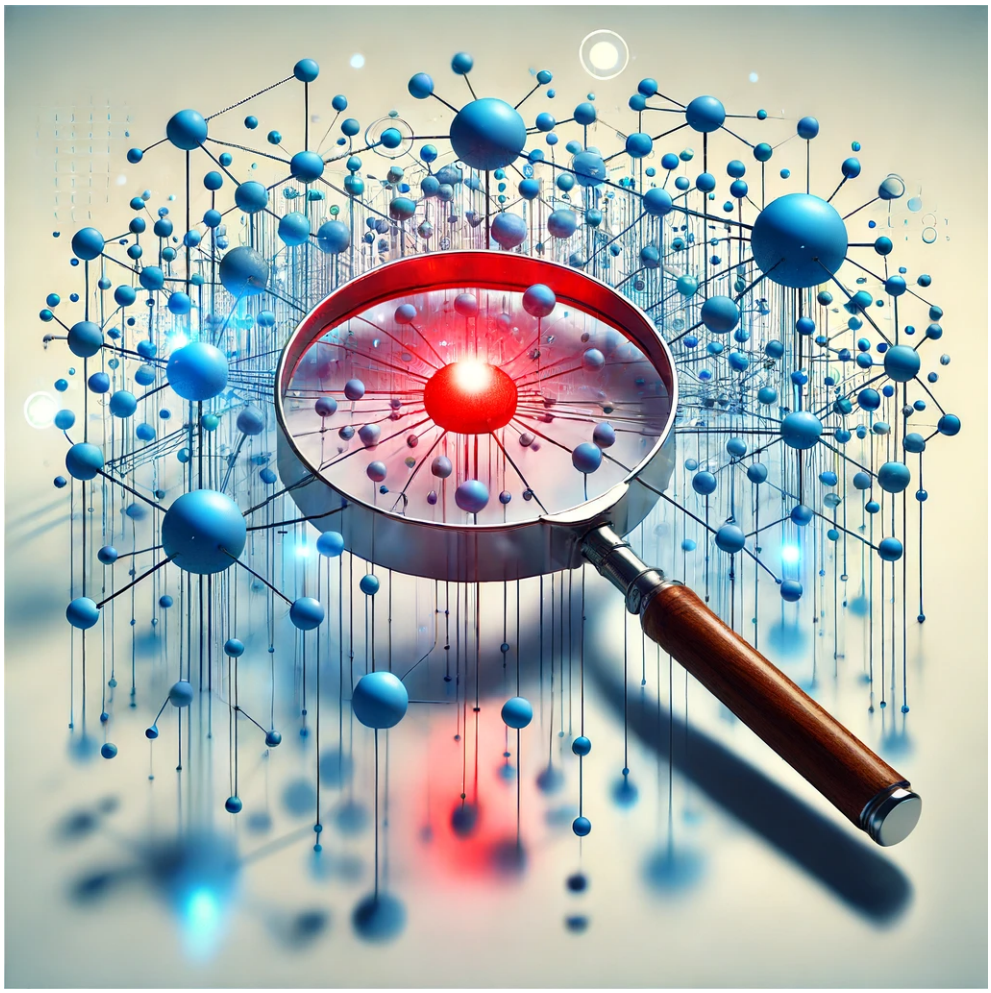
Leveraging Fine-grained Telemetry Data for the Detection and Prevention of Vulnerability Exploits

Version of June 18, 2024



Konrad Ponichtera

Leveraging Fine-grained Telemetry Data for the Detection and Prevention of Vulnerability Exploits

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Konrad Ponichtera
born in Warsaw, Poland



Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2024 Konrad Ponichtera.

Cover picture: An image generated using DALL·E 3 using a prompt, derived from the concepts of this research project.

The content of this thesis has been enhanced by using Grammarly for proofreading.
ChatGPT has been used for vision evaluation, as described in the methodology section.

Leveraging Fine-grained Telemetry Data for the Detection and Prevention of Vulnerability Exploits

Author: Konrad Ponichtera
Student ID: 5580749

Abstract

In the modern digital landscape, cybersecurity threats are a significant concern, particularly for publicly accessible computer systems. Vulnerabilities, or flaws in system design, can be exploited by malicious actors to compromise system security and integrity. This paper explores the challenges of handling vulnerabilities in software dependencies from the perspective of system operators, responsible for managing and monitoring infrastructure. The investigation is structured into two main steps. First, a detailed exploration of DevOps methodologies and current vulnerability handling techniques reveals key limitations and areas for improvement. Based on these insights, a new solution is proposed to enhance system operators' capabilities in terms of awareness, impact assessment, and actionability. To validate the proposed solution, semi-structured interviews were conducted with ten experienced software engineers. The analysis of these interviews, using grounded theory methods, helped to refine the proposed system's concepts and assess its potential impact on industry practices. In the second step, a fully functional prototype was developed, featuring the ability to monitor and mitigate vulnerability triggers in Java services through instrumentation. The evaluation of the prototype shows that the proposed solution effectively increases the granularity of vulnerability handling and is feasible for practical implementation from the resource utilization perspective.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. S. Proksch, Faculty EEMCS, TU Delft
Committee Member:	Dr. J. Decouchant, Faculty EEMCS, TU Delft

Preface

I want to express my gratitude to everybody who have supported me during my journey in completing this thesis.

First and foremost, I am profoundly grateful to my supervisor, Sebastian Proksch, for their guidance, insightful feedback, and continuous encouragement. Your expertise, dedication, and understanding of my engineer's nature have been invaluable in shaping this research, which showcases our mutual interest in the DevOps field of software engineering.

I also wish to thank my friends, who have been a constant source of support and motivation. The countless hours we spent studying in TU Delft's cybersecurity offices and hanging out together are memories that I am going to cherish forever. You gave me three years of the best student life I could dream of.

To my family, for your unconditional love and belief in me that have been my driving force. Thank you for always being there for me, providing me with the strength and perseverance needed to accomplish my goals.

Lastly, to my partner Heinrich, for your endless support, understanding, and encouragement that have been my anchor throughout this journey. Thank you for standing by my side and for your unwavering belief in my abilities, even in the most difficult moments.

This thesis would not have been possible without the contributions and support of all of you.

Konrad Ponichtera
Delft, the Netherlands
June 18, 2024

Contents

Preface iii

Contents v

1 Introduction 1

2 Challenges of handling vulnerabilities in operated systems 5
2.1 What is the DevOps pipeline? . . . 5
2.2 How is software deployed and monitored? . . . 7
2.3 How are vulnerabilities addressed? . . . 8
2.4 Limitations and improvement opportunities . . . 9

3 Improving vulnerability handling 13

4 Vision evaluation 19
4.1 Methodology . . . 19
4.2 Exploratory investigation . . . 23
4.3 Results and reflections . . . 47

5 Implementation 49
5.1 Structural primitives . . . 49
5.2 Prototype description . . . 53

6 Prototype evaluation 69
6.1 Methodology . . . 69
6.2 Qualitative evaluation . . . 71
6.3 Quantitative evaluation . . . 84

7 Discussion 99

8 Related work 105

9 Summary **109**

Bibliography **111**

Chapter 1

Introduction

In today's digital era, computers and related technologies play a central role in the lives of billions around the globe, influencing everything from daily communication to critical infrastructure. As our reliance on these systems grows, so does the landscape of cybersecurity threats. One of the primary concerns is the susceptibility of publicly accessible systems to malicious attacks. Computer systems accessible over the Internet are more vulnerable to attacks, compared to those hosted on closed networks, which are protected from the direct reach of external threats. Hackers often exploit these systems by identifying and leveraging errors in their building blocks or their setup. We classify these errors as *vulnerabilities*. Vulnerability is a flaw in the system design that, if exploited, could affect the system's behavior in a way that its architects and engineers did not intend. Usually, it means compromising the security and integrity of the system. Vulnerabilities can allow malicious actors to perform undesirable actions in a computer system, like obtaining unauthorized access to sensitive data or disrupting service operations. They can originate from various sources like errors in the code, improper configuration, or inadequate security practices.

Software engineers often introduce vulnerabilities into their applications by using external libraries as dependencies. These libraries can contain security flaws that may not be known to developers when integrating them into developed software. As a result, even if the original software code is secure, using vulnerable external libraries can compromise the entire application. This issue is compounded by many projects failing to keep these dependencies updated, and missing crucial patches that fix security issues. The most prominent example of such a problem in recent years was the discovery and disclosure of Log4Shell[1] zero-day vulnerability. Although it has been fixed in the newer versions of the affected library, its ubiquitousness and ease of exploit are expected to cause problems in the industry for many years to come [10]. Addressing vulnerabilities is therefore a vital part of software maintenance. Not doing so does not only jeopardize individual users but can also pose severe risks to organizations and national security. A clear understanding of their behavior and impact on the developed software is therefore essential for securing the systems and protecting against potential cyber threats.

Our investigation into current vulnerability handling practices has revealed that vulner-

[1] https://en.wikipedia.org/wiki/Log4Shell

abilities in software dependencies are most commonly discovered at the level of individual dependencies. This means that to determine if an application is vulnerable is to check whether any of its dependencies contain known vulnerabilities. This approach, however, is rather coarse-grained. It does not necessarily indicate that the developed application is calling the vulnerable code within these dependencies. Therefore, even if a dependency is flagged as vulnerable, there is no guarantee that the vulnerability is exploitable in the context of the developed application. This points to the need for more detailed ways to assess the impact of vulnerabilities in software.

In this paper, we present our vision of how handling vulnerabilities could be enhanced from the perspective of a system operator – the person responsible for managing infrastructure, deploying, and monitoring the applications, but without the ability to modify source code and artifacts of these applications. To tackle the aforementioned challenges, we need to find answers to the following research questions:

1. What are the challenges that system operators encounter with vulnerabilities in the system?
2. How to increase the involvement of the system operator in the vulnerability mitigation process?
3. How to mitigate the vulnerabilities within the system?
4. How to enhance the existing telemetry measures with information about triggered vulnerabilities?
5. How can we assess the overall impact of vulnerability on the entire system?
6. What is the performance impact of vulnerability trigger counting and mitigation?

Our research was structured into two distinct steps. In the initial step, we explored current DevOps methodologies and vulnerability handling techniques. This investigation revealed various limitations and opportunities for improvement, leading us to propose a new solution designed to enhance system operator capabilities in three critical areas – awareness, impact assessment, and actionability. Our envisioned solution offers a way to track vulnerabilities in operated software, provide a way to monitor their impact on all the system components within the service mesh, and monitor vulnerability triggers on the level of individual callables.[2] It also allows for granular vulnerability mitigation, by replacing vulnerable logic within dependencies through generically-defined mitigation strategies. To evaluate the relevance and usability of our idea, we carried out semi-structured interviews with ten software engineers possessing experience in system operation and administration. We applied grounded theory methods and coding techniques to analyze participants' answers and establish concepts, which helped us to get a bigger picture of how our idea could affect the practices within the industry.

In the second step, we developed a fully functional prototype of the envisioned system. Our prototype features the capability to monitor vulnerability triggers in Java services through instrumentation. It embeds information about the triggers into existing telemetry signals, which we then use to count the triggers in components of the operated system. By

[2] Abstract term for methods, functions, and other language constructs that can be invoked

collecting and processing these signals we provide a visualization in the form of *impact graphs*, allowing to determine parts of the service mesh, impacted by the vulnerability. The prototype also uses instrumentation as a way to mitigate vulnerable logic within dependencies. To identify vulnerable callables within application dependencies, we built upon the insights provided by the FASTEN research project [1]. We evaluated our implementation to assess its effectiveness and practicality. Qualitatively, we focused on determining whether the prototype indeed improved the awareness, impact assessment, and actionability of system operators as intended. Quantitatively, we examined the feasibility of integrating this system into existing workflows, particularly in terms of the additional overhead it introduced.

The results of our interviews with industry professionals reveal a clear demand and potential for increasing the granularity of vulnerability detection, and that our idea addresses this need effectively. Furthermore, the evaluation of our prototype demonstrates that the envisioned solution can be feasibly implemented in practical settings. Although the implementation targets Java applications, our vision applies to arbitrary language with well-defined instrumentation capabilities and available sources of vulnerable callables.

Overall, this thesis provides the following main contributions:

- Exploratory investigation and evaluation of the industry's needs in terms of increasing granularity of vulnerability detection
- Proposal for embedding callable-level trigger information into state-of-the-art telemetry signals
- Prototype of a system, that empowers system operators with the capability to explore vulnerabilities, and their impact on the operated system, as well as allows operators to apply mitigation strategies on the callable level

We have published two artifacts as appendices: the pseudonymized interview transcripts [33] and questionnaire results, along with the source code of the developed prototype and experimental setup [34].

