technische universität
dortmund

OCI-based Vulnerability Containers with Metadata for Automated Evaluation of Security Analysis Tools

Bachelor Thesis

David Mehren
2022-05-16

Supervised by
Prof. Dr. Falk Howar
Simon Dierl, M.Sc.

TU Dortmund University
Department of Computer Science
Chair 14 for Software Engineering
AQUA Research Group

AQUA
automated
quality assurance

Contents

Abstract

Security analysis tools search for vulnerabilities in source code or binaries. Testing these tools requires databases of software with known vulnerabilities, but no database suitable for automated analysis exists today. To decide if an analysis tool has detected the correct vulnerability, metadata must be sufficiently detailed, which existing projects lack.

This thesis presents a metadata format and a corresponding command line tool, together called *Containers for Security Analysis Tools* (CSAT). CSAT can be used to package security vulnerabilities in OCI containers and to automate the analysis of these containers with security analysis tools. CSAT's metadata enables verification of analyzer results and easy re-use of vulnerabilities with different analysis tools. It also allows to automatically check for compatibility between vulnerability and analyzer. The CSAT command line interface supports users in packaging vulnerabilities and automates the build and analysis workflow. It uses Docker containers as a packaging format and to execute analyses independent of the host operating system.

We demonstrate CSAT's modular architecture by integrating two build systems and four analysis tools. Finally, we use CSAT to automatically reproduce results from another paper, showing that our tool is suitable for real-world use.

1. Introduction

1.1. Security Analysis

As software becomes more and more ubiquitous and integrated into everyday items with trends like the *Internet of Things*, software security also moves into the spotlight. Since 2016, the number of publicly disclosed security vulnerabilities has steadily risen [1]. The automated analysis of software for security issues, also known as *application security testing* (AST) has therefore become more popular and widely used development platforms like GitHub and GitLab have integrated AST into their offerings.

Security analysis tools can automatically detect common security problems, e.g., usage of weak hash functions or insufficient validation of user-provided data, by analyzing software source code or binaries. Various techniques for analysis already exist and new approaches are continuously researched.

Developers of new tools naturally want to evaluate the effectiveness of their analyzer. A common approach is to utilize a preexisting collection of vulnerable software artifacts. An artifact typically encompasses the source code and metadata describing the vulnerability. The newly developed tool can then analyze each artifact and report its findings. These are compared to the metadata, allowing to judge whether the analyzer reliably and correctly detects security issues. Performing such an evaluation at scale is labor-intensive, as currently no sufficiently generic system for automating the process exists.

Creators of security benchmarks and vulnerability databases face related challenges. For example, a researcher developing a new technique for detecting vulnerabilities in public source code repositories cannot easily test the extracted issues with existing analyzers. No common standard for vulnerability packaging and description exists, so every project needs to create its own method of making artifacts available for use by others.

Another challenge in testing security analysis tools is posed by dependency problems: analyzed code may require older versions of libraries or runtimes, like end-of-life versions of the Java Runtime Environment or C libraries. Installing such versions in the normal operating system may be tedious or present security issues. This makes it more and more difficult to reproduce findings in aging software.

This thesis presents an automation framework and a format for vulnerability packaging and metadata, solving the challenges of both developers of analysis tools and vulnerability databases. Newly developed analysis tools can be integrated and have instant access to all vulnerabilities using our packaging. Vulnerable software can be packaged and described with our metadata format, allowing all integrated analyzers to examine it.

1.2. Related Work

In preparation of this thesis, we examined existing projects that collect buggy or vulnerable software. We also looked at the automation frameworks used by various scientific competitions on software research.

1.2.1. Evaluation of Existing Bug Databases

To be useful for testing of security analyzers, bug or vulnerability databases must provide the source code of the collected software in a way that allows automated processing. Additionally, the metadata must be sufficiently detailed to allow evaluation of analysis tool results. We created six criteria used in the evaluation:

- **Source Code Availability:** The source code of the software in the affected version is provided. Alternatively, a reference to a *version control system* (VCS) commit, from which the source code can be fetched, is available.
- **Build Information:** The database contains information about the build process for each artifact, so the source code can be compiled into a binary. Additionally, some generic way of executing the build process should be provided.
- **Bug Information:** The database contains information about the type of (security) issue and, if available, a unique identifier that allows to cross-reference the issue with other databases.
- **Bug Patch:** The database provides a patch for the vulnerability, either directly or as a VCS reference. The patch allows to generate a non-vulnerable version of the artifact, so that false-positive findings of analyzers can be detected. In the best case, the only change in the patch is the fix for the particular security issue.
- **Bug Location:** Each artifact has information about the precise location of the issue in the source code, including a filename, line, class or method.
- **Isolation from Host:** The project provides a way to execute or analyze the buggy software in a host-independent environment, allowing separate versions of runtimes or libraries to be used.

Table 1.1 contains an overview of the results of the examination. The evaluated projects can be roughly divided into two classes. First are projects that focus on collecting security issues and there-

Database	Programming Languages	Source Code	Build Info	Bug Info	Bug Patch	Bug Location	Isolated from Host
Big-Vul	C/C++	✓[1]	×	✓	✓[2]	×	×
Bugs.jar	Java	✓[1]	×	×	✓	×	×
BugsInPy	Python	✓[1]	✓[3]	×	✓	×	✓[4]
BugsJS	JavaScript	✓[1]	✓[5]	×	✓[2]	×	✓[4]
CrossVul	mixed	✓	×	✓	✓[2]	×	×
Defects4J	Java	✓[1]	✓[6]	×	✓[2]	×	×
NIST Julia Test Suites	C/C++, Java	✓	✓[7]	✓	×	✓	×
OWASP Benchmark	Java	✓	✓[8]	✓	×	×	×
SAP Project KB	Java, Python	✓[1]	×	✓	✓[9]	×	×

Table 1.1.: Existing Bug & Vulnerability Databases, compared for their suitability for automated security analysis

fore contain vulnerability-specific metadata, like Big-Vul [2], CrossVul [3], the OWASP Benchmark [4] and the NIST Julia Test Suites [5]. The other class contains projects that more generally collect buggy software without a focus on security and includes BugsJS [6], BugsInPy [7], Defects4J [8] and SAPs Project KB [9].

All evaluated databases provide the source code, mostly as references to commits in the Git version control system of the respective source software. Many projects also encompass patches for the bugs they contain, often as VCS references but also as diff files.

Availability of build information is mixed. BugsInPy and BugsJS both package bugs for interpreted languages, so do not need a build system. The NIST Julia Test Suites and the OWASP Benchmark provide build files for common Java build tools, but only the NIST databases also contain a system to automatically execute analyzers on their vulnerabilities. Also noteworthy is Defects4J, which abstracts build tools using a custom execution framework.

BugsInPy and BugsJS stand out by being the only projects to provide means for host isolation. Both use Docker (which we explain in more detail in Section 2.1) and provide a single image set up to exe-

[1] Referenced by VCS URL and/or commit hash
[2] The commit hash of a fix is available, so a patch can be generated
[3] Provides build script per application
[4] Can execute all tests inside a Docker container
[5] Provides Python tooling that executes tests
[6] Provides build system abstraction layer
[7] Provides Ant build files, and a Python utility to execute test cases with analyzers
[8] All vulnerabilities are part of a single Maven project
[9] The commit hash of a fix and archives of buggy and fixed source code are available, so a patch can be generated

cute tests. The individual bugs are still referenced by their Git commit identifier and not packaged as individual Docker images.

To correctly judge the results of security analysis tools, information about the location of bugs in the vulnerability source code is required. However, only the Julia Test Suites provides these details.

In summary, the evaluation made clear no existing project can satisfy all six criteria. Particularly, only one database contains information about bug locations and only two isolate the analysis from the host. Therefore, no existing bug collection is suitable for evaluation of security analysis tools. Additionally, all databases are incompatible with each other. While most projects use CSV or XML to store metadata, the specific data format is different, meaning data cannot be easily exchanged.

The new metadata format presented in this thesis will be specifically developed for testing security analysis tools. Therefore, it will include information about the location of security issues in the source code. Additionally, our approach will provide an execution environment with host isolation.

1.2.2. Software Competitions

The scientific community organizes several competitions on software research topics. Examples are *Test-Comp* [10] for software testing, *SV-COMP* [11], focusing on software verification and *SMT-COMP* [12] for SMT solvers. The participants of these competitions provide tools, often the result of their latest research. During the competition, the tools are given various tasks and are then evaluated using criteria such as accuracy or performance.

The nature of these competitions requires automation for the execution of the participating tools, as the number of tasks that must be performed makes a manual approach infeasible. For example, the 2022 edition of SV-COMP encompassed more than 15000 tasks with 47 participating verification systems [11]. Competitions therefore developed systems that automate test execution, which we evaluated for their suitability regarding testing of security analysis tools.

SMT-COMP utilizes the StarExec execution service and provides tasks to the entrants using SMT-LIB [13]. As this system is only able to operate on satisfiability modulo theories, is unsuitable for our use-case.

Both SV-COMP and Test-Comp perform the competition with *BenchExec* [14]. This framework executes the tools while measuring and limiting their resource usage. This allows for a fair comparison, ensuring, for example, that all participants are provided with the same amount of CPU time. However, BenchExec relies on the participating tool itself to determine whether a task was performed correctly. This presents a problem for evaluating security analysis tools, as these do not provide such functionality and require a separate tool with access to vulnerability metadata to judge results. Therefore, BenchExec is also not an appropriate solution.

1.3. Thesis Goals & Requirements

After the evaluation of preexisting solutions, we formulated two main research questions for this thesis:

- **RQ1: What metadata is required to effectively evaluate security analysis tools?**
  The basics of this topic were already considered in the evaluation, as we established that *build information*, *bug information* and *bug location* is needed. We will further detail the required metadata and develop a concrete format specification.
- **RQ2: What infrastructure is required to automate the evaluation of security analysis tools?**
  Our evaluation has shown that, while some vulnerability databases provide tooling to build the software they contain, no project has a method to automatically evaluate the results of security analysis tools. We will develop such a method in this thesis.

The resulting specification and implementation should also satisfy the following further requirements:

- **Ease of use**
  Integrating more analysis tools and describing vulnerabilities should be reasonably simple. The existing solutions often use complicated CSV oder XML formats to describe vulnerabilities and provide no to way to verify completeness of user-entered data.
- **Expandability**
  The architecture and implementation should not limit the integration of analysis tools to a specific programming language or analysis type.
- **Isolation from Host**
  To avoid dependency problems and improve reproducibility, vulnerable software should be executed and analysed in an isolated environment.
- **Support for vulnerability patching**
  To check if analyzers over-report possible security issues, the removal of vulnerabilities from analyzed code using patches should be supported.

1.4. Structure of this Thesis

The first chapter of this thesis contained an overview of the field of security analysis and evaluated existing collections of software bugs and vulnerabilities in addition to automation systems for software competitions. In the following chapter, we explain various technologies utilized in this thesis and justify their use.

The third chapter lays out the ideas and methods used to describe vulnerabilities and presents a package format. The thesis continues with a more detailed discussion of our implementation of the command line interface in the fourth chapter, highlighting metadata validation, the details of container packaging and the modular systems for integration of build and analysis tools. Chapter five focuses on the user interface, explaining the command line interface and how new analysis and build tool modules are added.

In chapter six, we discuss the analysis and build tools that we integrated for this thesis. Chapter seven evaluates our results, discussing the integration complexity, the metadata and packaging formats and the analysis infrastructure.

Finally, chapter eight presents current limits and possible future expansions and summarizes the thesis.

2. Foundations

To fulfill the requirements laid out previously, this thesis makes use of existing techniques and solutions. In this chapter, we introduce these foundations and give reasons for our choices.

