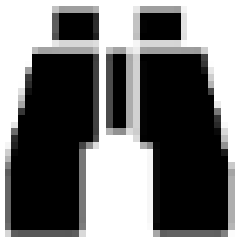
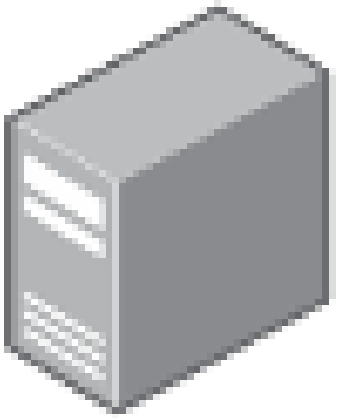
University of Applied Sciences and Arts Northwestern Switzerland
School of Computer Science

Tracing of Technical Debt in Heterogeneous Environments

P7

Windisch, January 2025





Student Esra Siegert

Fachbetreuer Sebastian Graf

Fachhochschule Nordwestschweiz, Hochschule für Technik

Abstract

Common IT-architectures seldom are operated on single platforms only: Spanning over archaic monoliths and cutting-edge cloud native systems, hunting down issues becomes difficult. The different natures of heterogeneous systems challenge an overall, comprehensive view on static technical debt such as security issues in dependencies. The architectures and the technologies are just too different. In this paper, we provide an in-depth analysis of this problem of limited visibility into software dependencies mapped to vulnerabilities in hybrid environments. By nominating a common denominator, the network between platforms in our concrete example implementation, we develop an approach to analyze technical debt across various platforms. Our example implementation bases on requests between components, which are enriched with any kind of specific platform-generated meta-information such as Software Bills-of-Material (SBOMs). We prove the feasibility of our approach with examples ranging from Monoliths on vintage platforms to Microservices, hosted on Cloud Native Environments. Our example solution, provided as open source framework based on eBPF, generates dynamically SBOMs, captures network events, and enables swift and comprehensive detection of statically analyzed technical debt.

Keywords:

Monoliths, Microservices, Software Supply Chain, Technical Debt, Cloud Native, CVE

Contents

List of Figures

1 Introduction

Today, common software products consist of millions of lines of code. To avoid rewriting solved problems, we commonly use existing components, which provide a certain, already implemented functionality. Technically, these components can range from legacy software running on legacy platforms to software dependencies utilized in the own source code. In all cases, product teams often do not reflect how these components are written and what weaknesses they potentially contain. To support these teams, widely used tools for static code analysis are available and applied within the build process, such as Sonarqube [1]. Since many components are not built on a regular basis, code smells that appear long after the build phase are not detected. The transparency in runtime of the supply chain of a product can not be assured, since tools differ in a heterogeneous environment. Nevertheless, especially when it comes to security, such technical debt might have an enormous impact. A famous example was the vulnerability of the supply chain [2]. Log4shell is a vulnerability that occurred in the widely used Log4j log framework. Through manipulated log statements, it is possible to execute dynamic remote code through JNDI on any server [3]. This CVE existed in nearly all modern Java applications: either as a software component accessed via defined APIs or via a direct dependency in the own source code. Technically, any foreign Java source code could be executed in your application using this CVE. It is impossible to gain awarness of such vulnerabilities at the time the dependency was included or the software was built. As a consequence, there is an inherent need to continuously analyze the running software on platforms against any code smells that come up after build time. This need becomes harder because products often span across several platforms with different stacks and technical maturities. It is difficult to have a holistic view of the own environment because common productive environments are often heterogeneous.

By analyzing real-life production systems, we realized that we need to use the only layer commonly used by all platforms and components within a heterogeneous software product: the network.

Data derived from continuously analyzed platforms are used to enrich traffic between different components. This enables us to detect technical debt across the borders of environments such as Kubernetes and Linux hosts in our proof-of-concept. Our example implementation provided as an open source tool is called CISIN (Cilium Span Injector) [4]. By mapping the findings to services, we are able to offer a variety of actions starting from an in-depth monitoring for product teams to automatic remediation of the service, if possible.