Chapter 2

Challenges of handling vulnerabilities in operated systems

In this chapter, we will discuss the state of modern software development methodologies, covering all the stages of software creation from writing code, all the way to deployment and monitoring of the live systems. Then, we will identify these methodologies' limitations from the vulnerability mitigation perspective. Finally, we will focus on what can be done to overcome these limitations.

2.1 What is the DevOps pipeline?

Historically, software development and IT operations processes were disjoint tasks and procedures. The former focused on shaping and building the final product, done mainly by software developers and testers. At the same time, the latter was responsible for configuring the necessary infrastructure, deploying the product, and ensuring its proper operation through monitoring. The development and operation departments within enterprises were often functionally and organizationally split, which led to problems with communication and competing objectives, even though the ultimate goal for both of them was the same: to create a working and maintainable software system.

The idea of combining software development and operations methodologies appeared first in the early 80s [16]. As the rapid growth of the IT industry led to bigger teams being formed, the sociological problems became more apparent. The amount of abstraction in the form of new frameworks and technologies available in the market has further deepened the gap between groups, responsible for developing and deploying the system. Around two decades later, the voices calling for the merging of these methodologies became more vocal [13], arguing that the status quo is causing dysfunction within the software industry.

Therefore, the idea of DevOps methodology has been conceived. It aims to promote a collaborative approach to the tasks, performed by both development and operations departments within the same organization [27]. This has led to the popularization of software development practices among administrative tasks, for instance through their automation (continuous integration and deployment pipelines), the declarative way of configuring the

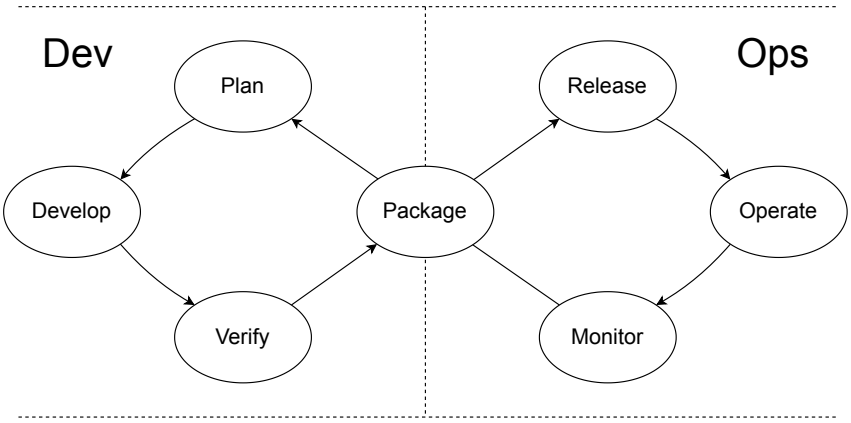


Figure 2.1: DevOps process lifecycle, depicted as an "infinity loop"

infrastructure (infrastructure as a code), and versioned configuration management (GitOps). Furthermore, the introduction of technologies like containerization allowed developers to take part in tasks, which were previously the responsibility of IT operations departments, creating a sense of cross-team ownership of the product. DevOps also enables developers to be more responsive to the needs of the live systems, by providing them with access to the runtime telemetry of the deployed applications. This makes necessary adjustments to features or fixing bugs more efficient since more data is available for root-cause analysis purposes.

DevOps is often seen as a high-level philosophy, that promotes blurring the border between the world of developers and operators and led to the creation of new software engineering practices like the twelve-factor app [44] methodology. However, in a more concrete interpretation, it can be seen as a list of steps, traditionally partitioned into separate categories of development and operation. First, the scope of changes and new features is discussed and developers are assigned tasks to modify the product's codebase. Once the changes are pushed to the code repositories, the verification phase is responsible for validating the code, for instance through static analysis and test suites. The validated code base is then packaged into a deployable artifact (like a JAR archive or Docker container) and released to one of the environments. By operating the application on the target infrastructure it is not only possible to identify problems that did not appear during the development phase (for instance regarding the scalability and load balancing between the instances), but also generate the runtime telemetry data. Through monitoring deployed applications, it is possible to identify the shortcomings, bottlenecks, and other flaws with the system's current state. This can be valuable information for the planning phase, where the insights from monitoring the live system can be used to make plans about the further development of the product. This infinite feedback loop is often strengthened by automating the majority of tasks into continuous integration, delivery, and deployment pipelines (CI/CD), which are responsible for respectively validating, packaging, and deploying applications to the infrastructure.

2.2 How is software deployed and monitored?

Splitting the workloads into separate microservices has been the leading design pattern in the software engineering industry for over a decade. It brought a lot of flexibility in terms of creating a coherent system from the separate building blocks, which could be developed in different technological stacks that are most suitable for the tasks that each of the microservices aims to solve. However, the microservice-based approach introduces challenges, which are not present in the older, monolithic architectures. For instance, using multiple technological stacks requires system operators to handle numerous ways of deploying the workloads. Since microservices rely on mutual communication, operators have to make sure that the traffic between the services works, is uniformly load-balanced and secure. Furthermore, the heterogeneous service environment creates observability challenges, like tracking the request propagation between the services. Whereas, in the monolith application, it would be merely a matter of browsing the logs, ordered by time.

Deployment stack The first problem was largely solved by introduction of the code-driven containerization, conceived by Docker in the past decade and standardized under the Open Container Initiative (OCI). Containers allow for running each service in a separate environment using the common runtime, regardless of the underlying technology of the service. Deploying the containers at scale on multiple machines is done by orchestrators like Docker Swarm or Kubernetes, which treat a cluster of multiple machines as a single logical unit, and provide networking abstraction, which allows the workloads to communicate, regardless of which machine they are deployed on.

To provide advanced network traffic management capabilities, container clusters are often enhanced by utilizing service meshes. A service mesh is an infrastructure layer, which intercepts ingress and egress traffic between the components of the microservice system. It is implemented by deploying a lightweight sidecar proxy next to each service instance. Handling of the traffic is done transparently from the application's point of view – no modifications of the executable code are necessary. Service meshes bring multiple benefits for operators and developers alike, for instance by transparently encrypting the traffic between services, and enabling advanced network traffic routing strategies, useful in canary deployments or A/B testing.

What's more, since sidecar proxies collect information about all the traffic coming in and out of the services, the mesh can track and visualize connections between the services. The proxies can also report metadata of the exchanged traffic, like the REST API endpoint paths, response statuses, payload sizes, and so on. For example, given an inbound request received by the service instance and a downstream call made by that service afterward, the mesh itself has no way of determining if the downstream call was caused by the inbound request. To do that, the service needs to propagate additional metadata.[1] This brings us to the concept of distributed tracing, which we will discuss next.

[1] https://istio.io/latest/about/faq/distributed-tracing/#istio-copy-headers

Observability stack The modern state of the art observability for microservice revolves around three components: logging, metrics, and tracing. Logging allows developers and operators to look into the output of the application, correlated with what is being done by the application threads at the moment. Metrics are used to report the state of the applications in a particular instant, like its memory consumption and distribution, CPU usage, and so on. Both logging and metrics-based observability can be used with monolithic services. Distributed tracing was developed specifically to tackle the problem of tracking the flow of downstream calls within the microservice system. It is used to track the request propagation within the microservice environment, allowing for determining which services were triggered by a single call from outside of the system. Tracing works similarly to the call stack within a single-process monolith, which allows for tracking executed methods.

How does distributed tracing work? Trace is a record of a unique transaction, handled by the system's entrypoint, triggering a certain amount of downstream services as the transaction is being executed. In the case of the system, exposing REST API, the transaction would be a single call to the API gateway. Each trace consists of spans, which are single operations within a trace. These can be a downstream service call, individual method execution, database query, and so on. Tracking of a request within a microservice system is done by propagating a unique tracing identifier, assigned to the request on the entrypoint – usually by the service mesh sidecar. In the case of REST services, an identifier is injected into the request's headers.[2] Each service within the system must be programmed to propagate this header value to all the downstream services. Since service mesh sidecar proxies intercept all the ingress and egress traffic of the service instances, they report encountered requests as spans to the tracing system like Jaeger[3]. The tracing system is then responsible for aggregating all the spans with the same tracing identifier into one trace. Applications are also able to report their spans on their own, for example by creating spans for individual method executions, as well as the traffic that cannot be read by the sidecar proxy (like when making database queries through a shared connection pool).

2.3 How are vulnerabilities addressed?

Vulnerabilities are design flaws and implementation errors, which pose a risk of being exploited by malicious actors to get unauthorized access to the system, disrupt its operations, or steal data. They vary in severity, which is usually bound to the scope of damage that could be caused in case they were to be exploited. For example, low-severity vulnerabilities can be seen as requiring attackers to perform social engineering and provide only limited unwanted access, while critical ones can be exploited remotely, without any extra steps, and lead to a privilege escalation that grants the attacker administrative access to the underlying system. Vulnerabilities can exist on any layer of the infrastructure stack, both in hardware and software. An example of hardware vulnerability would be susceptibility to side-channel attacks, stemming from characteristics of the hardware. A typical low-level

[2] https://github.com/openzipkin/b3-propagation
[3] https://www.jaegertracing.io/

software vulnerability is a buffer overflow, which occurs when too much data is written into a space-constrained place in the memory, overwriting unwanted memory areas and causing corruption or even arbitrary code execution. However, security flaws can also exist on higher abstraction levels of the software. In the case of SQL injection attackers can insert malicious database queries into a request input field. Meanwhile, path traversal bugs allow the attacker to freely browse the system directories outside of the application's working directory.

Software vulnerabilities can be introduced into the project's code base during development, as a result of a human error or deliberate supply-chain attack. Various static analysis tools like Snyk[4] can help spot them early, either in the developer's IDE or during CI/CD pipeline execution. However, modern application development often relies on using external dependencies like frameworks and libraries, which can have their own security flaws. Therefore, each introduction of a new dependency carries a risk of including new vulnerable code in the final product.

The ultimate goal of the vulnerability mitigation procedure is to eliminate the problematic parts from the developed system's code base. The current state-of-the-art mitigation techniques rely on continuous scanning of the used dependencies, and raising an alert whenever information about a new vulnerability is published [40]. If there is a patched version of the dependency, it can be used to replace the vulnerable one. It might happen that some functionality was modified or removed, especially if the previous version of the dependency was not updated for a long time. In such cases, incompatibilities will have to be addressed and solved by the developer. If a patched dependency version is not available, or patching would require too big of a refactor, the developer must assess if the vulnerable piece of code is being executed, and implement mitigation manually. This could be achieved by replacing the library with another one, or temporarily commenting out the code and returning an error instead.

2.4 Limitations and improvement opportunities

During our investigation of commonly used vulnerability mitigation techniques, we've identified several challenges and limitations. First of all, the current method of determining the vulnerability presence is very coarse-grained and relies only on scanning used dependencies. There is no way of determining which vulnerabilities are actually triggered during the normal operation of the system. The process of vulnerability mitigation is also time consuming, while the fast methods of disabling affected code require taking down entire microservices, potentially impacting parts of the system that are not affected. What's more, during the process of mitigation there is an imbalance in the distribution of responsibilities. Most of the work is done by the developers, and the operator's involvement is minimal or even non-existent, which directly clashes with DevOps principles of common ownership of the product. Finally, there seems to be no way to universally determine the impact of mitigation on the rest of the system. We will now elaborate on them and provide an idea of how they could be tackled.

[4] https://snyk.io/

No observability measure to count actual vulnerability triggering More often than not, the data provided in the vulnerability catalogs can be used only to determine the insecure version of the dependency. There is not enough information to determine which part of the program is affected on the callable level. Meanwhile, there is a chance that the reported vulnerability is not even being triggered by the application. Wang et al. [43] discovered during their empirical study that over half of the used libraries have at most 2% of their exposed APIs called across other projects. Therefore, treating the whole service as potentially vulnerable because one of its libraries is vulnerable has a high chance of effectively being a false positive, since the problematic code might end up not being called at all. As for the time of writing, no solution seems to exist that would allow for determining vulnerabilities, triggered by the live system, down to the callable level. From the perspective of all the teams, responsible for the system, it would be beneficial to have a way to determine which vulnerabilities are triggered during the system's operation, how often they are triggered, in which services they occur, and in which dependencies of those services. Having this information would allow the team to assess the risks, monitor the system for exploit attempts, and prepare a long-term mitigation plan. Therefore, having information about which callables within the code are vulnerable would allow for extending the current observability stack with a measure to collect executions of faulty callables.

