that, matched with the Strings components, determines whether the defined patterns match a given input.

Listing 1 presents an example of a rule made to detect a hypothetical malware family, Silent Banker.[8] This rule defines three different patterns (`$a`, `$b`, and `$c`) and triggers a match if any of them is found in the analyzed data.

### 3.2 YARA-X Dictionary Module

During the usage of YARA-X we faced the problem that it is designed for full text search, taking no advantage of structured information. Most tools use configuration files that are not flat. Their structure can be useful, making the process of matching specific keywords in specific fields easier.

---

[7] *Least Privilege* is a security concept that restricts users, applications, and systems to the minimum level of access necessary to perform their functions.

[8] Silent Banker is a monitoring trojan, type of malware that downloads disguised as a legitimate program, that captures screen shots and logs keystrokes.

**Listing 1** YARA-X rule example, "Silent Banker".

```
rule silent_banker : banker {
    meta:
        description = "This is just an example"
        threat_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"

    condition:
        $a or $b or $c
}
```

YARA-X has built-in modules and the possibility for custom ones, which allows the user to add functions and different ways of retrieving data from the file provided.

Taking advantage of this extensibility mechanism, we developed the Dictionary module, specially created for loading structured file formats, like JSON or YAML and storing it in a dot notation, allowing access to keys by their path. The module stored the data in a structure called Dictionary, which consists of a list of key-value pairs. Listing 2 shows an example of such a configuration file, while Table 1 shows the stored data.

**Listing 2** Github Actions pipeline example.

```
name: CI Pipeline
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Set up Node.js
        uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install
      - run: npm test
```

Beyond the structure, the module offers an API to facilitate access to specific fields:

- `full_key(key)`: Verifies if the given key exists in the dictionary. (e.g. `jobs.build.runs-on` would be true if that key exists);
- `is_key(key)`: Verifies if the given key is contained in one or more of the dictionary keys. (e.g. "jobs", "build", "runs-on", "jobs.build" and "build.runs-on" would all be true if `jobs.build.runs-on` exists);

**Table 1** Dictionary structure derived from the example pipeline.

| Key | Value |
|---|---|
| name | CI Pipeline |
| on.push.branches[0] | main |
| on.pull_request.branches[0] | main |
| jobs.build.runs-on | ubuntu-latest |
| jobs.build.steps[0].uses | actions/checkout@v4 |
| jobs.build.steps[1].name | Set up Node.js |
| jobs.build.steps[1].uses | actions/setup-node@v3 |
| jobs.build.steps[1].with.node-version | 14 |
| jobs.build.steps[2].run | npm install |
| jobs.build.steps[3].run | npm test |

- `full_value(value)`: Verifies if the given value exists in the dictionary. (e.g. `actions/checkout@v4` would be true if that value exists in the dictionary);
- `is_value(value)`: Verifies if the given value exists in the dictionary, and if used with a wildcard "`*`" anything that is inside that character counts for the match. (e.g. `actions/checkout@*"` would be true for `actions/checkout@v4` or `actions/checkout@v2`);
- `get_value(key)`: Given a full key, this function returns the value for that key (e.g. the key "name" would return the value "CI Pipeline");
- `any_contains(key, value)`: Verifies if a value belongs to a specific key. If the key has an array (`[]`), the function returns true if any of the values of that array match the given value. If the wildcard "`*`" is used inside the value parameter, then everything that is inside that character will count as a match (e.g. the key `on.build.steps[].uses` would be true for the values `actions/checkout@*` and `actions/setup-node@*`);
- `in_key(key, string)`: Uses the methods `is_key` and `any_contains` to verify if a given string belongs to the key, in cases that the vulnerability would happen for a key or a value with the same string (e.g. for the key "name" it would be true for "CI Pipeline" even if that string was a key and not a value);
- `shows_first(key, key)`: Verifies if the first given key shows first on the dictionary to the second given key (e.g. would be always true if the first key was "name" and the second any other existing key).