2.1. Docker & The Open Container Initiative

Containers isolate applications from each other and from the host system by using operating system virtualization. In contrast to classic or "full" virtualization, containers share the host kernel and, on Linux, use Control Groups (cgroups) for isolation. Each container has its own filesystem, networking and memory space, while cgroups also enable limiting the resource usage per container.

Containers are a popular way to package, ship and run software and solve different problems: They provide isolation between the host and the running software, they allow shipping software in a defined, reproducible environment, and have become the de facto packaging standard for distributing software.

The concept of running software in a container was popularized starting in 2013 by *Docker* [15]. Docker containers are commonly used without a full operating system inside them, running only one application process per container. In 2015 the *Open Container Initiative (OCI)* was launched to standardize the formats and technologies used. This resulted in the release of the OCI Runtime Specification [16] and the OCI Image Format Specification [17] in 2017[1]. Today, many other container engines capable of running OCI containers are available, e.g., Podman[2] or CRI-O[3]. Containers are also used in Kubernetes[4], a container orchestration toolkit. It manages containers across multiple hosts and provides advanced features like automatically scaling deployments depending on load or managing high availability.

As Docker is still one of the most popular container engines [18], and the advanced features of Kubernetes are not required, this thesis will focus on Docker and will use Docker tooling.

Docker containers are created from *images*, which contain metadata and filesystem *layers*. Multiple containers can be created from the same image and executed simultaneously, isolated from the other

[1]https://opencontainers.org/release-notices/v1-0-0/
[2]https://podman.io
[3]https://github.com/cri-o/cri-o
[4]https://kubernetes.io

containers. Each image layer records only the differences to the previous layer, saving disk space if multiple containers share the same layers. During runtime, all layers are merged, so the running application only sees the sum of all layers. Every Docker image has a *base*. This can either be another image, which means the filesystem layers of that are used as the starting point for the new image, or the special image `scratch`. This image is empty and the only image without a previous layer.

Docker allows adding labels to an image during its build process, which are stored in addition to the container filesystem layers. Each label is identified by a key and maps to an arbitrary string. The vulnerability metadata will be stored in such a Docker label, serialized as JSON. This allows introspection of metadata without needing to interact with the container filesystem.

The most common way to describe the build process of containers is using a *Dockerfile* [19], which employs a domain specific language to describe the steps required to build a container [20].

```
FROM maven
COPY src src
RUN mvn package
LABEL com.example.demo="Packaged with Maven"
```

We will now use the example in Listing 4 to discuss the structure of Dockerfiles in more detail: The `FROM` statement defines the base image. The `maven` image in the example contains an OpenJDK[5] installation in Ubuntu[6] and the Maven[7] project management tool. Using the `COPY` operation, files can be added to the image. The example uses this instruction to copy the source folder of a Maven project into the container. Next, the `RUN` operation is used to execute the Maven binary inside the container, which performs compilation. Both operations each create a new image layer. The final image therefore contains a number of layers from the `maven` base image, and two additional layers created using the Dockerfile. Finally, a label is added to the image using the `LABEL` operation, which only affects metadata and does not create a new layer.

A visualization of the docker image created in the example is displayed in Figure 2.1. The `maven` base image consists of multiple layers, containing the base operating system, a Java installation and the maven binary. The `COPY` command creates another layer, as does the `RUN` command. Adding a label to an image does not create a layer, as the label is included in the image metadata.

[5]https://openjdk.java.net/
[6]https://ubuntu.com/
[7]https://maven.apache.org/

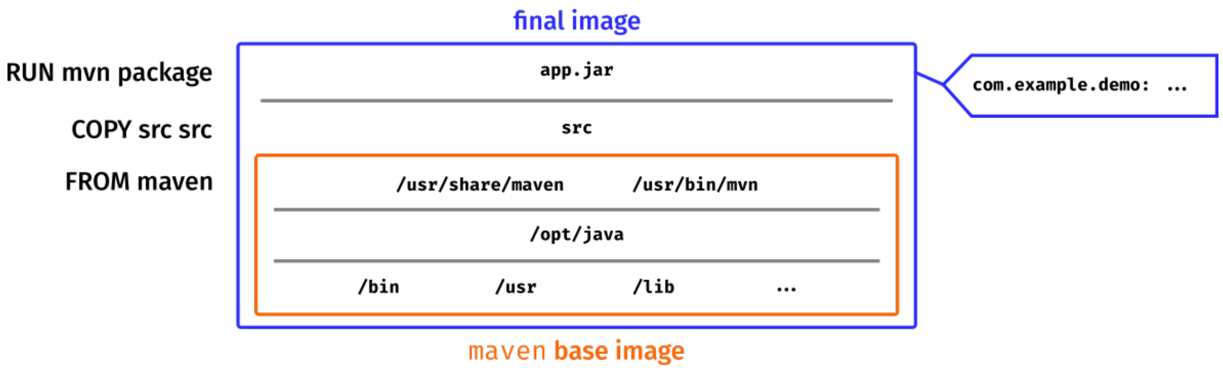


Figure 2.1.: Visualization of a Docker image, consisting of layers and an attached label. The Dockerfile commands creating each layer are displayed on the left.

Images created this way are commonly shared using centralized registries such as *Docker Hub*[8] or the *Quay Container Registry*[9], enabling fast delivery and exchange of applications.

We will leverage Docker to achieve the goal of host isolation. Docker images contain all dependencies, libraries and utilities required to run the packaged application, so the dependency problems outlined in the previous chapter are avoided. Docker is typically executed as a long-running daemon, which provides an HTTP-based API. This API can be used to build container images, up- or download them from registries and to manage container execution. Docker also provides its own command line interface for these tasks, which utilizes the API and is well known by users as the `docker` executable. Our approach will interact directly with the Docker daemon using HTTP, sidestepping the CLI.

2.2. Static & Dynamic Code Analysis

Software analysis techniques can be divided into two categories: Static analysis and dynamic analysis [21].

Static analysis is concerned with obtaining information about a program without running it. This can mean using the source code text itself or introspecting a binary or bytecode [22]. A static analyzer may use syntactic checks to look for calls to insecure API functions or use of a variable before a value has been assigned. It may also use semantic checks to track data or control flow [23]. Commonly used tools that employ static analysis include SpotBugs[10] and find-sec-bugs[11], which find bugs in Java applications. Bandit[12] is another analyzer, designed to detect security issues in Python code.

[8] https://hub.docker.com
[9] https://quay.io
[10] https://spotbugs.github.io
[11] https://find-sec-bugs.github.io
[12] https://bandit.readthedocs.io

In contrast to static analysis, *dynamic analysis* observes program behaviour during execution [24]. As only the code path taken during execution can be analyzed, dynamic analysis might not cover the complete codebase, but can still detect violations of previously defined rules. Some dynamic analysis systems can automatically try to increase the code coverage by generating more input data. Tools utilizing dynamic analysis require various inputs: Some operate directly on a binary and ignore source code. Others may add steps to the standard compilation process, instrumenting the resulting binary. Tools may even require a complete replacement of the build process, for example when they utilize a custom bytecode format for analysis.

```java
public void doPost(HttpServletRequest request,HttpServletResponse response){
  String param = "";
  java.util.Enumeration<String> headers = request.getHeaders("demo");
  if (headers != null && headers.hasMoreElements()) {
    param = headers.nextElement(); // just grab first element
  }
  param = java.net.URLDecoder.decode(param, "UTF-8");
  java.io.File fileTarget = new java.io.File(param, "/Test.txt");
  response.getWriter().println(
    "Access to file: '" +
    ↪ org.owasp.esapi.ESAPI.encoder().encodeForHTML(fileTarget.toString())
    ↪ + "' created."
  );
}
```

A type of dynamic analysis is *taint tracking*, commonly used to find security issues in web applications [23]. In this type of software, unwanted flow of user-provided data to security-sensitive sinks, such as database queries, is a common issue. To detect such flows, a *taint* is attached to user-provided data during runtime and tracked as data flows through the application. When data is passed through dedicated sanitization functions, the taint is removed. These sanitizers “clean” user-provided data, e.g., by escaping control characters in a string, so it cannot influence database queries anymore. When tainted data reaches a critical sink, the analysis environment monitoring the execution can report a possible security issue along with the recorded data flow path.

Listing 12 shows an example of a vulnerability in a web application, detectable by both types of analysis. The application extracts the user-provided value of the demo HTTP header and uses it in a `File` constructor after decoding. This constitutes a *path traversal*, as the value may contain `../`, referencing the parent directory and therefore allowing to access arbitrary directories in the filesystem. An attacker may, for example, send a header value of `../../../etc/passwd` to try to access the user database of a Linux operating system.

2.3. Static Analysis Results Interchange Format

```json
{
  "runs": [{
    "tool": {
      "driver": {
        "name": "SpotBugs", "version": "4.5.3", "language": "en",
        "rules": [{
          "id": "UC_USELESS_CONDITION",
          "shortDescription": { "text": "Condition has no effect." }
        }] } },
    "results": [{
      "ruleId": "UC_USELESS_CONDITION", "ruleIndex": 0,
      "message": {
        "id": "default",
        "text": "Condition has no effect"
      },
      "level": "note",
      "locations": [{
        "physicalLocation": {
          "artifactLocation": { "uri": "Main.java" },
          "region": { "startLine": 7 }
        },
        "logicalLocations": [{
          "name": "main(String[])",
          "kind": "function",
          "fullyQualifiedName": "Main.main(String[])"
        }]
}
```

To exchange and compare the results of the various analysis tools that are available, the *Static Analysis Results Interchange Format* (SARIF) was developed. It is a JSON-based format, standardized by the OASIS SARIF Technical Committee in 2020 [25]. The format is very flexible and can accommodate not only results from security analysis tools, but also from, for example, accessibility checkers or code linting tools. Although it was developed for static analysis, it is also possible to encode results of dynamic analysis with SARIF.

Listing 2.3 contains a shortened SARIF output of the SpotBugs analysis tool. The example shows the main types of information SARIF provides: The `tool` section describes the analyzer that was executed and includes a list of rules that were used for detection. These rules can carry more properties than shown here, for example, a URL to a help page with a more detailed description or tags for the rule type. Each individual result in the `results` array contains the identifier of the rule that was violated, a human-readable description and the location of the affected code. The `physicalLocation` describes the location of the issue in the source code text, while the `logicalLocations` point to the class or method names.

We use SARIF as an internal representation of analysis results and to make the results available to other tools. As SARIF is already supported by many analysis tools, like CodeSonar[13], the Clang Static Analyzer[14], SpotBugs[15] or PyLint[16] [26], we also use it as an input format.

[13]https://www.grammatech.com/products/source-code-analysis
[14]https://clang-analyzer.llvm.org
[15]https://spotbugs.github.io
[16]https://pylint.org

3. Vulnerability Metadata & Packaging

In the Chapter 1 we laid out the challenges faced by developers of security analysis tools and vulnerability databases. This chapter presents our approach for metadata and a packaging format, called *Containers for Security Analysis Tools* or CSAT. It enables easier exchange and re-use of vulnerabilities.

3.1. Metadata for Vulnerable Software

CSAT's metadata is used for two purposes: To describe vulnerabilities for users and to provide bug information used for automation and judgement of analysis results.

For the primarily user-facing part of the metadata, we include similar attributes as used by other databases. Vulnerable software is described by its name, version, the vendor and a URL. This can be used to e.g., display a list of available vulnerabilities to the user or to search for a specific software.

The second part of the vulnerability metadata is focused on automation. In Section 2.2 we explained the concept of dynamic code analysis. As this type of analysis observes a programm during runtime, it requires an executable. Vulnerability databases typically only contain source code, therefore compilation is required.