Long mitigation process The usual development process of a system used in production requires the software developer to implement the change and push it through automated testing and code quality pipelines. Once the pipeline finishes successfully, manual testers have to perform acceptance tests. Depending on the scope of the change, it might be necessary to perform full regression testing, which requires checking every scenario to make sure that nothing gets broken. During each of these steps, there is a risk of having to start the whole process from scratch – either when the automated analysis returns errors, or the manual testers report inconsistencies to the developers. This means that mitigating the vulnerability is highly dependent on the project's development methodology, and could take days or even weeks before mitigation makes it to the production environment. In the case of high-risk systems, it could be beneficial to apply a temporary mitigation measure, which prevents vulnerable code from being executed. Although it does not solve the problem of vulnerability being present in the artifact of the service, it would vastly shorten the initial reaction time. It would also give the team of developers extra time and peace of mind, knowing that the faulty code is not being called on a deployed environment. Applying mitigation in an automated and scalable way could be achieved, for instance, by instrumenting the services within the system, and replacing the execution of vulnerable callables with custom logic.

No fine-grained mitigation possibility In case of certain critical vulnerabilities, it would be beneficial to mitigate them as soon as possible – even if the mitigation would be as simple as throwing an error instead of executing faulty code. This would give developers time to implement legitimate mitigation, at the expense of certain parts of the system being non-operational, but with the assurance that the vulnerable code is not being triggered. The simplest mitigation would be to take down vulnerable services. This task can be assigned to

the operator, who has access to the administrative panels of the deployment clusters. However, depending on the scope of responsibilities of the vulnerable services, this operation might be too coarse-grained and result in the unavailability of the non-vulnerable functionalities of the system. Assuming that there exists a source of information on which callables are vulnerable, the aforementioned use of instrumentation could also tackle the problem of mitigation granularity, and allow for disabling only these parts of code, which are directly responsible for the vulnerability being present.

Lacking involvement of system operator Once the mitigation is implemented by developers, the operator can be assigned the task of deploying a patched version of the service. This means that unless they are asked to take down the vulnerable service, from the operator's perspective the mitigation procedure doesn't differ in any way from a normal deployment of the new version. What's more, if there is a sufficient level of automation and all the deployment is done by the CI/CD pipelines, the operator might not be involved at all. While this might be seen as a benefit, it doesn't fulfill the DevOps principles, where all the teams responsible for delivering the product are involved to a certain degree in what's happening with it. Therefore, it would be beneficial to introduce a way for the system operator to be informed about the vulnerabilities – both present and executed within the system. This could be achieved with a dashboard, which enables the operator to browse the triggered vulnerabilities and be able to apply mitigations, without involving a team of developers.

No way to assess mitigation impact The process of deciding to apply a certain mitigation strategy would require assessing two factors. The first one would be the risk, posed by the vulnerability being triggered in the system – the higher the risk, the more likely the mitigation would be to be applied. The other one would be the impact of the mitigation on the rest of the system – the higher the impact, the less likely it would be for the team to apply it. Introducing a mitigation strategy in one service can have a cascading effect on the other ones, which rely on making downstream calls to it. Both of these factors are highly dependent on the nature of the vulnerability and the architecture of the microservice system. Therefore, it would be good to have a universal measure to determine the impact of the mitigation on the whole system and help make the decision. The highly observable nature of the service meshes and components of the modern observability stack could be used to help with that.

Chapter 3

Improving vulnerability handling

To tackle the aforementioned challenges, we propose a solution, that will allow for monitoring vulnerabilities throughout the microservice systems in service meshes, provide the system operator with a way to apply short-term mitigation strategies to them, and allow for assessing the impact of these mitigations on the entire system.

Vulnerability triggers as new observability measure The first thing we are planning to introduce is extending the state-of-the-art observability stack with the capability to store information about vulnerable code, executed during runtime. We plan to leverage existing databases of vulnerable callables to determine vulnerable ones within the application's dependencies. Then, we will use metaprogramming and instrumentation to count vulnerability triggers in the central aggregator, along with their metadata like the callable name, vulnerability identifier, and so on. We will provide the operator with a dashboard like in the figure 3.1, allowing them to see at a glance what kind of vulnerabilities are present and triggered within the system, as well as list all the administrative tasks that have to be performed manually (for instance, restarting the application in the external service mesh management dashboard).

Increasing awareness about the runtime vulnerabilities As we discussed previously, the currently used methods of vulnerability assessment rely on static analysis and monitoring used dependencies, without taking into account any runtime information. This leads to false positives, where vulnerabilities in a faulty library might not be called during the application's operation, even though the sole presence of that library was enough to mark the service as potentially vulnerable.

In our solution, we aim to remove the potentiality from the assessment procedure and focus on the vulnerabilities, which are actually triggered by the system. Once the individual vulnerability triggers are stored in the central aggregator, it is possible to not only aggregate and count them, but also determine the exact services within the mesh, which are affected by the particular vulnerability. For each triggered vulnerability we will provide the operator with a dashboard like in the figure 3.2, allowing them to determine the affected services, and help in assessing the impact – for instance, by providing simple metrics like the trigger

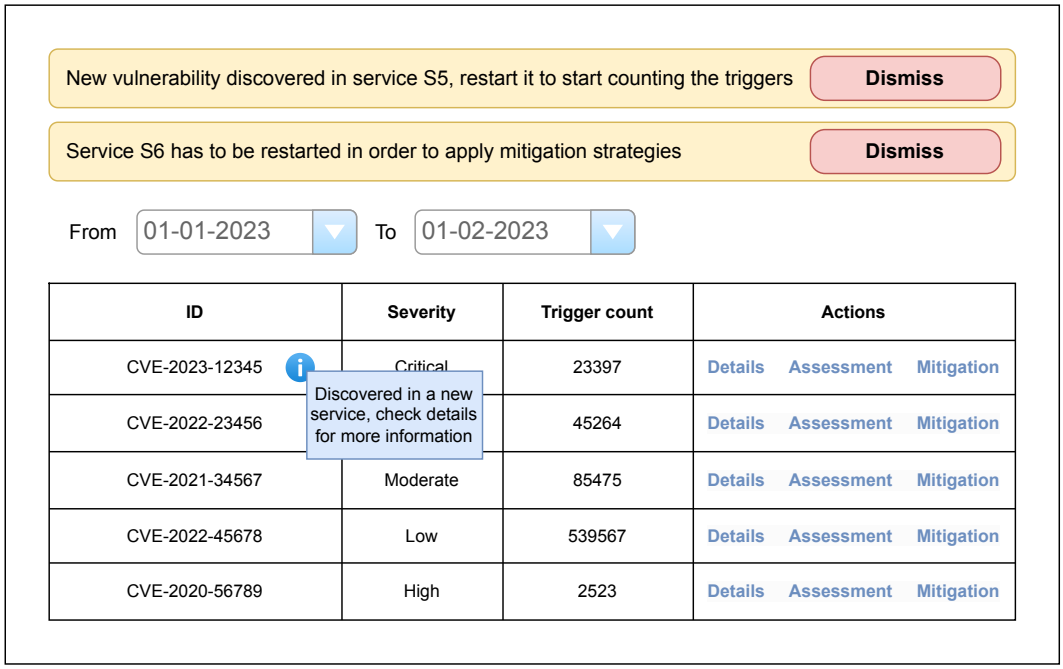


Figure 3.1: Idea of a dashboard for visualizing triggered vulnerabilities within the system

count per service. Additionally, we will display the Maven coordinates of the vulnerable dependency.

Rapid vulnerability mitigation The final goal of the vulnerability mitigation procedure is to eliminate the calls to the problematic code from the developed system's code base. This can be achieved in many ways – for instance, by upgrading the vulnerable dependency to the patched version or modifying the code to avoid executing vulnerable methods. However, as discussed before, this is a time-consuming process, which requires implementing the mitigation, verifying the code, performing tests, and so on.

Having an instrumentation-based solution for reporting the issues is a good baseline that could be extended to perform what we will call an *intrusive instrumentation*. In addition to reporting the triggered vulnerability, the agent could modify the code's standard behavior. This means that simple mitigation strategies could be applied using the same instrumentation toolkit, like replacing the vulnerable method call with an error or a no-op.[1] The error would be propagated in the application's stack and eventually caught and processed by the exception handler. For instance, it would mean that for the code that handles an API request, the response would be an error that can be handled further by the code that sent the request. Meanwhile, the no-op mitigation would remove all the statements from the vulnerable callable to avoid execution of the faulty logic and allow the program execution to

[1] no operation

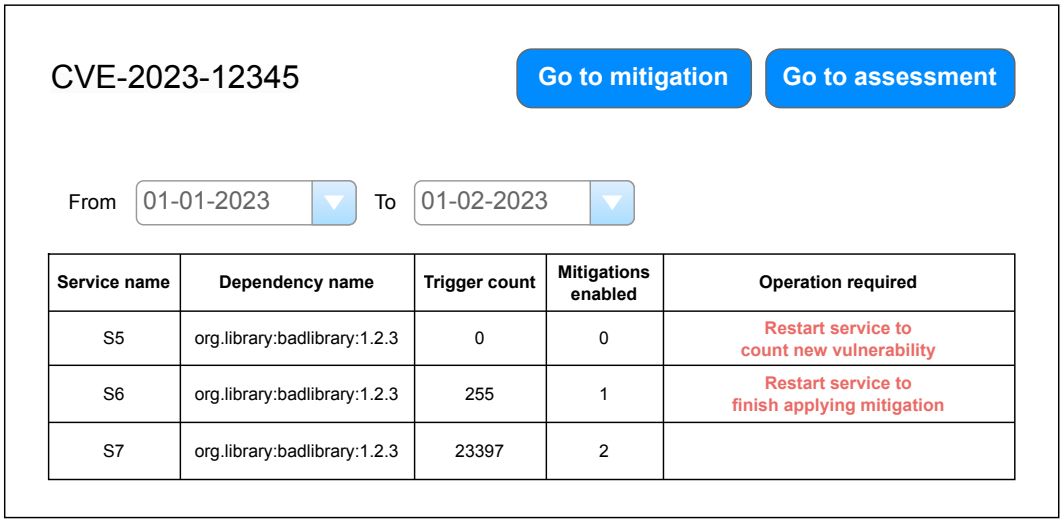


Figure 3.2: Idea of a dashboard for analyzing behavior of individual vulnerability within the system

continue. For functions that return a value, the returned value can be derived from the return type (*e.g.* zero for integers, false for the booleans, null for objects).

The system operator will be presented with a dedicated dashboard for each triggered vulnerability within the system, as shown in figure 3.3. They will be able to choose the service, affected by the vulnerability, and apply mitigation strategies to the vulnerable callable within its dependencies.

Mitigation impact assessment Modifying the code within the dependencies at runtime can lead to unexpected behavior that could have a cascading effect on the entire microservice system. For instance, if a vulnerable method is executed within the scope of a REST API downstream call, executed by another service, replacing the method's code with an error can lead to this error being propagated to the upstream services, rendering the ingress API endpoint of the mesh unusable. Therefore, giving system operators and developers a way to assess which services and their endpoints are being triggered on the way to the vulnerable callable would provide valuable insight when deciding if to mitigate the vulnerability or not.

The simplest way to do it would be to utilize the observability stack and the stored vulnerability trigger information and build an aggregate graph view of the system, showing downstream calls that ultimately lead to the service where vulnerable callable has been executed. This would allow us to determine which services could be affected if mitigation were to be applied. What's more, one could also calculate the metric for the edges of the aggregate graph, and show the amount of requests that were made between the provided pair of services. This metric could be a simple request count, but it could also be the percentage of vulnerability-triggering requests on this particular edge or the percentage of all the requests made to the system within the given timespan. Since the problem of

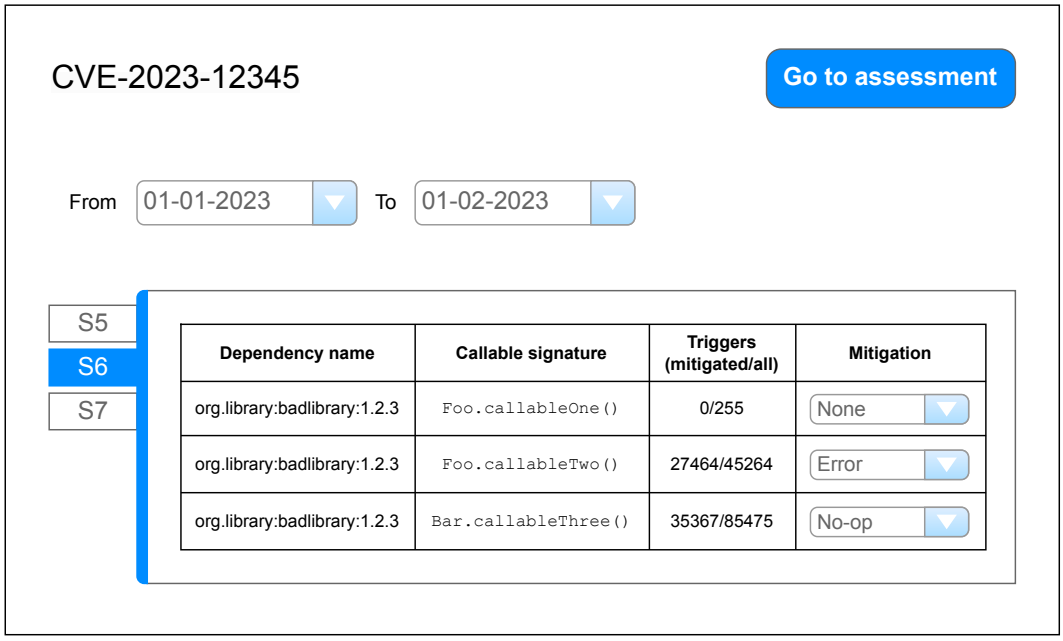


Figure 3.3: Idea of a dashboard for applying mitigation strategies to vulnerable callables

mitigation assessment is largely domain-specific, there should be room for introducing new metrics if deemed necessary.