## 3.3 SARIF Format

The Static Analysis Results Interchange Format (SARIF) is an industry standard format based in JSON for the output of static analysis tools [2]. Thus, to guarantee interoperability, the developed software should be able to export results in this format. An example of such a result in SARIF format is shown later in this article (see Listing 4).

# 4 Vulnerability Detection

The vulnerability detection in the CI/CD area can be divided in two smaller areas, since we can scan workflows from CI/CD tools (e.g. Github Actions pipeline) as well as plugins that belong to a CI/CD tool (e.g. Jenkins plugin). As will be explained in this section, these areas differ significantly and require different approaches, although they share some scanning methodologies.

## 4.1 CI/CD Workflows

Different CI/CD workflows, or CI/CD pipelines, can have different formats, rules and syntaxes that make them unique and distinguishable from each other. So how exactly can we detect these workflows in a modular way.

### Workflow Formatter

CI/CD workflows can be defined in various formats, depending on the CI/CD tool in use, including YAML, XML, and JSON, with Jenkins being the only exception, using a unique Groovy format.

All of these formats can easily be converted into a dictionary object. By converting each workflow to a dictionary object, before scanning, we ensure consistent parsing across different tools.

Several libraries already support parsing YAML, XML and JSON files to dictionary objects (e.g., Python's `yaml`, `json`, and `xmltodict` libraries). While there is no third-party solution to convert Groovy based Jenkinfiles into dictionaries, it is still achievable using a simple custom parsing function.

### Automatic Tool detector

It is also possible to detect automatically which CI/CD tool a specific workflow belongs to by scanning for certain patterns that are characteristic of that specific tool inside the given workflow.

For example, a workflow containing patterns like "jobs:", "steps:", "uses: actions/", "github.", and "runs-on" likely indicates a GitHub Actions pipeline.

By collecting multiple pattern sets for each tool and evaluating how many of them match a given workflow, we can calculate a confidence score and make an accurate tool prediction.

### Workflow Scan

A workflow can be described as a dictionary, so instead of just using the simple YARA-X features for pattern matching, we can use the YARA-X's Dictionary Module (see Section 3.2) to scan vulnerabilities more accurately.

This is made by using the YARA-X Python API to compile rules and scan the workflows. By converting a rule file into a string and passing it to the YARA-X `compile()` method, we create a `Rules` object that contains those rules. This object provides a `scan()` method, which we use to scan the provided workflow by first converting the already formatted dictionary into a JSON string and encoding it in `utf-8`.

Listing 3 shows an example of how to use the Dictionary Module.

By using the Dictionary module, we can guarantee that there will be a Pull Request Target that is trying to run a command using a checkout action [10], all while verifying if the action checkout is being made before running the command.

Since different CI/CD tools use different rules and syntaxes, we generally can not use detection rules made for a tool to another, however it is the only thing that needs to be different, meaning that we can just make a detection rule file using YARA-X rules and the Dictionary module for each CI/CD tool, easily creating new rulesets for new CI/CD tools and adding rules to existing ones.

**Listing 3** PR Target Rule with YARA-X and dictionary module.

```
import "dictionary"

rule check_pull_request_target {
    meta:
        name = "Pull request target"
        description = "Detects the use of pull_request_target trigger in GitHub
            Actions workflows, which can lead to security vulnerabilities specially
            when combined with an explicit PR checkout."
        impact = "May lead to malicious PR authors (i.e. attackers) being able to
            obtain repository write permissions or stealing repository secrets."
        recommendation = "Avoid using pull_request_target if the workflow doesn't
            need write repository permissions and doesn't use any repository
            secrets. Assign repository privileges only where needed explicitly
            through pull_request and workflow_run."
        reference = "https://securitylab.github.com/[...]-preventing-pwn-requests/"
        gravity = "high"
    condition:
        dictionary.in_key("on","pull_request_target") and
            dictionary.any_contains("jobs.build.steps[].uses","actions/checkout@*")
        and dictionary.is_key("jobs.build.steps[].run") and
            dictionary.shows_first("jobs.build.steps[].uses","jobs.build.steps[].run")
}
```

## 4.2 CI/CD Plugins

Only a few CI/CD tools, such as Jenkins and TeamCity, currently support plugins. However, the scanning module developed in this research to analyze such plugins can also be applied to plugins for other tools (e.g. VSCode or Visual Studio) or future CI/CD tools that implement plugin systems.