To automatically compile source code, our metadata needs to provide information about the build system. CSAT must be able to choose the correct compiler or build tool for packaged software. The metadata therefore describes the programming language and identifies the build system. To correctly execute the created binary, metadata includes information about the binary path, required arguments and the main class. As not all software needs to be compiled or has a main class, all of these attributes are optional.

The metadata also needs to describe the vulnerability sufficiently to make it possible to decide if an analyzer has detected it correctly. We include three key facts: The type of vulnerability, a unique identifier and the location in the source code. It is also possible not to include any vulnerability details, which signifies that the packaged software explicitly does not have any vulnerabilities. Packaging non-vulnerable software allows checking whether analysis tools over-report issues, i.e. have false-positives.

To describe the type of vulnerability, we make use of the *Common Weakness Enumeration* (CWE), a list of hardware and software weakness types [27]. For software, CWE categorizes weaknesses that “are frequently used or encountered in software development” [28], e.g. CWE-89 describes “Improper Neutralization of Special Elements used in an SQL Command” or SQL injection [29]. CWE categories are commonly used in security products, like static analyzers or vulnerability databases [30].

Another industry standard is the *Common Vulnerabilities and Exposures* (CVE) program [31]. In contrast to the vulnerability *types* identified using CWEs, a CVE identifies a specific, publicly known vulnerability. It includes a list of affected products, a description and references to e.g., the vulnerability announcement by the software vendor. By providing the CVE ID of a vulnerability in the metadata, we make it possible to cross-reference other databases for information.

The metadata includes information about the location of the vulnerability in the source code. It supports including the filename, line number, class name and function name, mirroring the approach from SARIF of including both the physical and logical locations. As detailed information is not always available for every vulnerability, only the filename is required, the other properties are optional.

```yaml
name: "SecureProgram"
vendor: "Secure Inc."
version: "1.0.0"
language: "java" # Supported: java
url: "https://example.com"
build:
  build_system: "javac" # Supported: javac, maven
exec:
  bin_path: "build/secure-program" # Optional
  main_class: "com.example.SecureProgram" # Optional
  arguments: "-c start" # Optional
vulnerability: # Optional
  cve: "CVE-2020-1234" # Optional
  cwe: "CWE-1234" # Optional
  location:
    file: "src/main/java/com/example/App.java"
    line: 5 # Optional
    class_name: "com.example.utils" # Optional
    function_name: "main" # Optional
provides_patch: false # Optional
```

Finally, a field for availability of a patch is included in the metadata. The patch must remove the vulnerability from the packaged software and is used by the command line utility to generate a non-vulnerable variant of the vulnerability package. This topic is further discussed in Section 4.3.

As can be seen in the example in Listing 3.1, CSAT’s metadata utilizes YAML. This decision will be dicussed in more detail in Section 4.2. The metadata fulfills the requirements from Section 1.2.1. Specifically, the `build` and `exec` sections contain *build information* necessary to create and execute a binary for dynamic analysis. The results of analysis tools can be compared to the included *bug information* and *bug location*, from the `vulnerability` section.

3.2. Packaging Vulnerabilities as Container Images

In addition to metadata describing the vulnerabilities, developers of vulnerability databases require a way to package the collected artifacts and make them available to other projects. Another challenge laid out in Section 1.1 is ensuring reproducibility of the build and analysis processes and independence from the host system.

Containers are a commonly used way to tackle these problems, as explained in Section 2.1. This section will provide an overview of our concept of packaging security vulnerabilities into Docker containers.

Each container image packages only one vulnerability. It encompasses the vulnerable software and associated metadata, attached as a label.

A container image with vulnerability source code is called a *source image* and consists of only one filesystem layer. The application source code is always stored in the `/src` directory of the container filesystem, which is otherwise empty. The source image is created using the user-provided metadata and source code. If a patch is also provided, a second source image is produced, containing the patched source code.

When a vulnerability is to be analyzed, either the source code or a binary is used, depending on the type of analysis. For analyzers that operate directly on the source code, the source container can be used as-is and no further steps are necessary. If the vulnerable software needs to be compiled, a build process as represented in Figure 3.1 is executed. It consists of three steps:

- First, a temporary build image is created. The base layer consists of the build tooling required by the vulnerable software. For many common tools like Maven, Ant or GNU Autotools, images are already available from a Docker registry. The source image is added as a second layer, bringing together the source code and the build tool in the same environment.
- The build process can then be executed by running the build tool. This will create a new filesystem layer with the build artifacts, which must be placed in the `/build` folder.
- This layer can then be used to create a final container image, that only contains the source code from the source image and the artifact. This image is called the *binary image* and can then be used for binary analysis.

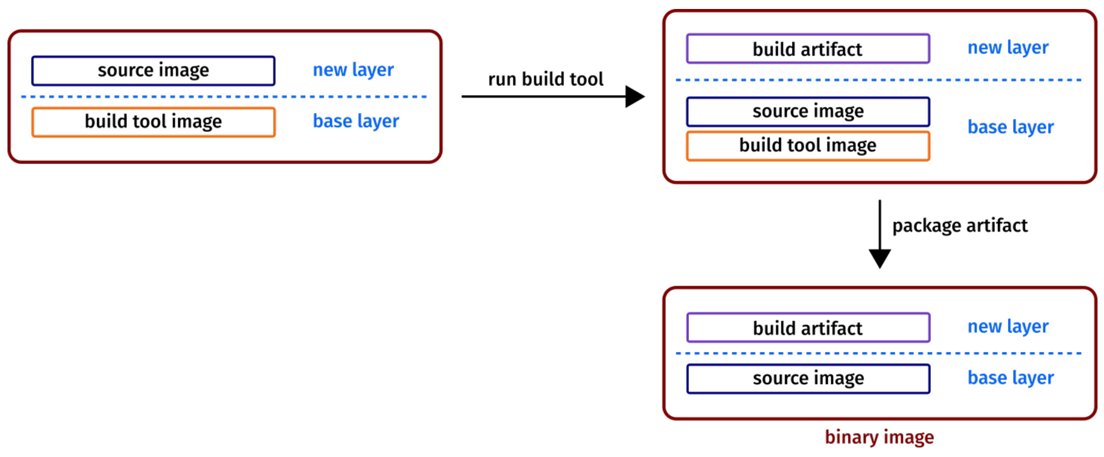


Figure 3.1.: A binary image is created using the source image and a build tool image. The final image only contains the build artifact and the source code.

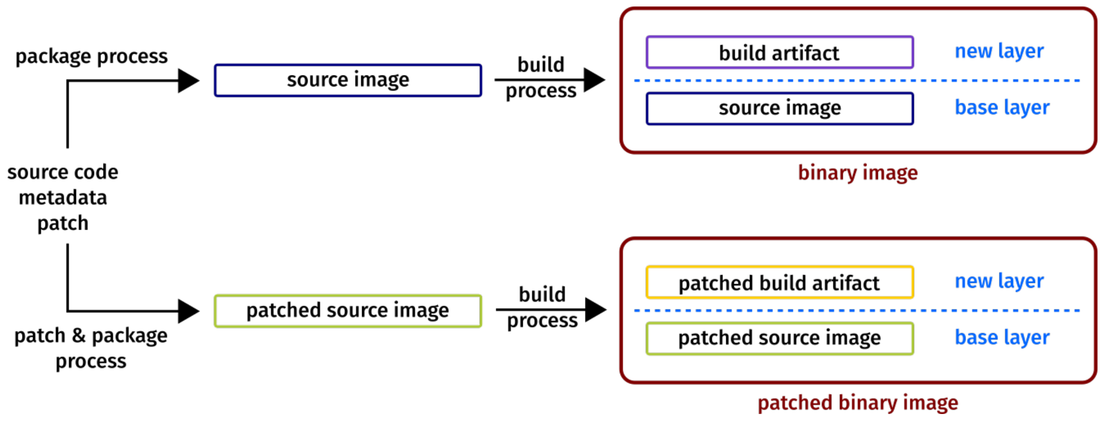


Figure 3.2.: Overview of CSAT's vulnerability build & patching process. Up to 4 image variants can exist per vulnerability.

In case a patched source container is available, the analysis can also be executed on the patched variant of the vulnerability. When the analyzer requires a binary image, the build process is repeated with the patched source image.

In summary, up to four images can exist per vulnerability, as shown in Figure 3.2: an (unpatched) source image, a corresponding binary image and a patched variant of each.

These Docker images can be easily shared using preexisting registry infrastructure like Docker Hub. Projects collecting vulnerabilities can upload their container images, making them globally available.

4. Architecture of the `csat` Command Line Tool

The CSAT metadata and packaging formats can already be used to describe and exchange vulnerabilities. We now present the accompanying `csat` command line interface (CLI). It provides a user-friendly way to interact with packaged vulnerabilities and automates the analysis process.

This chapter focuses on the internal architecture of the CLI, while the next chapter discusses the user interaction and integration of new tools in more detail.

4.1. Features and Architecture

The CSAT command line interface provides these main features:

- **Metadata Validation:** `csat` validates vulnerability metadata according to a schema and provides human-readable error messages.
- **Vulnerability Packaging:** `csat` packages vulnerability source code on the file system into a container image.
- **Analysis Automation:** `csat` automates the analysis of these containers with security analysis tools. The compatibility between analyzer and vulnerability is checked automatically and source code is automatically compiled if necessary.
- **Result Verification:** `csat` compares the analysis results with the expected values from vulnerability metadata.

While vulnerabilities, once packaged, reside in the local Docker image registry, build tools and analyzers are directly integrated in `csat`'s source code. We designed the integration of analysis tools and build systems in a modular way, to make it easy to add new tools to the analysis workflow.

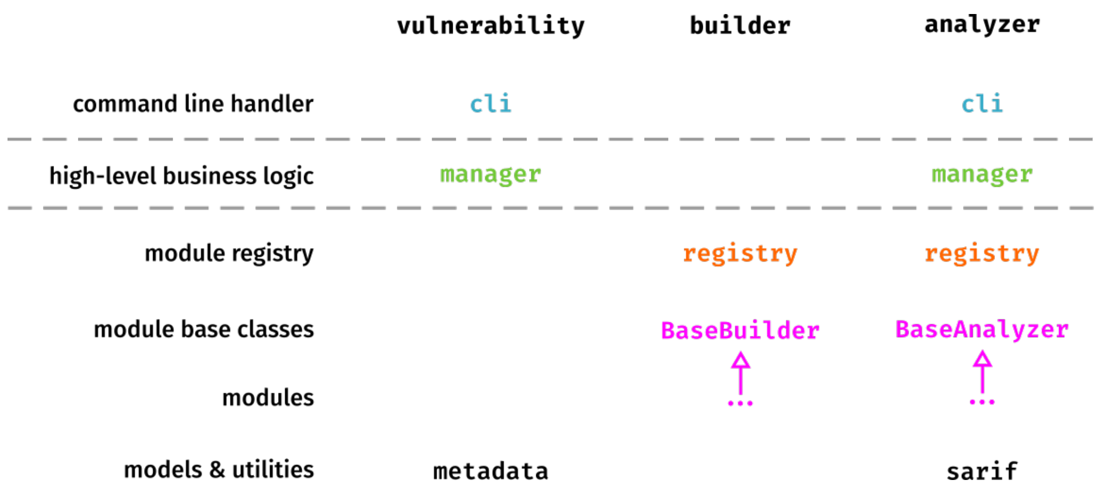


Figure 4.1.: CSAT architecture overview, each column represents one Python module

An overview of CSAT's architecture can be seen in Figure 4.1. The application consists of three main Python packages, each concerned with a separate aspect of the workflow. The diagram shows them as columns. The `vulnerability` package contains the metadata definition and handles its validation. It also manages the creation of new vulnerability containers and provides functionality to search for existing containers. The `builder` package contains the build systems and handles the build process. The analysis tool modules are part of `analyzer` package, which also manages the analysis process.