2 The challenge of identifying dependencies and technical debt in diverse IT environments

We addressed a pressing problem in product development: the lack of visibility into software and component dependencies. Using software vulnerabilities as a representative, we see that it is challenging to offer a monitoring and tracing that spans both modern and legacy environments. The main issues arise from the widespread use of third-party libraries and dependencies in software development. Today, between 200-300 transitive dependencies are easily part of a software product by only using standard frameworks. Product teams are simply often overwhelmed with comprehensive tracking, especially when it comes to legacy software. When technical debt is discovered in these transitive dependencies, such as high-profile incidents like Log4shell, organizations struggle to identify the parts of the systems which are affected.

The challenge is further compounded in complex, heterogeneous environments that combine modern cloud native applications with legacy host-based systems. Product teams need to understand what components of a software are running, their dependencies, and how services

interact with each other. Without this transparency, the resolution of any kind of technical debt in runtime is slow and reactive, leaving systems exposed for extended periods.

We solve the challenge of analyzing any kind of dependencies in heterogeneous environments, regardless of whether the software is containerized or host-based. By assessing the communication flows between services, we are able to provide actionable insights to prioritize any kind of findings regarding technical debt in run-time. Products teams are thereby enabled to gain insights of any kind of analysis continuously, regardless of whether their application is running on legacy only or also spanning to modern container platforms.

3 Enhancing observability and dependency management with eBPF, SBOMs, and a modular architecture

To address the complex challenges of modern and legacy system observability, dependency management, and technical debt detection, we present an integrated approach that utilizes eBPF for network monitoring, Software Bills of Materials (SBOMs) for dependency tracking, and a modular architecture for scalable and efficient vulnerability management. The last component regarding vulnerability management could be replaced with any kind of tool that also addresses other kinds of technical debt at runtime. Our solution enhances classical observability by enriching distributed tracing. This allows for seamless tracking of network events without requiring any code modifications. Our approach is based on eBPF, Cilium, and Hubble for real-time network monitoring and generates and manages SBOMs for both containerized and legacy systems. The modular design of the system ensures flexibility in monitoring, tracing, and managing dependencies across diverse environments, while providing actionable insights for technical debt remediation.

3.1 Utilizing eBPF to enrich distributed tracing

From an external perspective, our approach resembles the classical distributed tracing [5]. The key difference lies in how to obtain the trace data. In traditional distributed tracing, trace data is typically collected using auto-instrumentation libraries or by integrating dedicated tracing libraries directly into the application code. This works well for modern applications. Since our solution also aims at legacy systems, introducing tracing at the source code level is not feasible. Instead, the approach should work without requiring modifications from the developers.

Using eBPF [6], we can connect to legacy platforms at the operating system level. We retrieve network events via Cilium. Hubble, which provides efficient monitoring of network traffic at the kernel level, enables us to collect the necessary data. Capturing raw traffic details such as source and destination IPs, ports, and protocols, Hubble processes these into structured, metadata-rich events. On cloud platforms, these events include contextual details such as workloads, namespaces, and Kubernetes-specific identifiers.

To extend observability to legacy systems and external hosts, we incorporate them into Cilium's Cluster Mesh. This allows the network traffic of the legacy host to be seamlessly monitored alongside cloud native workloads.

To handle the high volume of network events generated, we apply filtering to exclude redundant or non-essential data, focusing on critical service-to-service communications only. The resulting events provide actionable insights into interactions and dependencies, forming the base for enriched traces linked to vulnerabilities and software dependencies.

To enable teams to interact directly with these network events, we convert Cilium data into distributed trace compatible traces. As illustrated in Figure 3.1, such traces consist of one

or more spans. Using the OpenTelemetry library [7], we generate traces enriched with SBOM metadata, associating them with spans derived from Cilium's network events. This approach offers the advantage of using a standardized format with robust tooling support, such as Jaeger [8], to visualize and analyze the traces.