### Store Results

Plugins are typically packaged as archive files (e.g. `.zip`, `.hpi`), so, to scan these plugins, we must first extract their contents to access the source code, however, this task can take a lot of time.

A good practice to circumvent this problem is to store the results of the already scanned plugins in a database, ensuring that a plugin only needs to be decompiled once.

Given that plugins can suffer changes over time due to updates or even tampering, we must store not just the plugins, but their versions (in case of an update) and hashes (in case of tampering) as well.

### Plugin's Scan

Plugins will also be scanned using YARA-X, though not with the Dictionary module, as plugin code does not follow a unified structure like CI/CD workflows.

With YARA-X we can make rules with additional information via metadata, which is going to be used to get the rule's description, author and version, as well as capture patterns within a file. With the YARA-X pattern detection, the pattern that was detected gives the match content, offset and length, and with the offset we can easily get in which line the pattern was detected as well.

This is also made by using the YARA-X Python API to compile rules and scan plugin files.

# 5 Security Services

Our research team built an API for each different CI/CD vulnerability detection area according to the discussion presented in the previous section. These APIs, that will be described in this section, serve as an easy way to interact with the scanning engines, allowing external tools or users to submit configurations or plugin data for analysis and receive structured vulnerability reports in return.

Both APIs that we developed were implemented using the FastAPI [15] framework, which provides high performance, automatic documentation, and easy integration with Python based tools and services.

## 5.1 Workflow Scan API

This is an API designed to scan vulnerabilities in CI/CD configuration files, in a modular way. It uses YARA-X and the Dictionary module, specially made for this API, to scan workflows more easily and more accurately. After analyzing a workflow, the API returns the scan results in the standardized SARIF or JSON format. This API was tested using the default configuration workflows of each CI/CD tool.

With modularity as its core objective, this API was developed with the following priorities in mind:

- Support for scanning multiple CI/CD configuration files from different CI/CD platforms;
- The ability to easily add new rules or rulesets to refine and expand detection capabilities across platforms;
- Converting any workflow into a unified dictionary object format to facilitate scanning;
- Automatically recognizing, with high accuracy, which CI/CD platform the configuration belongs to.

### API Endpoints

The API provides the following endpoints:

`POST /scan` – Scans a given CI/CD configuration file for vulnerabilities.

- **Parameters**:
  - `file` – The configuration file to be scanned (uploaded via form-data).
  - `rule_type` – (Optional) The name of the CI/CD platform the configuration belongs to (e.g., `github_actions`, `jenkins`, `circleCI`).
  - `format` – (Optional) The format that the results will be displayed on. Supported values are `sarif` (default) or `json`.
  - `automatic` – (Optional) Boolean flag to enable automatic detection of the CI/CD platform. When set to `true`, the `rule_type` parameter is not required.
- **Returns**: The scan results in SARIF or JSON format. Listing 4 shows an example of a result in SARIF.

`POST /add_rule` – Adds a new YARA-X rule to an existing ruleset.

- **Parameters**:
  - `file` – A `.txt`, `.yar`, or `.yara` file containing the new rule (uploaded via form-data).
  - `rule_type` – The name of the existing ruleset to which the rule should be added (e.g., `github_actions`, `jenkins`, `circleCI`).
- **Returns**: A message confirming the successful addition of the rule, or an error if the rule is invalid or already exists, or the `rule_set` does not exist.