The three packages are similarly structured: Handling of the command line interface is performed in a `cli` subpackage, while the high-level business logic used by the CLI is contained in a `manager` package. The diagram separates these from the other modules with horizontal lines. The module system for build tools and analyzers consists of a registry and a base class for each type of module, in addition to the modules themselves.

We use the Click[1] library to create CSAT's command line interface (CLI), as Click makes it easy to create user-friendly CLIs, e.g. by automatically generating help pages and providing autocompletion for the available commands. The CLI is the main entry point for the application. The `cli` modules in the `vulnerability` and `analyzer` packages each provide a CLI subcommand and contain code for command parsing and display of the command results.

To interact with the Docker daemon, CSAT uses the *Docker SDK for Python* [32], which provides an abstraction layer for the Docker HTTP API. We created a collection of utility methods for commonly used actions, for example getting the metadata for a specific container. These are contained in the `util.docker` module and are used throughout the application.

[1]https://click.palletsprojects.com

4.2. Metadata Schema & Validation

After establishing the contents of our vulnerability metadata, as described in Section 3.1, selection of an interchange format is required. As the metadata for new vulnerabilities will be entered by humans, user experience is important. Therefore, validation of the user-entered data is key, as it provides direct feedback about correctness and completeness to the user. We choose to use *YAML Ain't Markup Language* (YAML) [33] as the format to provide metadata in. YAML is a "human-friendly data serialization language", and a simple way to provide structured data using indentation. YAML is already used as a configuration language in popular projects, such as Ansible[2], docker-compose[3] or Kubernetes.

For checking the user-provided data, we first evaluated StrictYAML [34], a YAML parser for Python that supports a restricted subset of the YAML specification and provides type-safety and schema validation. However, StrictYAML provides no way to export the schema for use with other software. In the future, other tools might want to interoperate with the CSAT metadata format, so an interoperable metadata specification is required. Therefore, we concluded that StrictYAML does not fit our requirements.

After researching other options, focusing on schema export, we settled on Pydantic [35], a Python library for data validation. It allows defining a model as Python classes and also provides functionality to export the defined schema as *JSON Schema* [36], a format for describing the structure of JSON data. This schema can then be used by other projects to generate code or to validate the implementation. The CLI provides a command, which writes the schema to disk as a JSON file. The full schema is also included in the appendix.

4.3. Vulnerability Packaging and Patching

The `vulnerability.manager` package is responsible for the packaging process. The manager will first perform a few checks to catch basic errors, like validating the user-provided `metadata.yml` against the schema and checking if the `src` folder is not empty. If any check fails, the manager raises a corresponding error, which the command line handler uses to display an error message to the user. When all checks passed, the manager will continue with packaging and create the source image, using the `scratch` image as a base: A simple Dockerfile is copied into the source folder, used to build the Docker image and deleted afterwards. We use a Dockerfile on the filesystem to ensure that the user-provided `src` folder is correctly referenced and copied into the image. The metadata is serialized to JSON and attached to the newly created source image as a label. As YAML is a strict subset of JSON [33], no information is lost by serialization.

[2]https://www.ansible.com/
[3]https://docs.docker.com/compose/

```
FROM alpine as patcher
RUN apk add patch
COPY src /src
COPY patch patch
RUN patch -p1 -d /src < patch

FROM scratch
COPY --from=patcher /src /src
```

When the `metadata.yml` indicates that a patched container should be built, CSAT first checks if a patchfile is present in the source folder. This file must contain the source code changes that remove the vulnerability, in a format accepted by the GNU patch utility. To apply the patch to the source code, a multi-stage Docker build is performed using the Dockerfile in Listing 4.3.

First, the GNU patch utility[4] is installed in a temporary container. Then, source code and the patch file are copied into the container, and the patching is performed. The `patch` utility operates directly on the `src` folder and applies the changes described in the patch file. The second `FROM` statement begins a new build stage, starting with an empty base image. The patched source code is then copied from the previous stage. Therefore, both the patched and unpatched container image contain the same filesystem structure and only differ in metadata. The final image does not contain any remnants of the patching process itself.

4.4. Modular Build System

As previously explained, the capability to compile or build source code into binaries is important, as it allows analysis tools to examine the software during runtime. CSAT employs a modular build system and a central registry for builders, to ensure that new build tools can be easily integrated.

Each build system integrated into CSAT has a corresponding wrapper class in the `builder` package, which provides an abstraction layer and a common API for the rest of the application.

4.4.1. Builder Modules

All build systems are collected in the `builder` package and inherit from the `BaseBuilder` abstract base class. An excerpt from this class can be seen in Listing 4.4.1. It defines a list of attributes that each build tool module must provide and implements a `build` method, which utilizes these attributes to create a new Docker image. This implementation is reused in the inheriting classes, so new builders can be added just by declaring attributes.

[4] https://savannah.gnu.org/projects/patch/

The id is a unique identifier for the build system and used e.g. in the vulnerability metadata. build_image and build_steps specify the base image and the Dockerfile instructions for building, while run_image and run_steps provide the same information for building the execution environment. The run_command attribute contains a single command used for starting the application and is later used as the container entrypoint.

```
class BaseBuilder(ABC):
  id: str
  build_image: str
  build_steps: str
  run_image: str
  run_steps: str
  run_command: str
  artifact_type: ArtifactTypeEnum

  def build(self, src_tag: str, dest_tag: str) -> Image:
    ...
```

4.4.2. Builder Dockerfile

The build process utilizes a multi-stage Dockerfile, generated using a template, which can be seen in Listing 4.4.2. The template variables are filled using the attributes of the respective builder class. The Dockerfile first declares a new build stage src using the source container of the vulnerability. In the second builder stage, the build steps are executed. All build artifacts must be placed into the /build directory during the build process. The /build directory is then copied, together with the source code, into the final container in the last stage. The resulting container now has a /src folder with source code and a /build folder with the build artifacts. The multi-stage build ensures that no unwanted remnants of the build process are packaged into the final image.

```
FROM {{ src_image }} as src

FROM {{ build_image }} as builder
COPY --from=src /src /src
{{ build_steps }}

FROM {{ run_image }}
COPY --from=builder /src /src
COPY --from=builder /build /build
WORKDIR /build
{{ run_steps }}
CMD ["{{ run_command | join ('", "') }}"]
```

4.4.3. Builder Registry

To enable discovery of newly added builder modules, we make use of a central registry. Each build tool module has a unique identifier, which is used in vulnerability metadata and user interactions. The registry provides a mapping from a builder ID to its class and discovers all available builders automatically.

To achieve this, the registry walks all sub-packages of the `builder` package and imports them, as can be seen in Listing 4.4.3. Importing a Python package executes its `__init__.py` initialization code, which in turn calls the registries `register_builder` method, to make it aware of the builder module. New builders can therefore be easily added by creating a new package with a builder class and a call to `register_builder` in the package initialization code.

```
def setup_registry() -> None:
  for loader, module_name, is_pkg in pkgutil.walk_packages(
    [os.path.dirname(__file__)]
  ):
    if is_pkg:
      importlib.import_module(f".{module_name}", __package__)
  global setup_done
  setup_done = True
```

After being set up, the registry provides methods to query for existence of a builder module by ID and to fetch a builder class.

4.5. Analysis Tool Integration

Analysis tools examine software and produce a list of findings. This analysis can occur on source code or during runtime. CSAT uses a similar architecture for analyzers as for build tools: All analyzers are wrapped in a class, which inherits from a common `BaseAnalyzer` abstract base class. The base class contains common functionality, so that simple analyzers can be added just by overriding a number of variables. The `analyzer` module of the application also contains a registry for the available analyzers.

```
class BaseAnalyzer(ABC):
    id: str
    requires_build_changes: bool
    compatible_builders: list[str]
    compatible_artifact_types: list[str]
    build_steps: str
    analyzer_image: str
    analyzer_command = ""

    def source_tag(self) -> str:
        ...

    def __init__(self, vulnerability_name: str, patched_source: bool):
        self.source_vulnerability = vulnerability_name
        self.patched_source = patched_source
        self.tag = f"csat/{vulnerability_name}/analyze-{self.id}"

    def build_analyze_image(self):
        ...

    def parse_container_log(self, log: str) -> AnalysisRunResult:
        return AnalysisRunResult(sarif=SarifLog.parse_raw(log))

    def analyze(self) -> AnalysisRunResult:
        self.build_analyze_image()
        client = get_docker_client()
        container = client.containers.run(self.tag, self.analyzer_command,
            stdout=True, stderr=True, detach=True)
        ...
        container_log_str = str(container.logs(), "utf-8")
        return self.parse_container_log(container_log_str)
```

As can be seen in Listing 4.5, the `BaseAnalyzer` declares a list of required properties that the inheriting classes must define for their respective specific analyzer. Among these attributes are a unique identifier used in the registry and a flag that indicates whether the analyzer requires changes to the build process. An analyzer must also specify a list of compatible build systems and artifact types, which are used for the checks we laid out in the previous chapter. The `analyze` function is called by the analysis manager and utilizes the other functions to create a new Docker image, execute the analyzer and parse the results. `build_analyze_image` and `parse_container_log` can be individually reimplemented by the wrapper classes, if the default functionality of the `BaseAnalyzer` is insufficient. This may be the case when the analysis tool does not output its result as SARIF JSON and further processing of its output is therefore required.

4.6. Analysis Automation & Result Verification

The bulk of the analysis process itself is handled by the `analyzer.manager` package. Only comparing the analysis results with the expected values is performed directly by the command line handler, as this task involves frequent printing of results to the command line.

The complete analysis workflow consists of three parts, which are discussed in the following sections. An overview is also displayed in Figure 4.2.

4.6.1. Preparation

First, CSAT prepares its internal registries of the available builders and analyzers and checks if the requested analyzer exists. The builder of the analyzed software is fetched and the compatibility of its artifact with the analyzer is verified. The compatibility check utilizes vulnerability and analyzer metadata. Each analyzer declares compatibility with a list of artifact types, like Java class files or JARs. If the vulnerable software uses a build system that does not produce a compatible artifact, the analysis process cannot proceed and is aborted early. When the analyzer requires changes to the build process, we also check if it is compatible with the vulnerability's build tooling.

4.6.2. Analysis

The manager instantiates the analyzer class and calls its `analyze` method. The next steps depend on the implementation of this method in the analyzer class. Some analysis tools require changes to the default build process, for example to add instrumentation to the source code. CSAT provides two ways to accommodate such changes: The build process can be influenced by the analyzer using hooks, which allow common changes, like adding a new dependency to a Maven project. Alternatively, the build process can be completely replaced by a custom implementation provided by the analysis tool module. When an analyzer needs to change the build process, the analysis manager also performs an additional check: it verifies if the analyzer is compatible with the build system of the vulnerability, i.e., that the required build changes are possible and implemented.

The default implementation of the analysis process first ensures that the required vulnerability image is present. Depending on the type of analysis, a source or binary image may be required. If the binary image is missing, it will be built automatically. Next, the analysis image is created by copying the `/src` (and if required `/build`) folder into a new image based on the analyzer image.

Finally, the analysis itself is executed inside a container based on the image. If the analysis tool does not output SARIF JSON directly, its output is now parsed and converted to SARIF. If a patched variant of the vulnerability exists, the analysis is repeated once more, using the patched image.