The visualization of this idea can be seen in the figure 3.4, which shows the operator's dashboard for assessing the impact of vulnerability mitigation. In this example, there are two services S4 and S6, which are affected by the vulnerability CVE-2022-12345, and the problematic code is executed by the method `vulnerable()` in the class `VulnerableClass`. Since the remaining services call vulnerable ones either directly or indirectly, applying a mitigation strategy could lead to errors arising in these services as well. Let's assume that the metric on the downstream call edges is a percentage of calls between two services, which ultimately triggers the vulnerability. Therefore we can say that in the provided example 5% of the calls between S3 and S4 and over one-third of the calls between S5 and S6 call the faulty code. This information, combined with additional insights from the other observability measures (for instance, the executed API endpoint name) could help the operator to decide if the vulnerability in the given service should have the mitigation applied. For example, assuming that S4 is not vital for the behavior of the entire system, the operator could be inclined toward applying the mitigation to it, due to the low percentage of vulnerable requests. The high percentage on the edge between S5 and S6 could be also seen as an argument against applying the mitigation.

Empowering operator with vulnerability mitigation procedure The inherent part of our solution is the dashboard, discussed in the previous paragraphs. It allows operators to take part in the vulnerability assessment and mitigation process, by providing them with

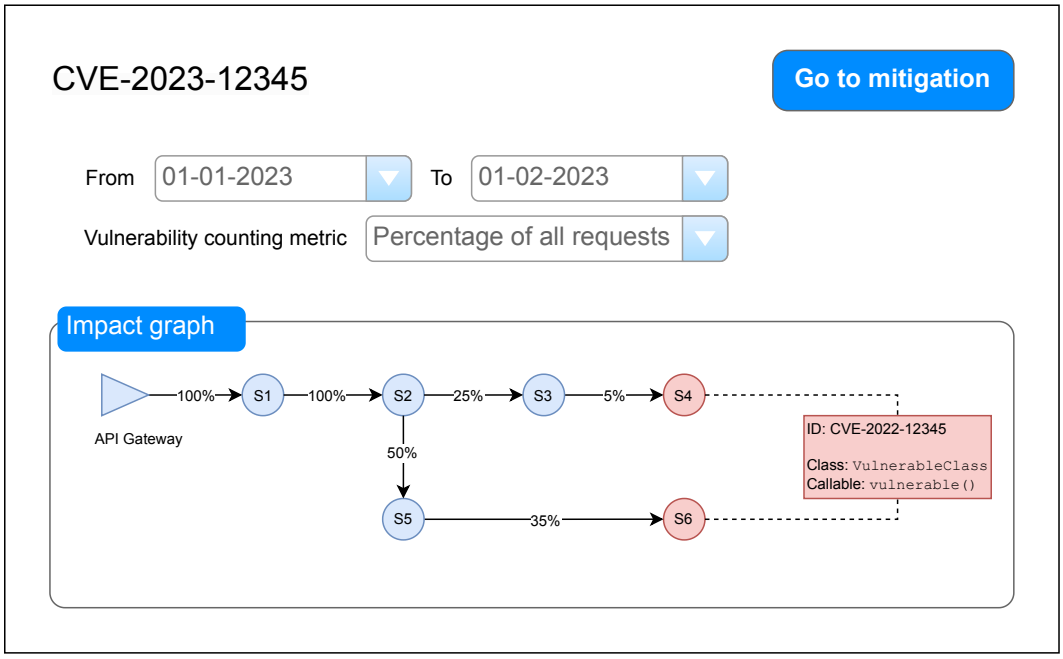


Figure 3.4: Idea of a dashboard for assessing the impact of vulnerability mitigation

a way to browse the triggered vulnerabilities, determine which services are affected, analyze the execution paths within the service mesh that end up executing faulty code, and potentially apply short-term mitigation strategies to give a team of developers extra time to implement long-term mitigation solution.

The procedure from the operator's perspective could look as follows. First, the main dashboard (figure 3.1) informs the operator that a new vulnerability has been detected in one of the dependencies, used by some of the services within the system. Since most metaprogramming frameworks apply runtime code modifications during the application's startup, the operator will be prompted to restart affected services, so that the counting instrumentation is enabled on the new vulnerable callables. After the services restart and counting is enabled, the operator can observe how the trigger count changes (figure 3.2). If the trigger count turns out to be high enough to sanction further investigation, the operator can increase the sampling rate of the observability measure, used to collect runtime vulnerabilities. Once enough samples are collected for a certain period, the sampling rate can be decreased once again and the impact of potential vulnerability mitigation can be assessed by looking at the service graph (figure 3.4). By weighing the criticality of the vulnerability, as well as the type and amount of requests, the operator can decide to apply short-term mitigation to the vulnerable callables, resulting either in errors or no-op behavior (figure 3.3). After the operator restarts services for which the mitigation strategies were enabled, the process of short-term vulnerability mitigation is complete. The operator can then utilize the remaining elements of the observability stack of the service mesh to measure the actual impact of the mitigation, and compare it with the assessed one.

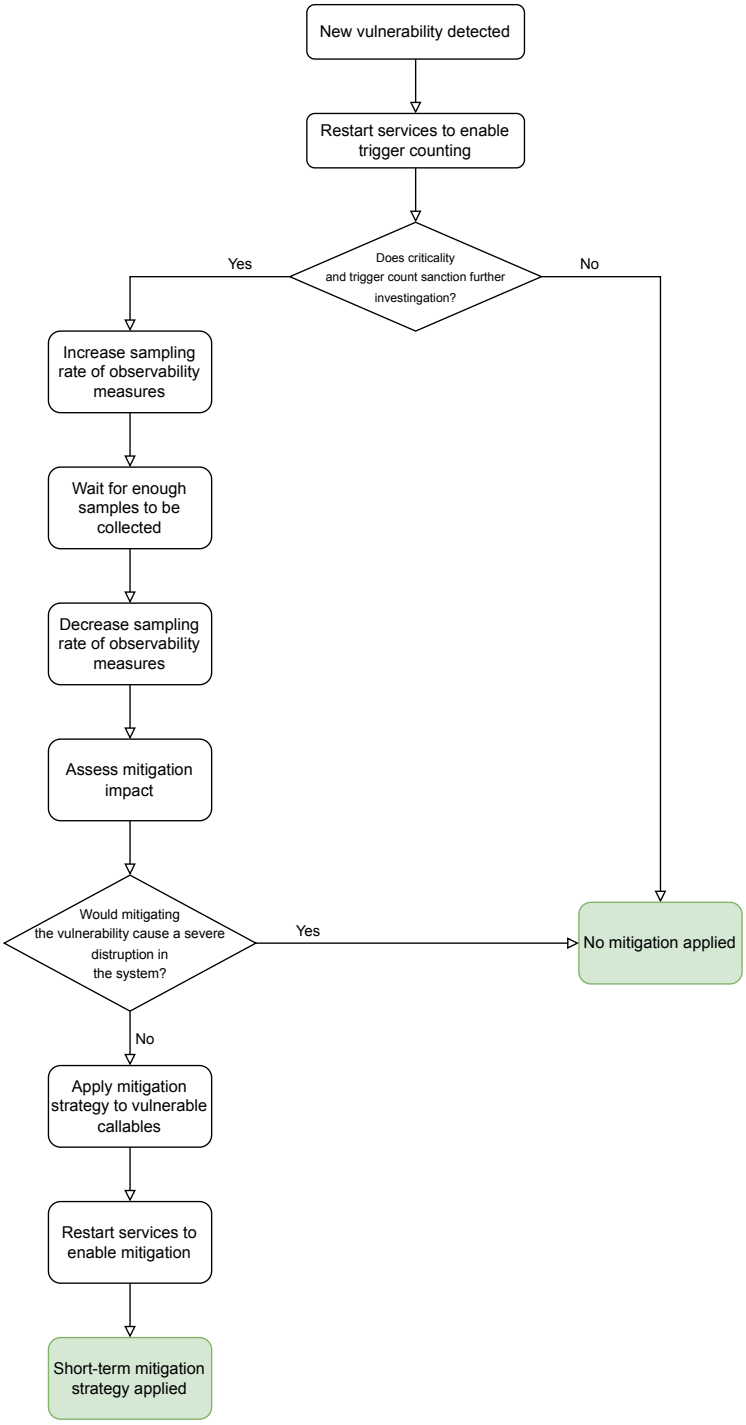


Figure 3.5: Example vulnerability mitigation procedure that the operator could follow by using our proposed solution

Chapter 4

Vision evaluation

Our analysis of modern vulnerability handling practices in software engineering, which we described in chapter 2, allowed us to identify the following key challenges of the operators: *awareness* (knowing about vulnerabilities in the system), *impact* (measuring the impact of vulnerability and its mitigation on rest of the system) and *mitigation* (preventing the faulty code from being executed). In this chapter, we will conduct an interview study among industry practitioners to establish the relevance of these challenges. We will also evaluate the potential of our envisioned tool to address them. Finally, we will perform a debriefing and collect insights, valuable for implementing our solution in a production setting.

4.1 Methodology

An overview of our vision and design evaluation methodology can be found in figure 4.1.

Building the vision As the first step, we looked into related work, done in the software engineering field around vulnerability handling pipelines. This includes building blocks of those pipelines, like used observability stacks, DevOps methodology principles, and so on. It allowed us to determine their limitations and, by extension, improvement opportunities. In our vision, we designed a system that would address those limitations and improve vulnerability handling from the perspective of increasing awareness, measuring impact, and providing the system operator with mitigation capability.

Semi-structured interviews To evaluate the relevance of our proposed solution, we conducted a series of semi-structured interviews with DevOps and software engineers with experience in system operation and administration. We interviewed 10 engineers of varying experience, as depicted in table 4.1.

The interviews were conducted online, and transcribed to the text file for further analysis. Each interview took approximately one hour. During each interview, we asked the participants to look into a problem of system vulnerabilities from the perspective of a system operator, who can configure and monitor the infrastructure, as well as deploy the applications. We also informed the participants that the operator cannot modify the applications'