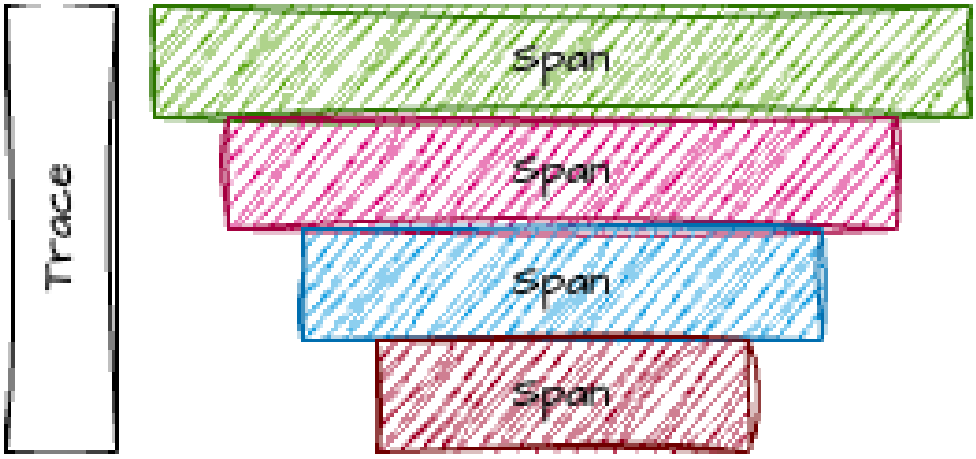


Figure 3.1: Trace

3.2 Generating and managing SBOMs for modern and legacy systems

To catalog dependencies, we used the concept of software bills of material (SBOM), which is commonly known. An SBOM contains a machine-readable list of dependencies stored in multiple formats such as CyclonDX [9] or SPDX [10]. The most accurate SBOMs are generated during the build time. Unfortunately, not all software or library providers provide them. Furthermore, awareness of managing SBOMs as a standardized meta-artifact just gained attention within the last few years. As a consequence, in some cases, an SBOM must be generated based on the software artifact itself.

In CISIN we differentiate between software artifacts on a legacy system and images used in a cloud native environments such as Kubernetes cluster. Images in a Kubernetes cluster are easier to handle. We can check the image registry if an SBOM has been provided for the image. If this is the case, we can simply pull the SBOM from the image registry. If there is no SBOM available, we have to generate it on our own. There exist various tools to scan the image and generate the SBOM. In our proof of concept, we chose the scanner tool Syft [11] for CISIN. Another option would have been Trivy [12].

SBOMs for the legacy system are more complicated. For the legacy systems, we have neither an image registry, where we can check for existing SBOMs, nor do we have software artifacts as OCI images, which are self-containing regarding their own dependencies. However, we can make use of the same tools as we did on modern platforms. We are able to scan parts of the file system and generate a SBOM for these parts. The disadvantage of this approach is the possible scanning overhead of a large filesystem. To mitigate this disadvantage, we are filtering parts of the root directory on legacy systems to accelerate the generation of the SBOM. Once an SBOM is generated for a legacy system, the SBOM is stored along with the other SBOMs of the cloud native systems.

3.3 Modular architecture for dependency and vulnerability management

Our architecture, visualized in Figure 3.2, is built around two primary components: the agent and the server, each playing a vital role in the capture, processing, and analysis of network events

to provide actionable insights into software dependencies and vulnerabilities. The supporting components are auxiliary systems for communication, storage, and visualization, creating a cohesive and scalable solution.