4.6.3. Result Verification

The analyzer manager returns an `AnalysisResult`, which contains vulnerability metadata and the SARIF data from the unpatched and patched analysis runs. The command line handler then first prints out the detected vulnerabilities. If both a patched and unpatched run was performed, both runs are then compared, expecting the patching to have decreased the number of detected issues. Finally, the detected location is compared with the expected one from metadata. To enable further processing of analysis results, they can also be saved to disk as a JSON file, which also includes the complete SARIF data from the analyzer.

After comparing the analysis results with the metadata, CSAT reports one of five outcomes:

- **True Positive:** The analysis tool reported (at least) the expected vulnerability at the expected location.
- **False Positive:** The analysis tool reported only unexpected vulnerabilities or vulnerabilities at the wrong code location.
- **True Negative:** The analysis tool did not report any vulnerabilities and none were expected. This can happen if a specifically non-vulnerable software was packaged.
- **False Negative:** The analysis tool did not report any issues, even though one was expected.
- **Error:** The analysis did not complete successfully.

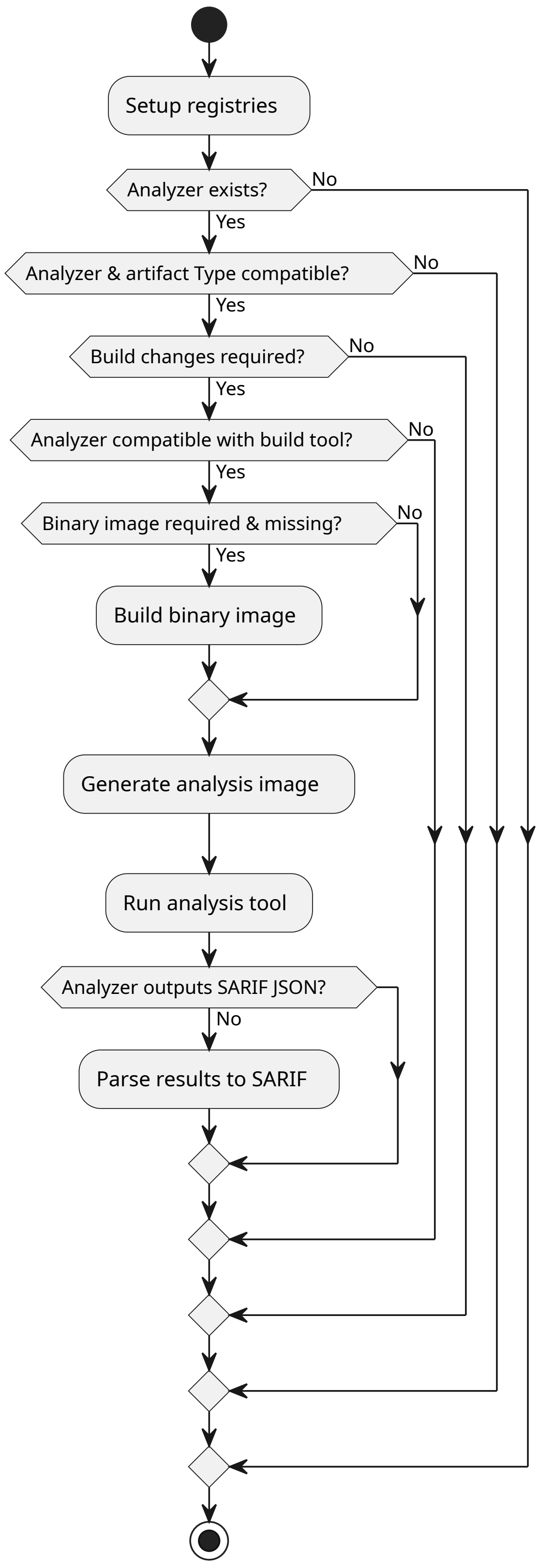


Figure 4.2.: Sequence Diagram of CSAT's Vulnerability Analysis Workflow

5. User Interface

Having explained the *internal* architecture of our command line interface, this chapter focuses on the *external* user interface. Users interact with CSAT in two ways: developers of analysis tools add new analyzers as Python classes using the module system. New build tools are added similarly. In contrast, users packaging vulnerabilities or performing analyses utilize the CLI and do not interact with Python code.

5.1. Adding Analysis & Build Tools

CSAT uses a module system for analyzers and build tools, as we laid out in Chapter 4. Each tool is wrapped in a Python class, which inherits from a corresponding abstract base class. A registry automatically detects all currently existing modules.

This simplifies integration of new tools. Although it is required to write Python code to add a new analyzer or build tool, in many cases, only the declaration of variables is necessary.

```
class SpotbugsAnalyzer(BaseAnalyzer):
  id = "spotbugs"
  logger = logging.getLogger("csat.analyzer.spotbugs")
  requires_build_changes = False
  compatible_artifact_types = [ArtifactTypeEnum.jar,
 ↪  ArtifactTypeEnum.class_files]
  analyzer_image = "docker.io/openjdk:11-jre"
  analyzer_command = "spotbugs -effort:max -sarif /build/"
  build_steps = """RUN cd / \
    && curl
 ↪  https://repo.maven.apache.org/maven2/com/github/spotbugs/spotbugs/4.5.3/
 ↪  spotbugs-4.5.3.tgz -L -o spotbugs.tgz \
    && tar xvzf spotbugs.tgz \
    && rm spotbugs.tgz \
    && chmod +x spotbugs-4.5.3/bin/spotbugs \
    && update-alternatives --install /usr/bin/spotbugs spotbugs
 ↪  /spotbugs-4.5.3/bin/spotbugs 1"""
```

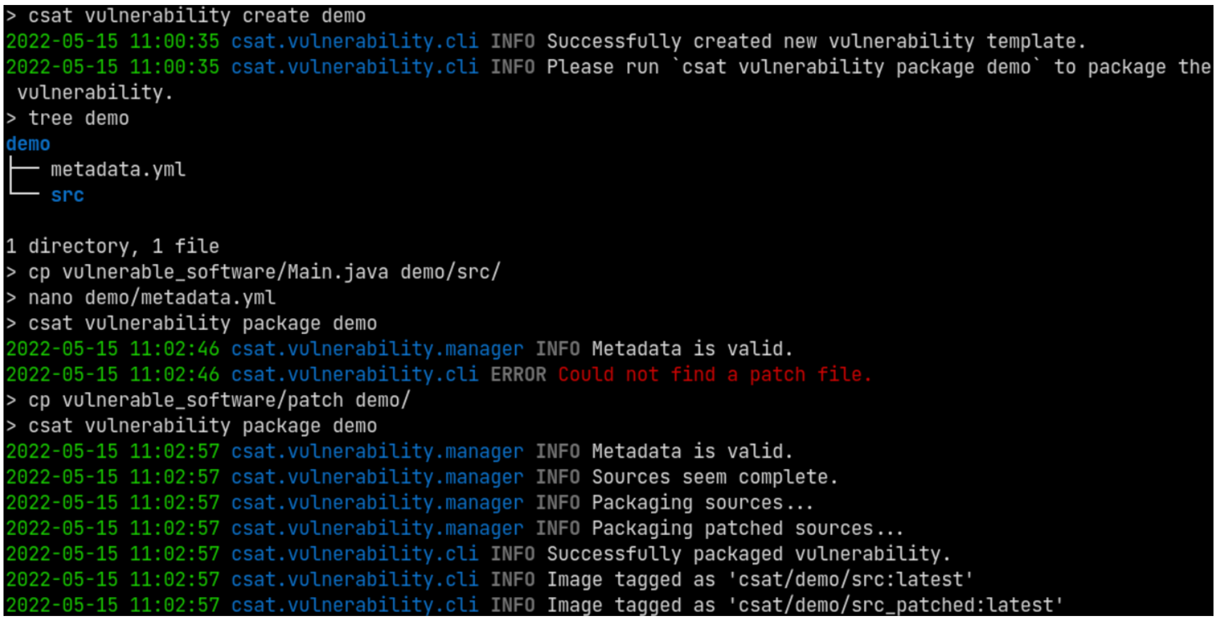
Listing 5.1 shows the wrapper class for the SpotBugs analyzer as an example. It sets the analyzer's identifier, declares that SpotBugs does not require changes to the build process and that it is compatible with both JARs and Java class files. The class also sets the Docker image used for execution and the command to run the analyzer. Finally, it includes the Docker build steps to set up SpotBugs inside the Docker image.

More complex cases can also be accommodated. For example, when the default analysis process as implemented in the `BaseAnalyzer` class is unsuitable for an analysis tool, the corresponding method can be overriden and the implementation replaced by a custom version.

Our modular system therefore provides both ease of use and flexibility, allowing people familiar with a build tool or analyzer to quickly integrate it into CSAT.

5.2. Vulnerability Packaging

When a new vulnerability is to be packaged, the user first utilizes the CLI to create an empty vulnerability template. Then the source files and metadata are added and finally the container image is created using the CLI.



```
> csat vulnerability create demo
2022-05-15 11:00:35 csat.vulnerability.cli INFO Successfully created new vulnerability template.
2022-05-15 11:00:35 csat.vulnerability.cli INFO Please run `csat vulnerability package demo` to package the
 vulnerability.
> tree demo
demo
├── metadata.yml
└── src

1 directory, 1 file
> cp vulnerable_software/Main.java demo/src/
> nano demo/metadata.yml
> csat vulnerability package demo
2022-05-15 11:02:46 csat.vulnerability.manager INFO Metadata is valid.
2022-05-15 11:02:46 csat.vulnerability.cli ERROR Could not find a patch file.
> cp vulnerable_software/patch demo/
> csat vulnerability package demo
2022-05-15 11:02:57 csat.vulnerability.manager INFO Metadata is valid.
2022-05-15 11:02:57 csat.vulnerability.manager INFO Sources seem complete.
2022-05-15 11:02:57 csat.vulnerability.manager INFO Packaging sources...
2022-05-15 11:02:57 csat.vulnerability.manager INFO Packaging patched sources...
2022-05-15 11:02:57 csat.vulnerability.cli INFO Successfully packaged vulnerability.
2022-05-15 11:02:57 csat.vulnerability.cli INFO Image tagged as 'csat/demo/src:latest'
2022-05-15 11:02:57 csat.vulnerability.cli INFO Image tagged as 'csat/demo/src_patched:latest'
```

Figure 5.1.: Example of packaging a new vulnerability with the CSAT CLI

Figure 5.1 shows such a packaging process. First, a vulnerability template called `demo` is created using the `vulnerability create` subcommand. CSAT creates a new folder, containing a `metadata.yml` template and an empty `src` directory. We already presented the contents of the metadata template in Listing 3.1. It includes optional properties with a corresponding annotation, making it easy for a user to provide the most detailed data possible.