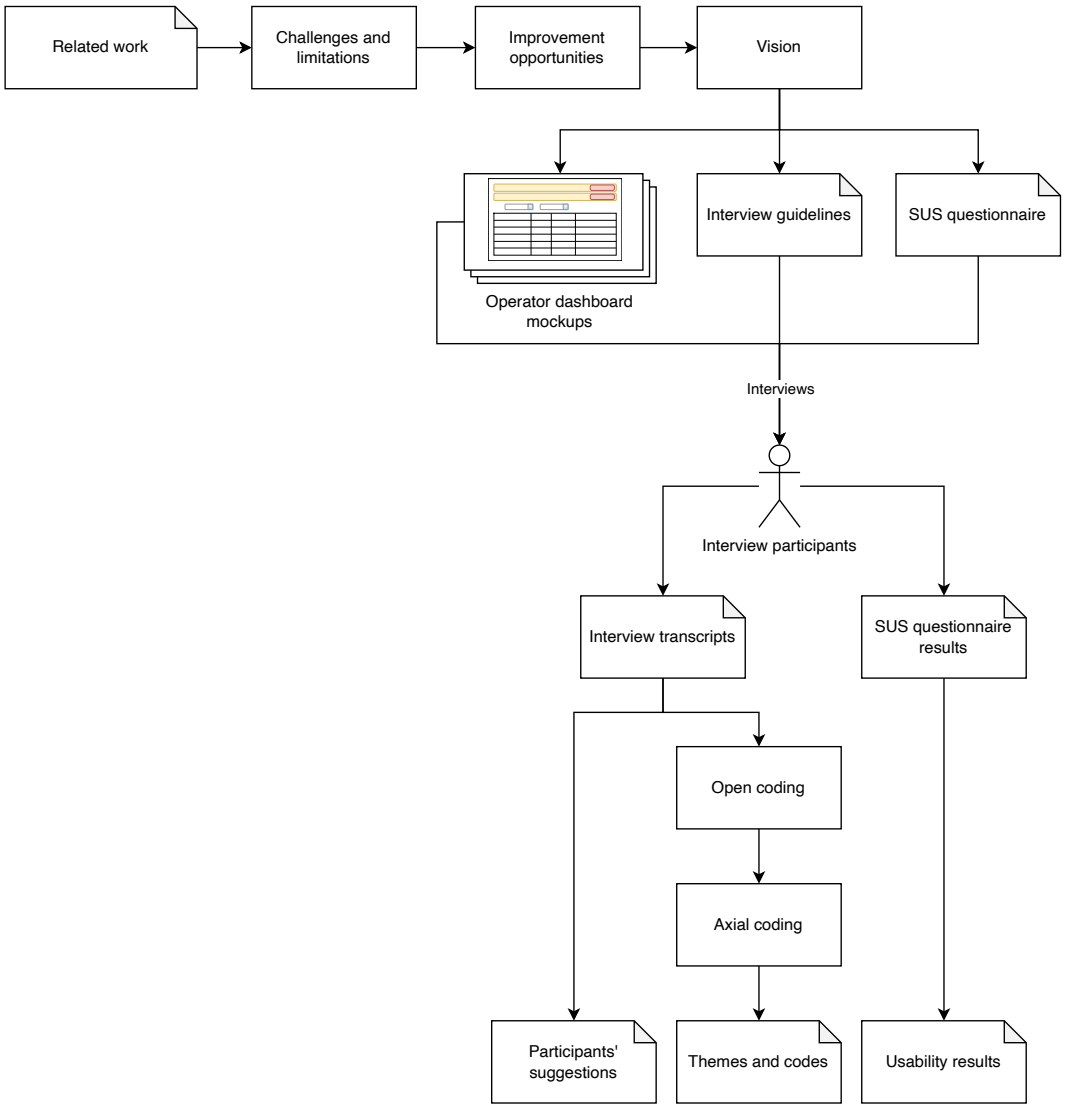


Figure 4.1: Vision and design prototyping methodology

ID	Current role	Industry experience
P1	Junior Software Engineer	1 year
P2	Software Engineer	4 years
P3	Senior Software Engineer	7 years
P4	Software Engineer	2 years
P5	Project Manager	6 years
P6	Software Engineer	6 years
P7	Senior Software Engineer	9 years
P8	Senior Associate	7 years
P9	Junior DevOps Engineer	3 years
P10	DevOps Engineer	2 years

Table 4.1: Interviewed engineers

source code and artifacts. Then we asked seven open questions about the status quo of vulnerability handling in software engineering and its three aspects of awareness, impact, and mitigation. The interviewees responded in accordance with their knowledge and experience.

Afterwards, the researcher guiding the session introduced the proposed system to the interviewees, by describing its goal and functionality. To avoid the moderator acceptance bias [20], we explained that the system was designed within the Software Engineering Research Group of Delft University of Technology. The mitigation of social desirability bias has been achieved by describing the actions taken by an imaginary operator named Albert.

After going through the wireframes[1] of the operator's dashboard interface and describing the content of each screen, the participants were asked to follow a supervised walkthrough, where they played the role of an operator. They were presented with a hypothetical situation, where a critical vulnerability, similar to the Log4Shell[2] appears on the dashboard. During this part, the guiding researcher was streaming the window of a wireframe editor. The interviewees were then asked to follow a think-aloud protocol, and indicate their actions as they "use" the system. Meanwhile, the researcher made changes to the wireframes to simulate the effect of the participants' actions.

After the walkthrough, participants filled out a System Usability Scale (SUS) questionnaire [12] and were asked additional questions about the impact the proposed system would have on their vulnerability awareness, impact analysis, and mitigation capability.

Result analysis We analyzed the interview results qualitatively by performing coding, following principles of the grounded theory [22]. It is a research methodology, used to develop a theory, grounded in systematically gathered and analyzed data. Generating theories from data allows for exploring new perspectives, as well as validating the existing ones. It is also well-suited for studying and understanding processes, by offering insights, grounded in the realities of those experiencing it.

[1] visual guide used in interface design to suggest the layout of fundamental elements in the interface

[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228

Coding is used to systematically break down and conceptualize the data, which allows for the identification of patterns and themes. The initial phase is called *open coding* and revolves around granular identification and categorization of phenomena into *codes* – labels, derived directly from the data and encapsulating the essence of a piece of data. The coded part can be as small as part of the sentence, or as large as multiple sentences. The exploratory nature of open coding encourages researchers to approach the data with an open mind, without considering relationships between the codes, which increases the chance to discover new perspectives and insights into the research topic. The second phase is *axial coding* and focuses on identifying relationships between the codes and grouping them into themes. During that process, the researcher works with the fractured data from the open coding phase and reassembles them to reveal the underlying structure and processes of the studied phenomenon. This includes looking for semantically similar codes and merging them, grouping codes into themes, and dropping codes, insignificant for theory building. The result of axial coding is a taxonomy of themes and codes, where each theme is backed by codes and each code is backed by references to the original data (for example quotes from the interview transcript), providing a framework to explain the phenomena of interest and formulate theories that are grounded in empirical evidence.

To streamline the coding, we decided to complement the manual coding process with large language models. Recent research shows that models like ChatGPT[3] can provide support for the coding process, similarly to the established analytics platforms [46] like nCoder.[4] The main constraints of the models are the short-term memory size and the fact that it works better with deductive approaches, rather than inductive ones [45]. Providing an input that is too large causes the model to lose context and generate unrelated output, making it unsuitable for performing open coding of the raw transcript. The model also struggles with capturing nuances of the data, for instance in the case of sentiment analysis. However, using the model to perform axial coding by providing it with manually derived codes resulted in themes that categorized the codes reasonably well.

To overcome these limitations, we performed open coding and partial axial coding of the transcripts manually, by identifying the codes and merging semantically similar ones. Theme extraction has been done with the help of ChatGPT, by providing it with the standardized prompt and the list of codes, obtained from the analyzed question. We instructed the model to provide the themes as an indented bullet list, with the first level representing themes and the indented one representing the theme's codes. We also explicitly informed the model to not change the codes and to not merge them. The obtained themes were then manually refined and provided to the model again if deemed necessary. We experimented with including and excluding interview questions in our initial prompt. However, we did not observe any difference between the responses to the prompts that contained the content of the question, and the prompts that did not contain it.

[3] https://chat.openai.com/
[4] https://www.n-coder.org/

4.2 Exploratory investigation

4.2.1 Does the operator need to know about the vulnerabilities in the system?

We asked this question to determine the attitude of the participants, regarding the operator's place and responsibilities within the project. We followed by asking about the benefits and drawbacks of introducing the operator into the loop of vulnerability detection and mitigation, to deepen our understanding of the given stance. Answers from participants have elicited the following concepts:

Operator's Role and Information Management in Vulnerability Management This theme emphasizes the need for operators to be well-informed about project happenings and vulnerabilities while managing the balance of their workload to avoid alert fatigue and neglect of other duties.

Operator should be informed about vulnerabilities (P1-P8) All participants agreed that the operator should be aware of the vulnerabilities within the handled system. P2 said that understanding the risks, associated with the application, can help them in making informed choices about the way the application is being deployed. P7 pointed out that knowing about the vulnerabilities in the application code base is not something that the operator can act upon directly. However, they also mentioned that "*there could be circumstances under which operators having that knowledge could improve the overall security of the system*", mainly in the context of forwarding that information to the responsible parties.

Knowledge on what's happening within the project (P1, P2) This code emphasizes the importance of general awareness for operators in the context of the entire project. P2 said that "*it would be better for the operator to understand what are the risks that are associated with the application, so that they can also make informed choices about the way it's being deployed*", showing that having more information about vulnerabilities can be important aspect in helping the operator with their daily tasks.

Forwarding important information to the responsible teams (P1, P2, P7, P8) This aspect revolves around communicating vulnerability information efficiently to appropriate team members. The recurring theme in the discussion was the operator's role in forwarding the information about vulnerabilities to the developer so that the mitigation can be implemented into the code base. P8 mentioned that "*it's not our (operator's) job to fix it, but it's our job to make sure that appropriate team is assigned for the task*".

Avoiding alert fatigue (P1, P4, P5, P8) The main issue mentioned by the interviewees, which could arise when informing the operator about the system vulnerabilities, is the risk of alert fatigue. P5 described how they worked with systems, where many false positive errors led to a situation in which nobody checked the errors at all. This creates a risk where important alerts could be overlooked. To avoid that, the participants emphasize the need for alert prioritization, with vulnerability severity being the metric. P8 mentions that in case of mitigating high and critical severity vulnerabilities "*we are the people who*

usually go to app developers and ask them to prioritize these kind of tasks" and that "*if it's something minor, then yeah, we can wait a bit*".

Neglecting other responsibilities (P9, P10) Another problem pointed out by the participants was the risk of neglecting other operator's tasks, which could happen if too many responsibilities are assigned to one person. P9 said that the drawback of the operator, knowing about the vulnerabilities and having to handle them could make them "*sacrifice time that would be normally spent on the other operational tasks*". P10 expressed their concern that the operator could spend excess time on vulnerability handling "*instead of focusing on the remaining parts of his job, which cannot be done by other people in the team*". They also mentioned the insufficient empowerment of the operator in terms of vulnerability mitigation, saying that they could "*spend time trying to fix those vulnerabilities, and maybe there will be no possibility for them to fix it*", which would ultimately make them "*waste their time*".

All participants ultimately declared that it is beneficial for the project if the operator is aware of the vulnerabilities. In line with the DevOps methodology, the reasoning stemmed from the benefits of general awareness of what is going on in the project. The benefit of having a more holistic view of the occurring problems is being able to tackle them faster since the entire team can be involved in the process.

However, participants also expressed their concern about assigning too many tasks to the operator, which could lead to neglecting other responsibilities. Most of the interviewees focused on the operator's role in forwarding information about the vulnerabilities to project managers and developers, who can then allocate time for implementing the fix in the project's code base.

Team Dynamics, Trust, and Shared Responsibility This theme focuses on the team aspects of vulnerability management, including trust, shared responsibilities, and role definitions within the team.

Shared responsibility in the team (P1, P3, P8) This code indicates that vulnerability risk management is a collective effort within project teams. The cooperation framework is described by the interviewees as an idea, where the operator informs developers about vulnerabilities in the applications so that they can mitigate them in the code base – for instance, by updating dependency to the newer version. P1 mentions that "*the system operator can monitor, whether one of the used libraries in software is vulnerable*" and that "*the operator would be able to notify the responsible party to make changes or update dependency versions*".

Operators responsible for the security of the runtime environment (P7, P8) Some interviewees pointed out that the operator's main responsibility is managing and securing the infrastructure and the runtime environment, where applications are meant to be deployed. P7 mentioned that "*the person operating the environment is not the entity, responsible for the security issues that are related to how the application itself is functioning*" and that "*they are only liable for security vulnerabilities that are related to the infrastructure and*

the runtime environment, but not the application itself". This shows how it's important that operator's new potential responsibilities do not distract them from their daily tasks.

Distrust between the teams (P2, P6) Interviewees mentioned that in some corner cases, it might be necessary to keep the discovered risks confidential. P6 said that "*the more people know about the inner workings of the system, the bigger the risk of exploiting the vulnerability*". However, it seems to be reserved for the niche methodologies, since the majority of participants did not mention it as an issue, and even the ones who did say that "*assuming that we all have shared goals*", they "*prefer the culture of openness*" (as said by P2).

Participants emphasized the role of joint responsibility for developed products, which benefits from the culture of openness. However, they also pointed out that operators are mainly responsible for the infrastructure and the runtime environment of the applications, and not the applications themselves. This shows that although the information exchange within the team is beneficial, it should not lead to a situation where one person has too many responsibilities.

Tools and Response Strategies This theme merges the need for specific tools (like dashboards) with strategies for responding to vulnerabilities, emphasizing both the technological and procedural aspects of effective vulnerability management.

Need for vulnerability monitoring dashboards (P1, P4, P8) Participants mentioned the importance of presenting new information in a way that is familiar to the operator. P4 says that "*there should be a system where administrators could review the vulnerability themselves, see what is its effect, then make a final assessment if it's serious enough to sanction additional steps*".

Early response to vulnerability appearance (P3, P8) Interviewees emphasized the need for prompt reaction upon discovery of a severe vulnerability. P3 mentioned that introducing operators into the vulnerability mitigation pipeline could help in that. They said that "*we should learn about vulnerabilities as early as we can, so maybe if more people are included in the process, maybe that will make it quicker*".

Conclusions The participants agreed that the operator should be involved in the vulnerability mitigation process. Most of them emphasize the operator's role as the first person to be informed about the alerts, coming from the runtime environment of the system. The actionability of the findings is usually limited to forwarding the information to other team members like project managers and developers.