`POST /add_ruleset` – Creates a new ruleset for a CI/CD platform.

- **Parameters**:
  - `file` – A `.txt`, `.yar`, or `.yara` file containing a complete ruleset (uploaded via form-data).
  - `rule_type` – The name for the new ruleset.
- **Returns**: A message confirming the creation of the ruleset or an error if the file is invalid or a ruleset with that name already exists.

**Listing 4** Example of a Result given in SARIF format of scanning a workflow that uses a PR Target.

```
{"version": "2.1.0",
  "$schema": "https://json.schemastore.org/sarif-2.1.0.json",
  "runs": [{
     "tool": {
       "driver": {
         "name": "SecureChain - CI/CD",
         "version": "1.0"}},
     "invocations": [{
         "startTimeUtc": "2025-04-23T13:43:42Z",
         "endTimeUtc": "2025-04-23T13:43:42Z",
         "executionSuccessful": true}],
     "results": [{
         "ruleId": "check_pull_request_target",
         "level": "warning",
         "message": {
           "text": (Description),
           "markdown": (Markdown with Metadata),},
         "locations": [{
            "physicalLocation": {
              "artifactLocation": {
                "uri": "pull_request_target.yml"
              }}}]}]}
```

## 5.2 Plugin Scan API

This API is designed to scan any type of plugins, identifying possible risks and vulnerabilities within those plugins; however, currently it is only able to scan Jenkins plugins. It uses YARA-X to detect these risks and vulnerabilities inside these plugins, supporting not just literal string matching but more string matching mechanisms, such as hexadecimal strings, regular expressions, and many more features. This API was tested using 117 randomly selected publicly available plugins in the Jenkins Marketplace [8].

Just like the Workflow Scan API, this API has the same core objective: "modularity"; and so, creating the rules is as easy and as modular as the previous API. However, when it comes to the file extraction engine itself, it depends a lot on what tool the plugin that is being scanned belongs to, so it is needed to create a special detector for each tool.

### API Endpoints

The API provides the following endpoints:

`POST /plugin_file` – Scans a single plugin file for potential risks and vulnerabilities.

- **Parameters**:
  - `file` – The plugin file to be scanned (e.g., a `.hpi` file), uploaded via form-data.
  - `cfr` – (Optional) The decompiler to use during analysis. Supported values are `cfr` (default) or `jadx`.
  - `format` – (Optional) The format that the results will be displayed on. Supported values are `sarif` (default) or `json`.
- **Returns**: A JSON or SARIF file with the detected plugins and plugin files matches, with the YARA rules and their metadata. If no issues are found, a message indicating the absence of matches is returned. Listing 5 shows an example of a result in JSON.

`POST /plugin_list` – Scans a list of plugins provided in a single file for batch analysis.

- **Parameters**:
  - `file` – A JSON file containing multiple plugins to be scanned.
  - `cfr` – (Optional) The decompiler to use during analysis. Supported values are `cfr` (default) or `jadx`.
- **Returns**: A JSON or SARIF file containing scan results for each plugin. Each entry includes rule matches or a message indicating that no matches were found.

`POST /plugin_url` – Downloads and scans a plugin from a given URL.

- **Parameters**:
  - `url` – The URL from which to download the plugin.
  - `cfr` – (Optional) The decompiler to use during analysis. Supported values are `cfr` (default) or `jadx`.
- **Returns**: A JSON or SARIF file with the YARA rule matches if any are detected. Otherwise, a message stating that no matches were found is returned.