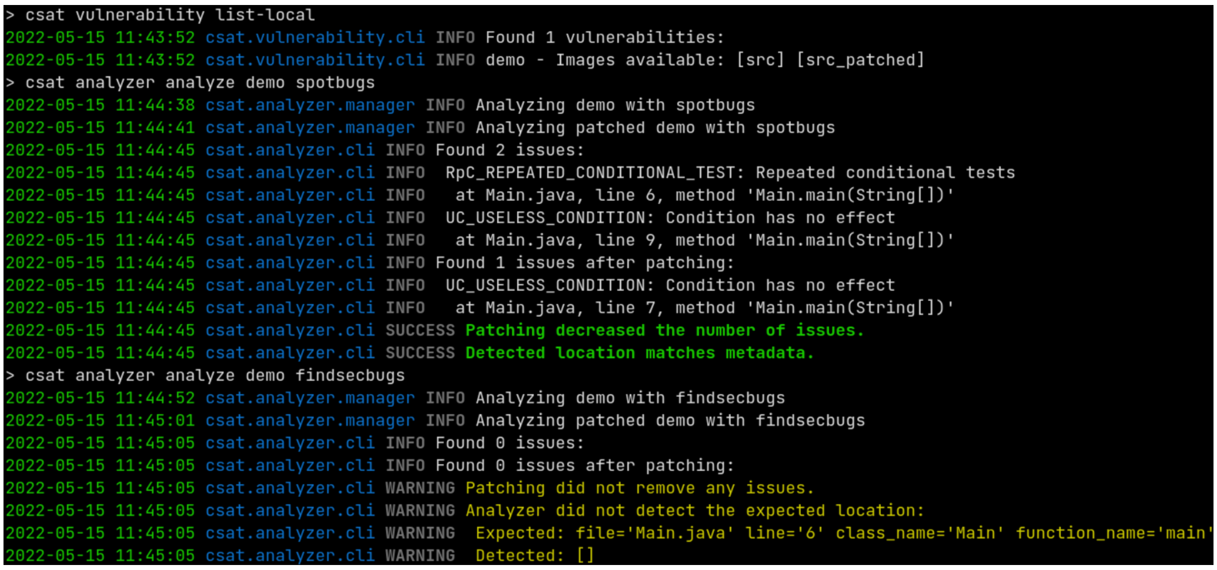
After adding a source code file and editing the metadata, our example continues with a first attempt of packaging the new vulnerability. CSAT detects that the metadata specifies a patch, but no patch file is present and aborts the packaging process. A similar error would occur if the metadata does not match the schema or no source code files are present.

After the missing patch has been added to the vulnerability directory, packaging succeeds. CSAT creates a new source image for the demo vulnerability. As a patch was provided, a second image with the patched source code is also generated.

The CSAT command line interface user-friendly, as it validates the entered metadata and provides immediate feedback to users. It also checks whether all expected files are present. To package vulnerabilities, users do not need to directly interact with Dockerfiles or the Docker command line interface, as CSAT automatically generates a Dockerfile for packaging and handles creation of the container image.

5.3. Analyzing Vulnerabilities

The command line interface supports users not only in packaging, but also in analyzing vulnerabilities, as shown in the example in Figure 5.2. It provides a command to list all locally available images, which also shows if a patched variant of a vulnerability is available.



```
> csat vulnerability list-local
2022-05-15 11:43:52 csat.vulnerability.cli INFO Found 1 vulnerabilities:
2022-05-15 11:43:52 csat.vulnerability.cli INFO demo - Images available: [src] [src_patched]
> csat analyzer analyze demo spotbugs
2022-05-15 11:44:38 csat.analyzer.manager INFO Analyzing demo with spotbugs
2022-05-15 11:44:41 csat.analyzer.manager INFO Analyzing patched demo with spotbugs
2022-05-15 11:44:45 csat.analyzer.cli INFO Found 2 issues:
2022-05-15 11:44:45 csat.analyzer.cli INFO  RpC_REPEATED_CONDITIONAL_TEST: Repeated conditional tests
2022-05-15 11:44:45 csat.analyzer.cli INFO   at Main.java, line 6, method 'Main.main(String[])'
2022-05-15 11:44:45 csat.analyzer.cli INFO  UC_USELESS_CONDITION: Condition has no effect
2022-05-15 11:44:45 csat.analyzer.cli INFO   at Main.java, line 9, method 'Main.main(String[])'
2022-05-15 11:44:45 csat.analyzer.cli INFO Found 1 issues after patching:
2022-05-15 11:44:45 csat.analyzer.cli INFO  UC_USELESS_CONDITION: Condition has no effect
2022-05-15 11:44:45 csat.analyzer.cli INFO   at Main.java, line 7, method 'Main.main(String[])'
2022-05-15 11:44:45 csat.analyzer.cli SUCCESS Patching decreased the number of issues.
2022-05-15 11:44:45 csat.analyzer.cli SUCCESS Detected location matches metadata.
> csat analyzer analyze demo findsecbugs
2022-05-15 11:44:52 csat.analyzer.manager INFO Analyzing demo with findsecbugs
2022-05-15 11:45:01 csat.analyzer.manager INFO Analyzing patched demo with findsecbugs
2022-05-15 11:45:05 csat.analyzer.cli INFO Found 0 issues:
2022-05-15 11:45:05 csat.analyzer.cli INFO Found 0 issues after patching:
2022-05-15 11:45:05 csat.analyzer.cli WARNING Patching did not remove any issues.
2022-05-15 11:45:05 csat.analyzer.cli WARNING Analyzer did not detect the expected location:
2022-05-15 11:45:05 csat.analyzer.cli WARNING  Expected: file='Main.java' line='6' class_name='Main' function_name='main'
2022-05-15 11:45:05 csat.analyzer.cli WARNING  Detected: []
```

Figure 5.2.: Analyzing a vulnerability with the CSAT CLI

In the example, the previously packaged `demo` vulnerability is first analyzed using SpotBugs. The analyzer detects two issues in the source code, but only one in the patched container. As the patch only removes the specific packaged vulnerability and SpotBugs detected it in the unpatched analysis run, CSAT concludes that the detection is no false-positive. The command line interface summarizes this with a corresponding message in green text. For each issue, a description and the exact location is also printed. As the location detected by SpotBugs matches the metadata, the CLI also reports success.

The same vulnerability is then analyzed with find-sec-bugs. This tool does not detect the type of issues present in the demo source code, so zero issues are reported. The CLI therefore warns the user in yellow text, printing the expected and the detected source code location.

As evident in the examples, our command line interface frequently uses color to increase readability. Unexpected events are colored in red or yellow, while successful events are printed in green. All console messages also include the message type as text (like `WARNING` or `SUCCESS`), to be readable by color-blind users.

6. Module Integration

The previous chapters presented our metadata format, the architecture of CSAT and its user interface. This chapter will show how we integrated various build tools and analyzers into CSAT.

6.1. Build Tool Integration

For the thesis, we integrated two Java build tools into CSAT. We chose Java, because multiple analysis tools exist for this language, making it a good candidate for testing our tool with build systems and analyzers of varying complexity. Additionally, Jaint, one of the analyzers we planned to integrate is designed for Java.

First, we added the standard `javac` Java compiler from OpenJDK. The build process we chose to implement is very simple: It consists of creating the target `/build` directory and then running `javac *.java && cp *.class /build` in the container. This compiles all `.java` files inside the `/src` directory and copies the resulting `.class` files to `/build`. Our implementation only allows compiling basic Java projects, as all files must be in the same top-level directory and no libraries can be used. Nevertheless, it allowed testing of CSAT's architecture during development and served as a starting point for integrating more complex tools.

For a more real-world scenario, we also implemented a module for the Maven build system. Maven allows for more complicated Java projects, which, for example, utilize external dependencies and may create a variety of different artifact types. Our integration currently only supports JARs. Analysis tool wrapper classes can provide means to influence the build process by implementing additional methods, which manipulate the internal state of a class instance. Our Maven module demonstrates this aspect of CSAT, as additional Maven plugins can be added to the build process. This is facilitated by the `add_plugin` method of the `MavenBuilder` class. Each invocation of this method adds a new build step is added to the Dockerfile. These steps edit the `pom.xml` Maven configuration file using `xmlstarlet` to add the plugin.

6.2. Analysis Tool Integration

We integrated four analyzers, SpotBugs for plain class files and JARs, SpotBugs for Maven, find-sec-bugs and Jaint, during the implementation phase of this thesis.

The following sections discuss these analysis tools and the complexity of their integration in more detail.

6.2.1. SpotBugs

SpotBugs [37] is a static analyzer for Java, available as plugins for build tools like Maven or Ant and as a standalone utility. Both the Maven plugin and the standalone version were integrated into CSAT.

The standalone version served as the first analyzer to be integrated into CSAT. The SpotBugs CLI operates on `.class` files or JARs and can output its analysis results as SARIF JSON. This made integration straightforward, as the `javac` builder produces `.class` files and SARIF JSON parsing is already handled by the base analyzer class. The `SpotbugsAnalyzer` class therefore only declares variables and does not need to override any functions.

```
class SpotbugsMavenAnalyzer(BaseAnalyzer):
  ...
  requires_build_changes = True
  compatible_builders = [MavenBuilder.id]
  compatible_artifact_types = [ArtifactTypeEnum.jar]

  analyzer_image = "docker.io/maven:3-openjdk-11"
  analyzer_command = "bash -c 'cd /src && mvn -q -Dspotbugs.sarifOutput=true
↪   spotbugs:spotbugs && cat target/spotbugsSarif.json'"

  def build_analyze_image(self):
    metadata = get_src_metadata(self.source_tag)
    builder = cast(MavenBuilder, registry.get_builder("maven")(metadata))
    builder.run_steps = self.build_steps
    builder.run_image = self.analyzer_image
    builder.add_plugin("com.github.spotbugs", "spotbugs-maven-plugin",
↪   "4.5.2.0")
    builder.build(self.source_tag, self.tag)
```

The SpotBugs Maven analyzer demonstrates the ability to change the build process by adding itself as a plugin. It overrides the `build_analysis_image` method to build a modified analysis image with the Maven builder, as seen in Listing 6.2.1. Using the `add_plugin` method, the SpotBugs plugin is added. All other analysis steps are performed by the implementation in the `BaseAnalyzer` class.

The Maven plugin of SpotBugs does not support outputting the SARIF result directly to standard output, so a workaround is employed. The `analyzer_command` first executes the build process with Maven, but suppresses all output. After the build is completed, the JSON file written by the SpotBugs plugin is printed to standard out using `cat`. This allows us to reuse the existing infrastructure that extracts the result from the container output stream.

The two SpotBugs analyzers were the first we implemented and took the longest amount of time, as they served as test-cases for various aspects of the application architecture, which was developed simultaneously. The final wrapper class for SpotBugs operating on plain class files or JARs was already presented in Listing 5.1 and is just 22 lines in length, including comments.

6.2.2. find-sec-bugs

Find-sec-bugs or *Find Security Bugs* [38] is a SpotBugs plugin that focuses on detecting security issues. It was integrated only as a standalone plugin supporting `.class` files and JARs, in a very similar way as the basic SpotBugs analyzer. The only difference is in log parsing, as find-sec-bugs prefixes the SARIF output with non-JSON data, which has to be removed.

The wrapper class for find-sec-bugs has a similar length as the SpotBugs wrapper, but implements additional logic to parse the log output of the analyzer. This is required, as find-sec-bugs prints additional output before the SARIF JSON, which has to be removed. We estimate the expenditure of time for implementing this analyzer to about 30 to 45 minutes, including the parsing logic and creation of the required Docker build steps.

6.2.3. Jaint

Jaint [39] is a framework for dynamic taint analysis of Java web applications. It utilizes *JDart* [40], a dynamic symbolic execution engine, itself based on *Java PathFinder* (JPF) [41], a Java Virtual Machine. We integrated Jaint based on the artifact from [39].

Jaint uses a domain-specific language (DSL), built using the Meta Programming System[1] (MPS), to define the security issues or taints it can detect. A code generator is used to create Java code per defined taint, which then monitors the data flow according to the constraints specified using the DSL. The artifact contains taint specifications for issues encountered in the OWASP benchmark. By default, the output of Jaint does not contain sufficient information about the type and the location of the detected taint. Therefore, we modified the DSL code generator in MPS to include this information.