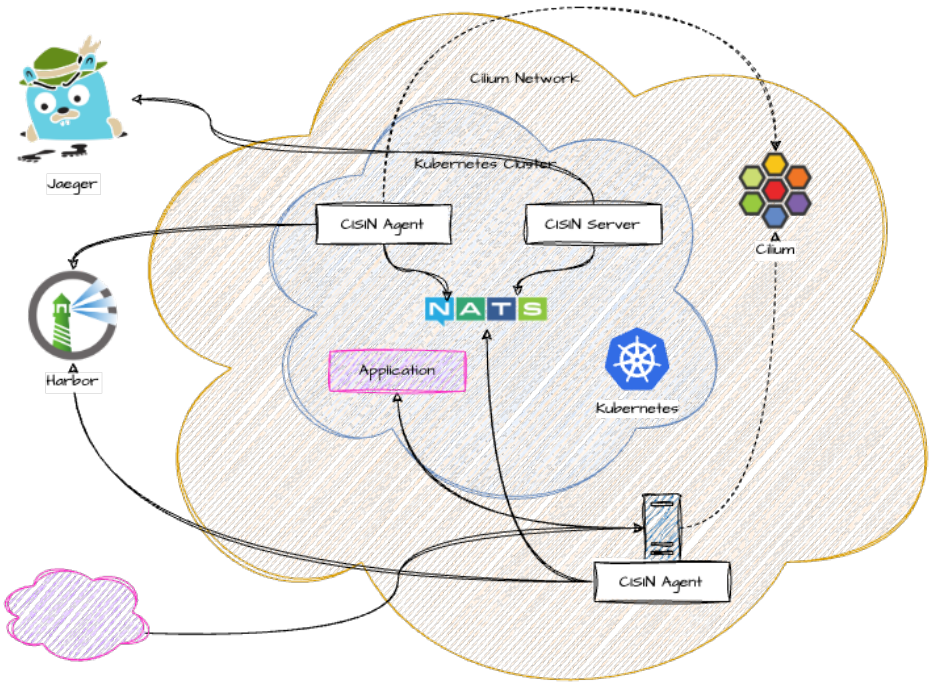


Figure 3.2: Architectural overview

The agent runs on each monitored system to collect network flow data and manages dependency analysis. In Kubernetes environments, we deploy the agent as a DaemonSet, ensuring that every node runs an instance. For legacy systems and external hosts, the agent is installed as a standalone process, seamlessly integrated into the Cilium Cluster Mesh. This integration allows us to monitor network flows across all systems, whether containerized or traditional.

Using Hubble's gRPC API, the agent collects real-time network events, enriched with metadata such as source and destination workloads, namespaces, and container details. For legacy systems, the agent retrieves the data from the network at the socket level to ensure comprehensive coverage. To further enhance the data, the agent generates SBOMs if necessary. For containerized workloads, we query the container runtime to identify running images and check for SBOMs in our OCI-compliant registry. For legacy systems, we perform filesystem scans to generate SBOMs, which are then pushed to the registry for central management.

To optimize performance, we designed the agent to filter and cache network events. Redundant or low-priority flows are excluded, ensuring only relevant data are forwarded. This filtered data is sent to the NATS [13] message queue, where it becomes accessible to the server for further processing.

The server acts as the central processor, aggregating data from multiple agents. By subscribing to the NATS message queue, the server collects network events and SBOM information, constructing detailed traces of service interactions. Each trace is built hierarchically, with spans representing individual service-to-service communications. Using the SBOM data provided by the agent, we enrich these spans with metadata linking to the software dependencies involved.

This trace enrichment ensures a clear mapping between network activity and potential vulnerabilities.

To manage scalability and persistence, we designed the server to process data in memory for speed while providing options for integration with external databases like Redis for longer-term storage. Once the traces are enriched, we format them using OpenTelemetry standards and export them to Jaeger for visualization. Jaeger allows us to display the traces as interactive hierarchies, linking service interactions with their associated vulnerabilities, helping administrators quickly prioritize remediation efforts.

Supporting components like NATS facilitate efficient and asynchronous communication between agents and the server. Additionally, we use an OCI-compliant registry to store SBOMs, ensuring centralized access for trace enrichment and analysis.

This architecture enables us to dynamically monitor network activity, map service dependencies, and identify vulnerabilities in diverse environments. Using the modularity of our agent and server components, we ensure scalability, extensibility, and efficient operation in both modern and legacy systems.

4 Related work

The paper highlights that integrating tracing and SBOMs is a relatively unexplored area. Although significant research exists in fields such as supply chain security, SBOMs, tracing, and eBPF, combining these approaches are rare. One notable exception is the work [14], which proposes a framework to generate SBOMs dynamically during the deployment of Docker containers. Their research also conducts security analyzes based on these SBOMs. However, their approach focuses exclusively on containerized environments, whereas our work includes legacy systems. Moreover, the integration of tracing mechanisms to establish dependencies among containers and virtual machines is a central element of our contribution, differentiating it from prior efforts.