**Listing 5** Example of a Result given in JSON format of scanning a plugin.

```
{"results": {
   "malicious-plugin-test": {
     "ExamplePlugin.java": {
       "suspicious_commands": {
         "category": "Commands",
         "metadata": [[
             "description",
             "Detects suspicious system commands often used in reconnaissance
                 or post-exploitation phases"],
           ["author",
             "Vasco Oliveira <vasco.oliveira@checkmarx.com>"],
           ["version",
             "1.0"]],
         "matches": [[
             "uname -a",
             21]
         ]}}}}}
```

## 6 Conclusion

This article serves as an introduction to Software Supply Chain Security and Continuous Integration/Continuous Deployment. It discusses how to develop scanning software that can scan CI/CD tools and their components in a modular way. The scanners and the APIs we provided were designed in a way to ensure that future improvements can be seamlessly integrated, requiring only the addition of new rules to detect known and new risks and vulnerabilities. While this publication focuses on the system itself, future publications will point out the risks and vulnerabilities to which these systems are susceptible. No empirical evaluation or performance benchmarks were included, since such analysis was outside the scope of this work.

The software is not yet publicly available but will be disclosed as open-source and published on Checkmarx's GitHub repository.

For future work, there is still much to be done, with the next step being to scan reusable components of CI/CD tools, such as actions for GitHub Actions and orbs in CircleCI. Since this work focuses on the framework for vulnerability and risk detection, the final step will be the responsibility of the application security team, who will develop the YARA-X rules used to identify risks in CI/CD components.

## References


1 Kent Beck. *Extreme Programming Explained: Embrace Change*. XP series. Addison-Wesley Professional, illustrated edition, 2000.

2 Microsoft Corporation. SARIF. `https://sarifweb.azurewebsites.net/`.

3 Clint Gibler and Francis Odum. An Overview of Software Supply Chain Security. `https://tldrsec.com/p/supply-chain-security-overview`, 2023.

4 GitHub. GitHub actions, 2024. URL: `https://github.com/features/actions`.

5 Red Hat. What is a CI/CD pipeline? `https://www.redhat.com/en/topics/devops/what-cicd-pipeline`, 2022.

6 Senior Technical Content Marketing Manager Jacob Schmitt. Software supply chain: What it is and how to keep it secure. `https://circleci.com/blog/secure-software-supply-chain/#what-is-the-software-supply-chain`, 2023.

7 Jenkins. Jenkins, 2024. URL: `https://www.jenkins.io/`.

8 Jenkins. Jenkins plugins index. `https://plugins.jenkins.io/`, 2025.

9 Legit Security. Software supply chain security, 2025. URL: `https://www.legitsecurity.com/software-supply-chain-security`.

10 Jaroslav Lobačevski. Keeping your GitHub Actions and workflows secure: Preventing pwn requests. `https://securitylab.github.com/resources/github-actions-preventing-pwn-requests/`, 2021.

11 McKinsey & Company. What is supply chain? `https://www.mckinsey.com/featured-insights/mckinsey-explainers/what-is-supply-chain`, 2023.

12 Open Source Security Foundation and Google. Supply-chain Levels for Software Artifacts (SLSA). `https://slsa.dev`, 2021. Accessed: 2025-04-28.

13 Foundation OWASP. OWASP Top 10 CI/CD Security Risks. `https://owasp.org/www-project-top-10-ci-cd-security-risks/#`, 2022.

14 OX Security. Ox security, 2025. URL: `https://www.ox.security/`.

15 Sebastián Ramírez. Fastapi. `https://fastapi.tiangolo.com/`. Accessed: 2025-04-21.

16 Pooya Rostami Mazrae, Tom Mens, Mehdi Golzadeh, and Alexandre Decan. On the usage, co-usage and migration of ci/cd tools: A qualitative analysis. *Empirical Software Engineering*, 28(52), 2023. `doi:10.1007/s10664-022-10285-5`.

**17** Bhaskara Sahithi Shanmukhi. Implementing and using ci/cd: Addressing key challenges faced by software developers. *International Journal of Scientific Research in Engineering and Management (IJSREM)*, 08(08), 2024.

**18** VirusTotal. YARA-X. `https://virustotal.github.io/yara-x/`, 2024.