The version of Jaint from [39] is limited in the set of Java applications that can be successfully analyzed. As we mentioned in Section 2.2, taint tracking observes the flow of data through an applica-

[1]https://www.jetbrains.com/mps/

tion. When an application utilizes library code that is not taint-aware, data flow cannot be tracked correctly. Therefore, libraries must be (partly) replaced with a version that supports taint analysis, a process called mocking.

Jaint also needs a test harness for analyzed applications, which instantiates classes and executes the functions under test. The artifact only provides mock implementations and a test harness sufficient to analyze vulnerable code snippets from the OWASP benchmark, other applications produce compilation errors. It may be possible to expand Jaint's mocking to support more generic vulnerabilities, but this is out of scope for this thesis.

Little documentation exists about the process to build Jaint itself, only a number of shell scripts are available in the artifact. The build requires multiple JDK versions and both the Ant and Maven build tools to generate the taint description using MPS and compile mocks, JPF and Jaint itself. Based on the code from the artifact, we created a fork of Jaint, which includes a newly written Dockerfile and the aforementioned modifications to the taint DSL code generator. Using the Dockerfile, we create the Jaint Docker image used by CSAT.

The OWASP snippets from the artifact are not compilable without mocks from Jaint. This requires special handling inside CSAT, as the build manager normally tries to compile a vulnerability standalone. CSAT cannot create binary containers from these OWASP benchmark tests, and the Jaint analyzer module implements its own `build_analyze_image` function to compile OWASP tests with Jaint mocks.

To package all 2740 OWASP code snippets from the artifact that Jaint is able to analyze, we created an additional command line tool, `owasp2csat`. It iterates over the OWASP vulnerabilities and parses the accompanying XML files, which contain information about the vulnerability type. It then creates a new source container per vulnerability, which can then be used by CSAT. Not all snippets contain vulnerable code, some serve to check that Jaint does not produce false-positives, a scenario that CSAT's metadata also supports.

Jaint was the final analyzer we integrated. Its integration was hampered by missing documentation, lacking details in the analysis output and a complicated build process for Jaint and its dependencies, requiring multiple different build tools and Java versions. The integration process therefore took multiple days, as a Docker image for Jaint had to be created and the log output had to be extended to yield useful information about the bug location. Much less effort was needed for the wrapper class in CSAT itself. The most amount of work, amounting to 66 of the total 118 lines of code, was invested in log parsing, which converts Jaint's output into a SARIF data structure. The wrapper class also overrides parts of the implementation of the base class, as Jaint and the vulnerabilities it analyzes require special build steps to account for the required mocking.

7. Evaluation

In this chapter, we show the results of integrating various build and analysis tools into CSAT. We also discuss our work in light of the two research questions we posed in Section 1.3.

7.1. Build Tool Integration

We added modules for two Java build tools to CSAT as part of this thesis. This work was performed during experimentation with the architecture of CSAT's command line tool, so no useful record of the required implementation time exists. The wrapper classes encompass less than 50 lines of Python code, and we estimate that new build tools will require a comparable amount of code.

As arbitrary commands can be executed during the build process, implementation complexity is not dependent on the programming language that is compiled. For example, the execution of the `javac` Java compiler can be replaced with `./configure && make` for a conceivable integration of the *GNU Build System*[1].

7.2. Analysis Tool Integration

Our integration of four analysis tools provides important data about the complexity of adding further analyzers to CSAT. While being somewhat subjective, we maintain that integration time and code size are still a good indicator for the required amount of work.

"Simple" analysis tools, which work inside the constraints of CSAT's default build and analysis steps and directly output SARIF JSON, do not require any custom Python code. Integration requires only the declaration of a number of variables, about 20 lines of code. Users who are already familiar with such tools can therefore easily add them. We expect the creation of a new module for such a tool to only require about 30 minutes.

We must note that this estimation assumes that Docker images are already available for the analyzer. If this is not the case, creating a Docker image may add significant work. An example for this scenario

[1]https://www.gnu.org/software/automake/manual/html_node/GNU-Build-System.html

was Jaint, were we had to develop a Dockerfile ourselves. This and other required additional work made the integration of Jaint take significantly more time when compared to the other analyzers.

We believe such long integration times to be the exception, as developers of new analysis tools are intimately familiar with the workings of their analyzer. Creating a Docker container for the analyzer will therefore be easier. New tools might also be developed with the SARIF interchange format in mind, which omits the necessity of implementing log parsing in CSAT.

We conclude that the module system of CSAT's command line interface makes it easy to integrate many analysis tools. Some tools may require more effort, for example if they do not output SARIF JSON directly. Still, CSAT provides enough flexibility to allow their integration.

7.3. Vulnerability Metadata and Packaging

The first research question we formulated in Section 1.3 is "What metadata is required to effectively evaluate security analysis tools?". Our approach, as laid out in Section 3.1 answers this question. The CSAT metadata format allows for automated compilation of vulnerable software by providing information about the build system. It also allows for automated evaluation of the analysis result, using information about the bug type and its location in the source code. We demonstrated the feasibility of this by implementing the `csat` command line utility. The metadata specification is defined using the pydantic library and exportable as in the JSON Schema format. This allows other software to generate and validate CSAT metadata and interoperate with our vulnerability packages.

We evaluated the suitability of our metadata format by packaging all vulnerabilities of the OWASP test suite for use with Jaint. Our `owasp2csat` command line tool demonstrates that conversion of existing collections of vulnerable software to CSAT containers is possible with little effort.

We also conclude that our YAML-based metadata format is significantly more user-friendly than other approaches with CSV or XML. Our validation system also provides direct feedback to users about errors in their metadata.

7.4. Analysis Infrastructure

Our second research question is concerned with the infrastructure required for automated evaluation of security analysis tools. CSAT provides such infrastructure, using vulnerability metadata to judge analysis results. The CSAT command line utility enables packaging of vulnerabilities into container images and automates the analysis process. A module system for both build tools and analyzers simplifies the integration of new tools into CSAT.

When a patch removing the vulnerability is available, CSAT automatically applies the patch and runs a second analysis, to compare whether the analyzer does not report a vulnerability for the patched variant. This helps to sort out false-positive findings.

Our implementation provides host isolation by using Docker containers, as we laid out in Section 2.1. This allows for reproducible build and analysis environments and sidesteps dependency problems. Industry-standard Docker images also enable sharing of vulnerabilities using already existing container registries like Docker Hub.

7.5. Reproduction of Vulnerability Detection with Jaint

In [39], Mues et al. presented the Jaint taint-analysis framework and evaluated it using the OWASP Benchmark. The authors used their own tooling for this evaluation and showed that Jaint successfully detects 100% of vulnerabilities in the benchmark. Using the published artifact containing the analysis framework and the used benchmark tests, we reproduced the results of the paper.

As described in Section 6.2.3, we integrated a Jaint module into CSAT and implemented the `owasp2csat` tool to convert the OWASP tests to CSAT images. After performing the conversion process, we utilized the `csat` command line tool to run Jaint against all tests. Our tool was able to successfully automate the analysis process for all 2740 containers. However, the command line interface does not yet provide a summarized report for bulk analysis runs. Still, we were able to manually verify the results and concluded that Jaint achieves the same precision when run with CSAT as in the original paper. Therefore, we demonstrated that CSAT can be integrated with a complicated analysis tool and is useful for evaluation.

8. Conclusion

This thesis introduced the increasing importance of security analysis tools and laid out the challenges encountered when testing such tools. The first chapter showed that existing bug and vulnerability databases are not suitable for evaluation of security analyzers, as they lack the required information and are not automated.

We presented our approach for a new solution, the *Containers for Security Analysis Tools* (CSAT) command line utility and metadata format.

The metadata laid out in Section 3.1 provides all required information to automatically compile source code, making it available for analysis at runtime. It also includes details about the vulnerability type and location in code, enabling the CSAT command line utility to automatically determine whether an analysis tool found the correct issue. Our metadata design therefore answers research question one.

We utilize Docker images to package and exchange vulnerable software including metadata. By executing the analysis process in Docker containers, we achieve host isolation. Dependencies can be installed independently of the host system and other containers in a reproducible environment that avoids dependency problems.

The CSAT CLI supports users in packaging new vulnerabilities and automates the build and analysis process. Sections 5.2 and 5.3 showed our focus on ease of use in the CLI design. When a patch is available, CSAT automatically applies it to the vulnerable source code to generate a second, non-vulnerable variant of the vulnerability package.

CSAT's architecture allows to integrate build tools and analyzers on a modular basis. As presented in Chapter 6, we integrated two build tools and four analyzer variants as part of this thesis, demonstrating that adding new analysis tools is simple and time-efficient. CSAT is expandable and provides enough flexibility to integrate tools that require special handling, like Jaint. In conjunction with the metadata format, the CSAT command line interface is able to automatically evaluate security analysis tools, therefore answering research question two.

Finally, we were able to reproduce the results of Mues et al., using CSAT to analyze thousands of OWASP benchmark tests with Jaint. This shows that CSAT is usable in practical scenarios.

8.1. Limits and Future Work

Due to time constraints, our implementation of CSAT has certain limits, specifically its command line tool.

We only added Java-based build tools and analyzers, but do not expect CSAT's architecture to limit the integration of tools for other programming languages. As Docker is used to execute a tool, as long as it can run inside a Docker container, integration is possible. The list of supported languages is currently hardcoded, but can be easily expanded.

The Docker images containing vulnerabilities can be exchanged using container registries like Docker Hub. These registries typically do not provide direct access to the labels of the hosted images, but CSAT's metadata is stored in a label. Therefore, it is currently not possible to search for vulnerabilities matching certain criteria (like a specific programming language or build system) without first downloading at least the metadata of all images. A separate database collecting all available CSAT-compatible vulnerabilities in container registries would be required to provide such a search feature.

The reporting features of CSAT's command line interface are currently rather simple. We only implemented human-readable log output and export of the results of a single analysis run. The export encompasses analyzing one vulnerability with one analyzer, including the patched variant if available. Future work might expand the output features to, for example, include statistics comparing multiple analysis tools over a collection of vulnerabilities. Still, we were able to successfully replicate the results of Jaint's paper, even though the analysis results had to be manually tallied.

In Section 1.2.1 we compared various existing vulnerability databases. Creating tools to import the software already collected in these databases could prove valuable, as it makes hundreds of vulnerabilities available to the analysis tools integrated into CSAT. However, most databases do not include information about the bug location, rendering CSAT unable to decide if the correct vulnerability was detected.

When CSAT's abilities regarding search and reporting are expanded, we envision it to be suitable for use as a packaging format for newly developed vulnerability collections or as an automation tool in competitions such as SV-COMP.

9. References

[1] S. Özkan, “Browse cve vulnerabilities by date. CVE details.” [Online]. Available: https://www.cvedetails.com/browse-by-date.php. [Accessed: 15-Apr-2022]

[2] J. Fan, Y. Li, S. Wang, and T. N. Nguyen, “A c/c++ code vulnerability dataset with code changes and CVE summaries,” in *Proceedings of the 17th international conference on mining software repositories*, 2020, pp. 508–512, doi: 10.1145/3379597.3387501.

[3] G. Nikitopoulos, K. Dritsa, P. Louridas, and D. Mitropoulos, “CrossVul: A cross-language vulnerability dataset with commit data,” in *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2021, pp. 1565–1569, doi: 10.1145/3468264.3473122.

[4] D. Wichers, “OWASP benchmark,” 12-Apr-2022. [Online]. Available: https://github.com/OWASP-Benchmark/BenchmarkJava. [Accessed: 16-Apr-2022]

[5] P. E. Black, “Juliet 1.3 test suite: Changes from 1.2,” National Institute of Standards and Technology, Gaithersburg, MD, NIST TN 1995, Jun. 2018 [Online]. Available: http://nvlpubs.nist.gov/nistpubs/TechnicalNotes/NIST.TN.1995.pdf. [Accessed: 16-Apr-2022]

[6] P. Gyimesi *et al.*, “BUGSJS: A benchmark and taxonomy of JavaScript bugs,” *Software Testing, Verification and Reliability*, vol. 31, no. 4, p. e1751, 2021, doi: 10.1002/stvr.1751.