Some interviewees mentioned the possibility of requesting the operator to take down vulnerable components. However, they mostly reserve that solution to the most severe vulnerabilities, since the introduced disruption might be difficult to assess. This creates room for improvement, where giving the operator additional options could further streamline the risk mitigation process. All of this shows that the idea of empowering operators with additional vulnerability handling tools and telemetry can be beneficial, assuming that it does not lead to fatigue and does not interfere with other tasks.

4.2.2 How do you get informed about system vulnerabilities as an operator or developer?

The point of that question is to check the ways of becoming aware of presence of the vulnerabilities in software, from the perspective of two roles with hands-on experience in delivering the software: operators and developers. By analyzing the participants' answers, we established the following themes:

Vulnerability Identification Tools and Methods This theme revolves around methods and tools used by developers and operators to identify and detect system vulnerabilities.

Leveraging online vulnerability catalogues (P2, P8, P10) Participants mentioned the usage of publicly available catalogs and databases to stay informed about new vulnerabilities. Some participants like P10 admitted subscribing to blogs and pages, which are likely to mention disclosures of high-risk vulnerabilities.

Scanning artifacts for vulnerabilities as part of the release pipeline (P3, P5, P6, P8) Many interviewees discussed the usage of static analysis tools, which are integrated into the release process of the developed software, and allow for scanning the artifacts before deployment. P5 and P8 mentioned the tools, which check for vulnerable dependencies in applications. P3 and P6 talked about the usage of containerization as the preferred way of packaging applications in their teams, and how it resulted in implementing Docker image scanning. In this case, the scanner can find vulnerable binaries within the application's runtime environment.

Scanning infrastructure for vulnerabilities during runtime (P7, P9) There was a discussion of how vulnerabilities can occur on different levels of software abstraction – from the level of vulnerable dependencies and binaries in the runtime to the faulty configuration of the components (for example unencrypted connections). Participants mentioned the usage of holistic monitoring software, which scans the infrastructure components in search for this kind of issues.

Running periodic vulnerability scans on the source code (P7, P8) One of the ways of discovering vulnerabilities in the developed software is by running scans at regular intervals. These scans can cover mostly the source code of the developed applications, but can also check other layers of the release stack like "*infrastructure or deployment-related information*", as mentioned by P7.

Based on the answers of the interviewees, we can say that the main way of getting informed about vulnerabilities in developed software is periodic scanning. This idea seems well-established, ranging from simple dependency check, all the way to container image analysis. However, relying on the scans being periodic means that the response time is strongly correlated to the scanning intervals.

Vulnerability Monitoring and Management Strategies This theme focuses on the strategies for monitoring vulnerabilities and the processes for handling them once detected.

The need for centralized vulnerability monitoring dashboard (P1, P4) Whenever monitoring is involved, participants choose a dashboard as a preferable way of being informed about the system state. When asked about the way that developers and operators could get informed about vulnerabilities, P4 answered that "*ideally it would be something like a dashboard*". They also mentioned the benefits of summarizing the current state in the form of a single, easily digestible report, which could be generated by the dashboard.

Company process for handling detected vulnerabilities (P7) Standardized company procedures for handling vulnerabilities seem to be the norm for certain interviewees. P7 mentions that in their case "*the person responsible for supporting the production application is also responsible for checking various things on a daily basis, and that includes the security scans*". This allows them to "*someone will take care of this once the vulnerabilities are detected*".

Keeping track of vulnerabilities, accepted in the code base (P2) Some vulnerabilities might be retained in the code base if they were analyzed in the past and confirmed to pose no risk.

Participants indicated a dashboard as a preferable way of getting informed about the vulnerabilities in developed systems. This shows that whenever a new tool is to be developed, it should either provide its own easily accessible panel or integrate with existing dashboard solutions like Grafana.

Large companies and enterprises often standardize their procedures to ensure repeatable results for all their products. This includes the process of performing security scans and analyzing the results. It often requires delegating employees to perform this kind of task. Smaller companies might not have enough workforce to do this kind of audit regularly. Having a dedicated solution for performing security scans and issuing alerts whenever new risk is detected could be beneficial for both types of companies. In the large ones, employees could be freed from the (often mundane) task of analyzing the reports. The small companies, which previously could not afford to perform this kind of scan, could improve the safety of their software.

Specificity and Scope of Vulnerability Detection This theme deals with a depth of vulnerability detection, focusing on the abstraction layers of the software and the granularity of detection.

Vulnerabilities on different levels of the product (P2) Vulnerabilities can occur in different parts of the stack, used to develop and deploy the software. In addition to dependency vulnerabilities, participants pointed out that it is likely for the developers to introduce the issues directly into the developed software code base. What's more, security issues can also occur on the level of the infrastructure – an issue that becomes more apparent as the popularity of cloud deployments increases. In such cases, the infrastructure is managed by a third party and offered to the clients largely as a black box. The clients have to trust the provider to do their best in securing the offered solutions.

Dependency level as the most granular form of vulnerability detection (P2) Participants consider scanning used dependencies to be the most granular vulnerability detection option

available. P2 mentions that it is "*probably a good level of granularity to identify the vulnerabilities*".

Participants are aware of the fact, that the problem of vulnerabilities in software engineering is not limited to importing flawed dependencies. These issues might arise on all levels of software development

From the perspective of software development, interviewees consider dependency-level detection to be the most granular form of finding vulnerabilities that were not introduced directly into the code base by developers.

Conclusions No participants mentioned the usage of real-time vulnerability monitoring, where alerts are sent the moment a vulnerability is disclosed. This creates an opportunity to further streamline the process and shorten the response time. The presence of infrastructure scanning in search of vulnerabilities shows that multiple companies are already considering extending their telemetry beyond static scans of developed software.

Participants think of maintaining and periodic scanning of the SBOM[5] to be the most precise way of detecting vulnerabilities in developed systems. This means that even if there are solutions, that increase the granularity level to the individual methods in dependencies, they are either not mature enough to be used in practice or too cumbersome to integrate with deployed applications.

4.2.3 How do you think informing about the vulnerabilities could be improved?

We asked this question to identify common issues with the current ways of getting informed about the vulnerabilities in the projects. Analysis of the participants' answers allowed us to extract the following concepts:

Awareness and Knowledge Improvement This theme focuses on enhancing the understanding and awareness of developers regarding vulnerabilities in the libraries they use when making software. This includes increasing the amount of available information and making vulnerability reports more accessible and understandable.

Developers are not aware of vulnerabilities in libraries they use (P1, P8) Participants point out a lack of awareness among developers about potential vulnerabilities in the libraries they use. P8 calls for a need to educate developers about the potential impact of different vulnerability types, saying that "*if they are not paying attention, whether it is using some dodgy dependency packages or not paying attention to your own code, it may result in some major security breaches*".

Insufficient amount of notifications about vulnerabilities (P1) There seems to be a need for more notifications to keep developers informed about vulnerabilities. P1 mentions that in the case of dependency management tools like NPM, the information about imported vulnerable libraries appears only once during project preparation. It does not appear

[5] Software bill of materials – list of used software components, libraries, etc.

again to remind developers about the issues unless additional tools like IDE plugins and analyzers are used and executed periodically to remind developers about the potential danger.

More information to determine the vulnerability severity (P4, P7, P8) According to the participants' answers, detailed information is crucial in assessing the risk associated with a vulnerability. As P7 said, "*the more information we have, the faster we can ensure the overall security*". P8 also points out that "*just reading about some package being broken at the time being, even if it's a critical vulnerability, doesn't really tell me much*". According to P4, the important information would be details of "*what are the possible exploits that could happen*".

Providing vulnerability report in approachable form (P8) P8 talked about making vulnerability reports more useful to all parties by making them "*easy to read, easy to understand what is a top priority and what is not*". The issue is especially important as large systems can consist of hundreds of dependencies, making it difficult to determine priorities.

Participants talked about the lack of awareness in terms of vulnerability presence and impact. The way of notifying about security problems leaves a lot of space for improvement, both in terms of form and frequency. The information about new vulnerabilities is often presented in a way, which is easy to overlook. Moreover, the information itself often lacks details which could help in assessing if a particular issue causes a potential for exploit.

The fact that vulnerability reports are often limited to stating which application dependencies are vulnerable makes them hard to act upon. Since the number of dependencies in large systems can be large, repeatedly including all of them in the reports could lead to alert fatigue. Extending that information with runtime telemetry insights about vulnerability triggers could help in determining priorities, by showing which vulnerabilities are potentially being exploited.

Dynamic and Real-Time Analysis This theme discusses the potential of dynamic analysis in evaluating vulnerabilities' behavior in the system and providing real-time updates about potential threats in dependencies.

Dynamic analysis of the system's vulnerable behavior (P2, P3, P4, P10) Many participants mentioned that using dynamic analysis could help in deciding if to act on the present security issues or not. P3 pointed out that static analysis, executed by developers or delivery pipelines, is a non-exhaustive way to detect vulnerabilities. Moreover, P4 called for the necessity of observing the transient behavior of the vulnerabilities in developed solutions.

Real-time information about vulnerabilities in used libraries (P5) P5 emphasized the importance of receiving timely updates about vulnerabilities in used libraries, saying that "*if you could get real-time information if there is some new issue with any of the libraries that you use, that would be great*".

Analysis of the codes and participants' answers shows that there is a need for real-time alerting solutions. This includes informing about newly disclosed vulnerabilities, as well as the fact if they are being triggered.

Automated Detection The following theme emphasizes the use of automation in scanning for vulnerabilities, particularly in systems that are infrequently updated.

Automated vulnerability scans in rarely-updated systems (P2, P5, P6, P9, P10) Participants called for making security scans more frequent to decrease the response time whenever a new vulnerability is disclosed. The problem was discussed especially in the context of the systems in the maintenance stage, which are not actively developed. Interviewees expressed their concern regarding the ways how often vulnerability detection is performed there.

Automated handling of detected vulnerabilities (P8) P8 mentioned that by collecting enough information about the vulnerabilities it would be possible to automate certain parts of the mitigation workflow. They said that generated vulnerability report "*could suggest some steps you can take to streamline the process even further, and potentially automate it at some point*".

The biggest problem with static analysis, executed by developers and the delivery pipeline, is the fact that systems that are not actively developed will not have this kind of solution executed regularly. Therefore, creating a system that keeps track of the SBOM of solutions in the maintenance stage and monitors their dependency for newly disclosed vulnerabilities would allow for faster reactions to security issues. It would also save computing resources by not having to periodically run the scan for every software component in the company.

Automation of the mitigation workflow could dramatically decrease the response time, but having no human input in altering the infrastructure components and application runtime could lead to unpredictable behavior.

Mitigation Strategies and Planning This concept focuses on the usage of tools and team collaboration to streamline the procedure of mitigating disclosed vulnerabilities and decrease the response time.

Determining the quickest path to vulnerability mitigation (P8) P8 talked about multiple aspects that have to be taken into consideration when reacting to a newly discovered vulnerability. This includes things like checking for the availability of a patched version and making sure that there are no breaking changes. In case there are, developers will be required to modify the code to ensure the same behavior. Sometimes if there is no patch available, it might be necessary to implement an alternative fix – for example, by changing the database driver or a logger of the service. Depending on the code structure, such action might be a simple drop-in replacement or require substantial infrastructural changes. All of these solutions vary highly in terms of how long they would take to implement, either because of the work required for implementation or because of quality assurance procedures within the organization. Therefore, the important task in such a case is figuring out the fastest way to neutralize the problem. As P8 said, "*in an ideal scenario, I would like to know what are the quickest steps we can take to mitigate the issue*".

Improving exchange vulnerability information within the organization (P10) It might happen that the vulnerability under analysis has been already handled by somebody in the

team, or another team within the same organization. P10 mentioned that the creation of a dedicated communication channel within the company could help in exchanging information and avoiding double work being performed by different people if one of them already managed to mitigate the same vulnerability.

The emergence of a high-severity security issue in the developed system requires swift reaction, often under time pressure. From this perspective, having a universal course of action that could be applied to arbitrary security issues would make the decisive process much faster. Even if such a solution would be just a temporary fix, it would give time for developers to implement necessary changes in the code base, without sacrificing quality for the speed of implementation. Moreover, it would allow for discussing these changes on the team and organization level, without the stress of having the vulnerability being potentially exploited in the meantime.

Conclusions Interviewees expressed their concerns about the current state of informing about vulnerabilities, present in the codebase. The main problems were insufficient frequency of checking for new vulnerabilities, as well as lacking information about how the issue could be exploited and pose a real risk.