Regarding eBPF, [15] presented an approach that uses eBPF and Linux IMA (Integrity Measurement Architecture) to enforce SBOM constraints, preventing unauthorized file modifications. While their approach shares certain methodologies, it targets different goals and applies eBPF in a distinct context.

5 Conclusion and further work

In this work, we showed that there is an imminent need for product teams to gain insights into static technical debt. Today, heterogeneous different platforms that host a software product hamper direct, platform-centric monitoring approaches.

With Cisin, we developed a tool that enables visibility into software dependencies and technical debt, such as vulnerabilities across diverse IT environments, integrating containerized and legacy systems. Using Cilium, eBPF, and Hubble, we can dynamically generate SBOMs, enrich network traces, and provide actionable insights into service interactions and associated risks. This approach bridges the gap between modern cloud-native infrastructures and traditional host-based setups, offering a unified framework for dependency tracking and technical debt management at run-time.

Looking ahead, we see significant opportunities to expand our work. One priority is introducing automated alerting based on vulnerability severity, enabling faster responses. Another key direction is the development of automated mitigation strategies, such as dynamic isolating vulnerable

services through network policies. These enhancements would further strengthen security while balancing service availability.

Bibliography

[1] G. A. Campbell and P. P. Papapetrou, *SonarQube in Action*. Manning Publications Co., 2013.

[2] *CVE-2021-44228*. Available from NVF, CVE-ID CVE-2021-44228. Oct. 3, 2021. [Online]. Available: `https://nvd.nist.gov/vuln/detail/CVE-2021-44228` (visited on Jan. 23, 205).

[3] „NVD - CVE-2021-44228“. (), [Online]. Available: `https://nvd.nist.gov/vuln/detail/CVE-2021-44228` (visited on Apr. 28, 2024).

[4] „Cisin: Cilium span injector“. (), [Online]. Available: `https://github.com/fhnw-imvs/fhnw-cisin` (visited on Jan. 25, 2025).

[5] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, 2020.

[6] H. Sharaf, I. Ahmad, and T. Dimitriou, „Extended berkeley packet filter: An application perspective“, *IEEE access : practical innovations, open solutions*, vol. 10, pp. 126 370–126 393, 2022.

[7] „OpenTelemetry“. (), [Online]. Available: `https://opentelemetry.io/` (visited on Apr. 28, 2024).

[8] „Jaeger: Open source, distributed tracing platform“. (), [Online]. Available: `https://www.jaegertracing.io/` (visited on Jul. 7, 2024).

[9] „OWASP CycloneDX Software Bill of Materials (SBOM) Standard“. (Jul. 1, 2024), [Online]. Available: `https://cyclonedx.org/` (visited on Jul. 6, 2024).

[10] „SPDX – Linux Foundation Projects Site“. (), [Online]. Available: `https://spdx.dev/` (visited on Jul. 6, 2024).

[11] *Anchore/syft*, Anchore, Inc., Apr. 20, 2024. [Online]. Available: `https://github.com/anchore/syft` (visited on Apr. 20, 2024).

[12] „Aquasecurity/trivy: Find vulnerabilities, misconfigurations, secrets, SBOM in containers, Kubernetes, code repositories, clouds and more“. (), [Online]. Available: `https://github.com/aquasecurity/trivy` (visited on Apr. 20, 2024).

[13] „NATS.io“, NATS.io. (), [Online]. Available: `https://nats.io/` (visited on Jul. 14, 2024).

[14] N. Kawaguchi and C. Hart, „On the Deployment Control and Runtime Monitoring of Containers Based on Consumer Side SBOMs“, in *2024 IEEE 21st Consumer Communications & Networking Conference (CCNC)*, 2024, pp. 1022–1025. DOI: `10.1109/CCNC51664.2024.10454654`.

[15] A. Crawford, E. Yakubovich, and R. Szumski, „Enforcing SBOMs through the Linux kernel with eBPF and IMA“, in *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, Copenhagen Denmark: ACM, Nov. 30, 2023, pp. 77–78, ISBN: 9798400702631. DOI: `10.1145/3605770.3625206`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3605770.3625206` (visited on Apr. 21, 2024).