[7] R. Widyasari *et al.*, “BugsInPy: A database of existing bugs in python programs to enable controlled testing and debugging studies,” in *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2020, pp. 1556–1560, doi: 10.1145/3368089.3417943.

[8] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: A database of existing faults to enable controlled testing studies for java programs,” in *Proceedings of the 2014 international symposium on software testing and analysis*, 2014, pp. 437–440, doi: 10.1145/2610384.2628055.

[9] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, “A manually-curated dataset of fixes to vulnerabilities of open-source software,” in *2019 IEEE/ACM 16th international conference on mining software repositories (MSR)*, 2019, pp. 383–387, doi: 10.1109/MSR.2019.00064.

[10] D. Beyer, “Advances in automatic software testing: Test-comp 2022,” in *Fundamental approaches to software engineering*, 2022, pp. 321–335, doi: 10.1007/978-3-030-99429-7_18.

[11] D. Beyer, “Progress on software verification: SV-COMP 2022,” in *Tools and algorithms for the construction and analysis of systems*, 2022, vol. 13244, pp. 375–402, doi: 10.1007/978-3-030-99527-0_20.

[12] T. Weber, S. Conchon, D. Déharbe, M. Heizmann, A. Niemetz, and G. Reger, “The SMT competition 2015–2018,” *SAT*, vol. 11, no. 1, pp. 221–259, Sep. 2019, doi: 10.3233/SAT190123.

[13] H. Barbosa, J. Hoenicke, and A. Hyvarinen, “16th international satisfiability modulo theories competition (SMT-COMP 2021): Rules and procedures,” p. 24, May 2021 [Online]. Available: https://smt-comp.github.io/2021/rules.pdf. [Accessed: 16-May-2022]

[14] D. Beyer, S. Löwe, and P. Wendler, “Reliable benchmarking: Requirements and solutions,” *Int J Softw Tools Technol Transfer*, vol. 21, no. 1, pp. 1–29, Feb. 2019, doi: 10.1007/s10009-017-0469-y.

[15] D. Merkel *et al.*, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[16] Open Container Initiative, “OCI runtime specification version 1.0.2,” 27-Mar-2020. [Online]. Available: https://github.com/opencontainers/runtime-spec/blob/v1.0.2/spec.md. [Accessed: 16-Mar-2022]

[17] Open Container Initiative, “OCI image format specification version 1.0.2,” 17-Nov-2021. [Online]. Available: https://github.com/opencontainers/image-spec/blob/v1.0.2/spec.md. [Accessed: 16-Mar-2022]

[18] Flexera, “2022 Flexera state of the cloud report.” [Online]. Available: https://info.flexera.com/CM-REPORT-State-of-the-Cloud. [Accessed: 17-Apr-2022]

[19] “Dockerfile reference,” 2021. [Online]. Available: https://github.com/docker/cli/blob/v20.10.13/docs/reference/builder.md. [Accessed: 16-Mar-2022]

[20] J. Cook, “The dockerfile,” in *Docker for data science: Building scalable and extensible data infrastructure around the jupyter notebook server*, 2017, pp. 81–101, doi: 10.1007/978-1-4842-3012-1_5.

[21] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, “A survey on automated dynamic malware-analysis techniques and tools,” in *ACM Computing Surveys*, 2008, vol. 44, doi: 10.1145/2089125.2089126.

[22] B. A. Wichmann, A. A. Canning, D. W. R. Marsh, D. L. Clutterbuck, L. A. Winsborrow, and N. J. Ward, “Industrial perspective on static analysis,” in *Software Engineering Journal*, 1995, vol. 10, pp. 69–75, doi: 10.1049/sej.1995.0010.

[23] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, “Chapter one - security testing: A survey,” in *Advances in computers*, 2016, vol. 101, pp. 1–51, doi: 10.1016/bs.adcom.2015.11.003.

[24] T. Ball, “The concept of dynamic analysis,” *SIGSOFT Softw. Eng. Notes*, vol. 24, no. 6, pp. 216–234, Oct. 1999, doi: 10.1145/318774.318944.

[25] M. C. Fanning and L. J. Golding, Eds., “Static analysis results interchange format (SARIF) version 2.1.0,” 17-Mar-2020. [Online]. Available: https://docs.oasis-open.org/sarif/sarif/v2.1.0/os/sarif-v2.1.0-os.html. [Accessed: 16-Mar-2022]

[26] P. Anderson, Ł. Kot, N. Gilmore, and D. Vitek, “SARIF-enabled tooling to encourage gradual technical debt reduction,” in *2019 IEEE/ACM international conference on technical debt (TechDebt)*, 2019, pp. 71–72, doi: 10.1109/TechDebt.2019.00024.

[27] MITRE, “CWE - common weakness enumeration.” [Online]. Available: https://cwe.mitre.org. [Accessed: 31-Mar-2022]

[28] MITRE, “CWE - CWE-699: Software development (4.6).” [Online]. Available: https://cwe.mitre.org/data/definitions/699.html. [Accessed: 31-Mar-2022]

[29] MITRE, “CWE - CWE-89: Improper neutralization of special elements used in an SQL command (’SQL injection’) (4.6).” [Online]. Available: https://cwe.mitre.org/data/definitions/89.html. [Accessed: 31-Mar-2022]

[30] MITRE, “CWE - CWE-compatible products and services.” [Online]. Available: https://cwe.mitre.org/compatible/product.html. [Accessed: 31-Mar-2022]

[31] D. E. Mann and S. M. Christey, “Towards a common enumeration of vulnerabilities,” presented at the 2nd workshop on research with security vulnerability databases, 1999.

[32] Docker Inc., “Docker SDK for python,” 07-Oct-2021. [Online]. Available: https://docker-py.readthedocs.io/en/stable/. [Accessed: 16-May-2022]

[33] YAML Language Development Team, “YAML ain’t markup language (YAML™) revision 1.2.2,” 01-Oct-2021. [Online]. Available: https://yaml.org/spec/1.2.2/. [Accessed: 14-May-2022]

[34] C. O’Connor, “StrictYAML,” 2021. [Online]. Available: https://hitchdev.com/strictyaml/. [Accessed: 31-Mar-2022]

[35] S. Colvin, “Pydantic,” 31-Mar-2022. [Online]. Available: https://github.com/samuelcolvin/pydantic. [Accessed: 31-Mar-2022]

[36] A. Wright, H. Andrews, B. Hutton, and G. Dennis, “JSON schema: A media type for describing JSON documents,” Internet Engineering Task Force, Internet Draft draft-bhutton-json-schema-00, Dec. 2020 [Online]. Available: https://datatracker.ietf.org/doc/draft-bhutton-json-schema-00. [Accessed: 14-May-2022]

[37] The SpotBugs Core Team, “SpotBugs.” [Online]. Available: https://spotbugs.github.io/. [Accessed: 16-May-2022]

[38] P. Arteau, “Find security bugs.” [Online]. Available: https://find-sec-bugs.github.io/. [Accessed: 16-May-2022]

[39] M. Mues, T. Schallau, and F. Howar, “Jaint: A framework for user-defined dynamic taint-analyses based on dynamic symbolic execution of java programs,” in *Integrated formal methods*, 2020, pp. 123–140, doi: 10.1007/978-3-030-63461-2_7.

[40] K. Luckow *et al.*, “JDart: A dynamic symbolic analysis framework,” in *Tools and algorithms for the construction and analysis of systems*, 2016, pp. 442–459, doi: 10.1007/978-3-662-49674-9_26.

[41] K. Havelund and T. Pressburger, “Model checking JAVA programs using JAVA PathFinder,” *STTT*, vol. 2, no. 4, pp. 366–381, Mar. 2000, doi: 10.1007/s100090050043.

A. Appendix

Listing A.1: JSON Schema of CSAT’s vulnerability metadata

```json
{
  "title": "Metadata",
  "description": "`pydantic.BaseModel` class with built-in YAML support.\n\nYou can alternatively inherit from this to implement your model:\n`(pydantic_yaml.YamlModelMixin, pydantic.BaseModel)`\n\nSee Also\n--------\npydantic-yaml: https://github.com/NowanIlfideme/pydantic-yaml\npydantic: https://pydantic-docs.helpmanual.io/\npyyaml: https://pyyaml.org/\nruamel.yaml: https://yaml.readthedocs.io/en/latest/index.html",
  "type": "object",
  "properties": {
    "name": {
      "title": "Name",
      "type": "string"
    },
    "vendor": {
      "title": "Vendor",
      "type": "string"
    },
    "version": {
      "title": "Version",
      "type": "string"
    },
    "url": {
      "title": "Url",
      "minLength": 1,
      "maxLength": 2083,
      "format": "uri",
      "type": "string"
    },
    "language": {
      "$ref": "#/definitions/SupportedProgrammingLanguagesEnum"
    },
    "build": {
      "$ref": "#/definitions/BuildMetadata"
    },
    "exec": {
      "$ref": "#/definitions/ExecMetadata"
    },
    "vulnerability": {
      "$ref": "#/definitions/VulnerabilityMetadata"
    },
    "provides_patch": {
      "title": "Provides Patch",
      "default": false,
      "type": "boolean"
    }
  },
  "required": [
    "name",
    "vendor",
    "version",
    "url",
    "language",
    "build"
  ],
  "additionalProperties": false,
  "definitions": {
    "SupportedProgrammingLanguagesEnum": {
      "title": "SupportedProgrammingLanguagesEnum",
      "description": "An enumeration.",
      "enum": [
        "java"
      ],
      "type": "string"
    },
    "SupportedBuildSystemsEnum": {
      "title": "SupportedBuildSystemsEnum",
      "description": "An enumeration.",
      "enum": [
        "javac",
        "maven"
      ],
      "type": "string"
    },
    "BuildMetadata": {
      "title": "BuildMetadata",
      "type": "object",
      "properties": {
        "build_system": {
          "$ref": "#/definitions/SupportedBuildSystemsEnum"
        }
      },
      "required": [
        "build_system"
      ],
      "additionalProperties": false
    },
    "ExecMetadata": {
      "title": "ExecMetadata",
        "type": "object",
        "properties": {
          "bin_path": {
            "title": "Bin Path",
            "type": "string"
          },
          "main_class": {
            "title": "Main Class",
            "type": "string"
          },
          "arguments": {
            "title": "Arguments",
            "type": "string"
          }
        },begin{d
        "additionalProperties": false
      },
      "VulnerabilityLocationMetadata": {
        "title": "VulnerabilityLocationMetadata",
        "type": "object",
        "properties": {
          "file": {
            "title": "File",
            "type": "string"
          },
          "line": {
            "title": "Line",
            "type": "string"
          },
          "class_name": {
            "title": "Class Name",
            "type": "string"
          },
          "function_name": {
            "title": "Function Name",
            "type": "string"
          }
        },
        "required": [
          "file"
        ],
        "additionalProperties": false
      },
      "VulnerabilityMetadata": {
        "title": "VulnerabilityMetadata",
        "type": "object",
        "properties": {
          "cve": {
            "title": "Cve",
            "type": "string"
          },
```

```json
          "cwe": {
            "title": "Cwe",
            "type": "string"
          },
          "location": {
            "$ref": "#/definitions/VulnerabilityLocationMetadata"
          }
        },
        "required": [
          "location"
        ],
        "additionalProperties": false
      }
    }
}
```