Participants pointed to real-time dynamic analysis of the application behavior as a solution, which could inform interested parties as soon as the vulnerability is disclosed, as well as show if it is being triggered. These two aspects could determine the real severity of the issue. Moreover, having access to information about the vulnerable part of the code in dependencies (for example, a function or method) could allow for the creation of a tool for dynamic modification of the application's behavior, in a way that makes it impossible to trigger the vulnerability. This would vastly decrease the initial response time and pressure on developers when they work on the fix in the application's code base.

4.2.4 Do developers and operators need a different kind of information about the vulnerabilities in the system?

The goal of this question was to identify what details about disclosed vulnerability could be helpful for application developers and system operators in performing their part of the vulnerability mitigation process. By analyzing the participants' answers, we established the following concepts:

Information Accessibility and Sharing This theme emphasizes the importance of access to information about vulnerabilities for all parties involved. It underscores the need to keep both developers and operators informed about new vulnerabilities so that they can do their best to mitigate the problem.

All teams having access to all information about vulnerabilities (P4, P7) Participants agree that there are no benefits in trying to limit access to information about vulnerabilities, based on the engineer's role. P4 said that "*I don't think that there should necessarily be a limitation, in a way that some information is being shared only with the operators, for instance*". However, at the same time, P8 also mentioned that "*it would still be beneficial*

to have separate views, of how those vulnerabilities are being presented to a developer, and how they are being presented to an operator". P7 talked about how the reports from periodic security scans would be a good starting point for both developers and operators to begin solving the issue.

Merging roles of developer and operator (P2, P7) The majority of participants spoke in favor of transparency and comprehensive information sharing among people in the project, regardless of their role in the software development process. P2 mentioned that in their team the roles often converge and the same engineers are often responsible for both the usual development and the environment administration. Having a holistic view of the problem is therefore beneficial.

Benefits of prior experience in case of role changes (P8) P8 talked about a situation, where a person who used to be a developer could be assigned to perform operational tasks in one of the applications or vice versa. In such cases, the background of the individual can help in finding a solution without involving additional people, which would negatively impact the response time.

The majority of participants spoke in favor of transparency and comprehensive information sharing among people in the project, regardless of their role in the software development process. This stance seems to be caused by the flexibility of roles in the software engineering industry, where one person could start their career as a developer, and then transition to operating the systems. It means that when given enough details, such an individual's holistic experience might be beneficial in making informed decisions about the issue.

Vulnerability Monitoring and Dashboard Usage This theme focuses on the type of tools, used for monitoring vulnerabilities, including the use of specialized dashboards and the specific roles of operators in tracking vulnerabilities during software development and maintenance.

Dashboard for monitoring vulnerabilities (P1, P2, P8) Multiple participants indicated the use of dashboards as the most efficient way of monitoring vulnerabilities. P1 mentioned that they are useful not only for operators but also for developers. P8 talked about how dashboards and direct communication channels like emails complement each other in making sure that the responsible party is promptly informed about the problem and can start acting on it as soon as possible.

Operators monitoring the amount of vulnerabilities within the software (P2) P2 talked about how monitoring vulnerabilities in software is one of the operator's responsibilities. One of the alert-worthy metrics, mentioned by them, is a change in the system's status quo in terms of vulnerabilities. This includes an increase in the number of vulnerabilities – either added by deploying a new version or disclosed when software is already deployed and running.

Operators monitoring vulnerabilities in the application runtime (P9, P10) Operators talked about how one of their main responsibilities in terms of vulnerability mitigation, is checking the application runtime environment for outdated, vulnerable system components and

binaries. This includes monitoring and maintenance of scalable deployment methods like container orchestrators (Kubernetes), as well as cloud deployments.

Operators monitoring request propagation that triggers vulnerabilities (P5) P5 mentioned how operators are mostly concerned with what part of the system is affected by vulnerability. This is more coarse-grained information than what developers usually look for. According to P5, developers are interested in details of particular vulnerability occurrences, for instance, the coordinates of the faulty library.

Although information about vulnerability disclosures and similar high-priority issues should be delivered to all interested parties using direct channels like email or push notification, participants emphasized the role of a dashboard in allowing to see the current state of the system, aggregate the issues, and look into details which might not be conveyed in a single message.

Operators' vulnerability monitoring responsibilities are focused on observing how the number of security issues in the project changes and which applications it affects. The only monitoring method that allows for direct actionability is the possibility of updating the runtime environment components with patched versions.

Developers' Information Needs This theme addresses the information requirements of developers, particularly their need for detailed data about vulnerabilities, especially those associated with dependencies in their projects.

Developers need granular information about vulnerability (P5, P7, P8, P9, P10) Many participants emphasized how developers require details about vulnerabilities to handle them efficiently. In addition to description and severity, developers need information on the availability of the patch and reproducibility steps. P5 said that "*the most important is where and in which package this vulnerability occurs*". P7 talked about going as far as checking if each service can be exploited, saying that "*reproducing the problem to exploit a particular issue could be very valuable information*".

The scope of the developer's actionability is broad, compared to the one of the operator. By having access to the source code and the possibility to compile, run, and debug system components, they can assess the real risk of the issue. However, in order to do it, they need more information than most other roles involved in the software development lifecycle, like project managers and operators.

Operators' Information Needs and Limitations This theme captures the specific responsibilities and constraints of operators in relation to vulnerabilities. It includes their role in monitoring and recognizing vulnerabilities, their preference for broader overviews of vulnerabilities, and their limitations in directly mitigating these vulnerabilities.

Operators being satisfied with coarse information about vulnerabilities (P3, P4, P8, P10) Interviewees' answers suggest that operators may prefer or require only a general overview of vulnerabilities rather than details about them. P10 argumented it by saying that operators "*don't really need those informations because they are not developing code*". When it comes to deployment, P3 said that operators "*only need to decide if it's safe to deploy a new*

version of application or not", which can be roughly estimated from the severity of the discovered vulnerability.

Operators not being able to mitigate vulnerabilities (P2, P3, P8) There is a belief among the participants that operators' actionability in terms of affecting the application's behavior to mitigate vulnerabilities is either severely limited or simply non-existent. When talking about sharing vulnerability details with operators, P8 said that "*operators don't need that (information) since they won't do it (fix the vulnerability)*". P3 says that "*I don't think there's much they can do, once they learn that there's some problem*". On the other hand, P2 mentions that there is an opportunity to empower the operators with vulnerability mitigation capabilities, by offering them issues that could be solved during runtime.

Conclusions Judging from participants' talk about the prevalence of people with both development and operational experience, we can say that any new solution in the area as broad as vulnerability mitigation should focus on solving a problem, instead of aiming to be a tool for a specific type of software engineer. It means that information access should not be deliberately limited by the potential user's role. Instead, it should allow the user to get informed to the best extent of their background and experience. Dashboards seem to be a preferable option for a management and monitoring frontend, due to their usefulness and approachability for all people, regardless of the role.

The monitoring responsibilities of operators and developers are strictly correlated with their roles' actionability. Operators focus on high-level vulnerability behavior like count and system coverage, which can help them assess the impact of vulnerabilities and determine if it's necessary to inform the rest of the team. Meanwhile, developers prefer more detailed insights, which comes from the fact that it's their task to eradicate security problems from the code base. Changing these responsibilities would require improving the roles' actionability – for instance, by offering operators a way to mitigate vulnerabilities in a more fine-grained way rather than shutting services down.

4.2.5 Assuming you discover a vulnerability in the currently developed/operated system, how would you assess its impact on the working system? How do you investigate whether you need to take an action?

We asked this question to determine the actions that can be performed after receiving information about new vulnerabilities in developed software. We also aim to check what is the threshold, after which the vulnerability should be acted upon. The following concepts have been extracted when analyzing the participants' responses:

Vulnerability Assessment Techniques This theme encompasses various methods and approaches used to identify and assess the severity and impact of vulnerabilities within software systems. It includes both proactive measures like testing and analysis, as well as reactive strategies such as searching for solutions and information post-detection.

Searching for the information on the internet (P4, P8, P9, P10) Many interviewees indicated that their first action would be checking online for details about the vulnerability. P8 pointed out that from the high-level perspective, each publicly disclosed vulnerability probably already has enough information to estimate its initial impact. As they said, "*if the vulnerability is documented in the exploit database or whatever it is another online source for those, then it's probably already assessed*". Participants also mentioned that they would rely on the online vulnerability catalogs and community discussions when making their decision, as illustrated by P8's response "*if some people reach a consensus that it is a critical vulnerability, then I'm fine with it. There are a lot of people online who know this stuff, and I would be fairly confident that they know what they are talking about*".

Trying to pinpoint which method is vulnerable (P1, P2, P5, P9) Identifying specific vulnerable callables within the software is a popular problem. One of the reasons is the fact, that vulnerable code in the imported dependency might not be used by the end product. P5 brought an example of a machine learning library, allowing for text and image recognition. If there is an exploit in the part, responsible for image recognition, but only a text recognition feature is used, it might be safe to say that the system is not vulnerable. P2 mentioned that one of the ways to determine if vulnerable methods are executed could be to analyze the results from the code coverage tests, which are capable of counting and identifying the lines the control flow goes through.

Using application metrics to assess impact on the clients (P2) One of the ways of determining the vulnerability impact that interviewees use is by checking application metrics. P2 talked about them as a potential telemetry measure to monitor the behavior of vulnerable applications, and how they affect application users.

Trying to figure out if vulnerability is even triggered (P5, P10) Participants discussed the problem of determining if vulnerable code, discovered in developed or operated applications, is even being executed. P5 mentioned that it might happen that vulnerability "*even though reported, cannot be exploited in your system*". They also said that when it comes to checking if a vulnerability is triggered "*sometimes without the the code knowledge you cannot do that*". P10 also stated that checking it would have to be done manually by the developer.

Manual analysis of other services, affected by the vulnerable one (P2) P2 also mentioned that explicit analysis of communication between the affected service and other services that make downstream calls to it. According to the interviewee, it allows them to determine the potential mitigation impact if the service was to be isolated from the rest of the system or taken down.

Analyzing the input from perspective of being able to trigger the vulnerability (P3) Another way of impact analysis is by checking the format and source of the service input, and if it has a chance of triggering vulnerability. P3 talked about a situation, where the service is not exposed directly to a non-trusted network (like the Internet). Instead, it is being invoked by other services in the mesh. In this specific case, P4 argued that the chance for vulnerability to be exploited is minimal. They also talked about an example with a

vulnerable regular expression parser library, where knowing the format of the input for the parser could help determine the risk.

Trying to trigger and exploit vulnerability on the testing environment (P7, P10) Interviewees talked about an in-depth analysis of vulnerabilities, where the developer measures the impact by checking if vulnerability and its harmful effect can be triggered. They do that by experimenting with affected applications on the local development environment, or on deployed test one.

Participants discussed various methods of estimating vulnerability impact in their software. Considering that many of them rely on checking online resources, there seems to be a lot of trust in the collaborative and community aspects of software development. Most of the discussed impact assessment methods rely on application-specific insights, like metrics and manual code analysis. However, although they are more accurate, analyzing them requires time and extensive knowledge of the system. In the case of some of these insights like metrics, it is necessary to implement exposing them in advance, so that they are available during an emergency.

Response and Management Strategies This theme focuses on how individuals and organizations respond to identified vulnerabilities. It includes considerations of prioritization based on severity, the limitations of current knowledge and practices, and the challenges of continuous vulnerability patching in the code base.

Operator being only able to inform developers about vulnerability presence (P4, P8, P10) According to interviewees, the main responsibility of operators is to inform the software developers team about vulnerability once they learn about it so that developers can perform more in-depth analysis. P4 mentions that after collecting as much information about the vulnerability as possible, the operator could "*show it to the developers so that they can read and discuss it internally, because I don't think that the operator can necessarily guide them. I think that the operator's job most likely would be telling them that there is an issue and they should investigate it in the source code*". P8 also stated that their first reaction would be to inform the developer's team. P10, being an operator, said that "*when I find out that this is some vulnerability in dependencies or libraries, I will speak to developers and tell them that they have to fix this*".

Adjusting response time based on the vulnerability severity (P1, P7) Participants talked about reaction time being correlated with how severe a discovered vulnerability is. This includes both the severity from the database and application-specific severity, assigned during in-house impact assessment. P1 said that whenever they discover a vulnerability, they try to act on it as soon as possible, but "*if it's something very obscure, something that we do not use and has no relevance to our usage of the given library, I think wouldn't be as urgent*". P7 also mentioned that in case the vulnerability is present in dependencies but is not used and doesn't pose a threat, they just classify its existence as a code smell and schedule it for cleanup during the normal software development lifecycle.

No knowledge of runtime analysis and trigger solutions (P2, P5) There seems to be no knowledge among participants about any solutions in the industry that would allow to track

vulnerability executions in the software. After discussing the idea for a code coverage-based solution, P2 said that they are "*not sure how it could be done during runtime, when the application is already compiled*".

Alert fatigue of having to fix every vulnerability (P6) P6 described the vulnerability management strategy at their company, which requires all vulnerabilities to be handled and fixed, regardless of severity. This includes vulnerabilities in unused parts of dependencies or unused binaries in Docker containers.

There seems to be a recurring theme of the operator's limited actionability in terms of working with vulnerabilities. Moreover, participants mentioned that they were not aware of how vulnerability's transient behavior could be monitored during application runtime. The strategies for managing vulnerabilities in developed software vary in strictness. In most cases, the reaction depends on the assessed severity of a particular issue. However, some organizations chose a no-tolerance policy instead, where every vulnerability has to be handled in the same way.

Conclusions Through the answers to our question, we managed to establish typical impact assessment methods performed by interviewees whenever they discover a vulnerability in their developed software. For most of the application-specific methods, it is necessary to assign a task to one of the developers or operators. Not only will this increase the response time, but it also requires the assignees to commit full-time to the task. Defining a more general way of determining vulnerability impact could help in shortening the time before the first reaction, after which a more curated fix can be prepared.

The fact that interviewees were inclined to ignore vulnerabilities in unused parts of dependencies shows a trend, where it is acceptable to have vulnerabilities, as long as their existence and runtime behavior are tracked. It happens that vulnerability affects only a part of the library or dependency, so being able to determine which part of it is responsible for the problem can make a difference between having a critical situation with an active exploit in the product, or merely assigning a low-priority task of upgrading dependency during periodic maintenance release. Judging by participants' responses, they have initial ideas of how to pinpoint vulnerable method triggers. However, they are also aware of a limitation of this solution – it would require high test coverage of the developed application, which is not always the metric that tester teams choose to maximize. Creating a solution where vulnerability triggers are being recorded during runtime would bring additional telemetry to help in assessing the real impact of vulnerabilities in the developed software.

Assuming that a vulnerable service is secure because of the fact it resides in a trusted network can be problematic in certain corner cases. In case of a supply chain attack, a malicious actor could modify the code of a service that makes downstream calls to the vulnerable one. Also, if the service is consuming data from a queue, polluting the queue with malicious requests could lead to a vulnerability trigger. It would be good to have an impact analysis measure, which does not rely on analyzing the input since tracking all invocations of a service could be non-trivial.

4.2.6 Assume that the vulnerability is critical enough to sanction a disruption in the system. Who in your team would be responsible for mitigating this risk? What could they do?

We asked participants about this scenario to establish the actionability of developers and operators in case of emergency, where discovered vulnerability's exploit potential could lead to data theft or other loss. Participants were informed that disabling system components would be allowed, despite potential unavailability resulting from such action. We also asked about the time, required to implement the fix, measured from discovering the problem to deploying the patched version to the production environment. After collecting and analyzing the answers, we established the following themes:

Initial Response to Vulnerability This theme captures who typically takes the first step in responding to a detected vulnerability. It reflects the initial reaction and immediate actions taken to address an emergency.

Operators being the first to try and mitigate the vulnerability (P1, P2, P4, P7, P10) Multiple participants indicated site reliability engineers, DevOps engineers, and other operator roles to be the first ones to try and mitigate the problem. P2 talked about the idea of an on-call engineer, who performs operator duties periodically and actively monitors the system. P4 pointed at the administrative privileges of operators to be the argument for putting them in charge of the first reaction to vulnerability disclosure. P7 mentioned that although developers are responsible for fixing the issue in the code base, in cases that require immediate reaction, the first steps are done by the team, operating the application.

Developers being first to try and mitigate the vulnerability (P1, P2, P3, P6) Some interviewees talked about developers being the first to deal with the problem of detected vulnerabilities. However, the majority of them highlighted that the reason for that is role integration in their teams, where developers are also introduced to administrative tasks in the environment. P2 mentioned that in their case the development and operation of the environments is performed by the same people from the same team. In P6's team developers are responsible for the environment management since most of the infrastructure is defined as a code.

Based on the participants' answers we can say that the initial response to vulnerability is not strictly correlated with the individual's role, but rather with their permissions in the project. Interviewees highlighted that development and operational tasks in their teams are often performed by the same people, so it is difficult to indicate clearly who is responsible for the first action.

Operators' Mitigation Actions This theme focuses on the specific actions that operators undertake whenever a new vulnerability is identified.

Notifying rest of the team about vulnerability (P1, P8, P10) One of the common first reactions, mentioned by participants, is having operators inform other people in their team about the problem. This includes informing developers (as mentioned by P1 and P10), as well as non-technical parties like business users and stakeholders (as P8 suggested).

Rolling back to the non-vulnerable version (P3, P4, P10) In case the vulnerability turns out to be present in a dependency that was only recently introduced into the product, it might be possible to perform a rollback and restore the older version of the software. Such operation would give time for developers to fix the issue, without introducing unnecessary disturbance into the system.

Disabling vulnerable parts of the application (P3, P7) The most granular mitigation strategy, discussed by interviewees, is dynamically disabling vulnerable functionality of the affected service. P3 mentioned the concept of feature flags, which are toggleable properties usually used for enabling and disabling groups of application functionalities, without having to rebuild the artifacts. P7 talked about limiting access to vulnerable parts of services by disabling API endpoints on the reverse proxy level.

Shutting down vulnerable components (P3, P7, P8, P10) As a protective measure, operators sometimes shut down components found to be vulnerable. Being a highly disruptive operation, it is often left as a last resort once other more granular mitigation strategies fail. Although by definition microservices should follow the single responsibility principle, P7 mentioned that typically they are "*a little bit bigger, and it's not practical to just shut them down*".

No ideas regarding how operator could mitigate the vulnerability (P1, P6, P9) Several participants mentioned the existence of a limit to the operator's actionability. P1 said that "*in case there is no fix, I don't think much can be done about it*". P6 and P9 also stated that they are not sure how the operator could actively mitigate the vulnerability.

The operator's actionability, as described by the interviewees, is limited to notifying the remaining members of the team about the problem and basic operational tasks. These tasks range from relatively granular operations of toggling feature flags and disabling application endpoints, all the way to rollbacking the service or shutting it down. During asking questions, some of the participants did not have ideas of how the operator could mitigate the vulnerability.

Developers' Mitigation Actions This theme outlines the various actions developers employ to fix vulnerabilities.

Developers responsible for ultimately fixing the issue (P7, P8, P9) Participants often pointed out that regardless of all the short-term mitigations that could be done by people with different roles in the project, it is developers who are responsible for removing vulnerabilities from the code base.

Quickly mitigating vulnerable code by commenting it out (P4, P5) One of the rapid response methods, mentioned by the interviewees, is the developer's capability to remove vulnerable code executions from the code base and deploy the change, while the long-term fix is being prepared. This varies from simply commenting out parts of the code (as mentioned by P4) to removing the entire library and commenting out its executions (as P5 described).

Updating library to the fixed version (P5) A straightforward response to vulnerability in a dependency is checking for the existence of a newer version, in which the issue is not present anymore.

Changing library to another, non-vulnerable one (P5) Another more involved solution is replacing the vulnerable dependency with a different one. P5 talked about this solution as the last resort, whenever there is a major infrastructural flaw or no simple drop-in replacement for the previous library.

Participants discussed developers' actionability both in the context of temporary solutions and permanent fixes, such as commenting out vulnerable code, updating libraries, or replacing them entirely. The answers further underlined the gap in actionability between developer and operator, where the latter is limited to, as described by P8, having to "*make sure that business users and other stakeholders are informed about what's going on*".

Balancing Rapid Response and Procedural Constraints This concept highlights the need for a quick response in emergencies, which is often impaired by business procedures and protocols.

Business procedures limiting the rapid mitigation options (P3, P7, P8) One of the issues that impair the fast response capability is processes, imposed on the software development lifecycle by the business departments of the organization. They are usually put in place to ensure consistency in handling releases of new software versions and their maintenance within large enterprises. The process usually consists of multiple quality assurance steps like automated static analysis and automated testing, followed by user acceptance tests (UATs) and so on. Participants discussed how procedures prolong deploying even the smallest fix, increasing response time multiple times depending on the size and complexity of the system.

Business importance blocking rapid mitigation options (P8) Some business procedures can go as far as to block the potential for any disruptive short-term mitigation. P8 describes that "*there are components that even if we discover some vulnerability in them, we will never bring them down because they are just too important from the business perspective*".

Skipping procedures during emergencies (P3) Participants mentioned that often the business departments are "*agile enough to allow skipping some procedures in case of emergency*" (as said by P3). This depends on the type of product and the department's attitude. P3 also mentioned the panic factor being an important aspect, leading to procedure skipping to deliver the fix promptly.

Interviewees talked about the procedures and inter-company standards being the major factors in prolonging the vulnerability mitigation process, sometimes even leading to blocking certain disruptive options from being used at all.

We asked participants about the time it takes in their organization to solve the problem of vulnerability, from learning about its existence to deploying the patch to production. P5 said that in case of emergencies, it is usually "*more (time) than we have*". The exact timing varied a lot, being highly dependent on the system's complexity and procedures within the

company. Most interviewees mentioned it would take between one hour and a day for a simple fix and even multiple weeks for changes that require infrastructural refactoring.

Conclusions Participants discussed multiple scenarios and measures that could be undertaken by different people in their teams. However, some of these actions have limited applicability – especially for the operator. For example, the potential for rollbacking a vulnerable version of the application assumes that no changes were made that would make the older version incompatible with the rest of the environment (like database schema changes). Disabling vulnerable parts of system applications relies on prior preparation of toggleable feature flags by developers. On top of that, business departments might also decide against taking disruptive measures to mitigate the vulnerability. This leaves operators in a spot, where the only action they can always undertake is to notify other parties about vulnerability so that they can act on it.

However, the actionability of developers can be also limited by multiple factors. Mitigating the problem by commenting out the code or removing the library is a tedious process, which requires identifying all vulnerability usages manually. Changing the entire vulnerable library assumes the existence of a different one of the same type. In the case of a generic solution (like a database driver or a logger), it might be a simple drop-in replacement. Still, there can be niche libraries without immediately available alternatives, or ones that would require an overhaul of the system's infrastructure.

Interestingly, even though operators are seen as having fewer options in terms of reacting to newly discovered vulnerability, they were mentioned by the majority of participants to be the first ones to potentially react to it. From this perspective, the idea of empowering operators with additional options for mitigating security issues could make sense. Moreover, increasing mitigation granularity to the level of individual methods could help in cases, where more coarse-grained solutions like taking down the service could not be applied because of business reasons.

4.2.7 Assume you have a vulnerability and exploit that allows the attacker to execute arbitrary code. How would you react to that?

This is a follow-up to the previous question. In this scenario, we presented participants with a scenario where one of the most severe vulnerability types emerges in their product – one that allows the execution of arbitrary code in the affected environment. An example of such vulnerability is Log4shell[6], discovered in 2021 in a popular Java logging framework. Instead of asking about the actionability in the context of the team, we asked the participants directly about their first reactions.

After we finished analyzing the answers, the following concepts have emerged:

The Role of Business Impact in Decision Making This theme reflects the influence of business considerations in the decision-making process when dealing with vulnerabilities.

[6] https://www.cve.org/CVERecord?id=CVE-2021-44228

Disruption in the system being a business decision (P1, P2, P5, P6, P8) Multiple participants mentioned that they have to consult the business department whenever their first action could result in system disruption. Then, as said by P1, it would be the department's decision if the exploit risk outweighs the risk of having a system that is not fully operational. P5 talked about how business departments usually do not allow for the system to be non-operational when the fix is being developed. They said that "*there is always the big business perspective, where people from business – if you ask them if they prefer to potentially leak users' passwords, or for sure lose millions of dollars – they will most likely decide that they prefer to have this slight risk*". Usually, as P7 described, the department decides to accept the presence of the vulnerability while prioritizing the development of the patch.

Business priorities potentially postponing the vulnerability handling (P8) P8 talked about the situation, where even in case of severe vulnerability disclosure, the development of a fix might end up being postponed if it interferes with the current business plans. Allocating the developer's time to fix the vulnerability could mean delaying the delivery of business-critical functionalities, which can be time-constrained (for example, a marketing campaign). Even if no system components are being taken down to limit vulnerability's exploit potential, this might lead to what P8 called an "*unrealized loss*" – a situation where the company did not lose any money, but they missed an opportunity to make it.

Participants talked about the constraints, imposed on their actionability by the business departments of their organizations. They mentioned how every action that could lead to system disruption has to be first discussed with the business representatives. In most cases, the decision is to let the vulnerability be present in the system while focusing on patching the problem in the code base. However, it might happen that the ongoing business priorities do not allow developers to postpone their tasks to work on security problems, which delays the delivery of the fix even further, increasing the risk of potential vulnerability exploitation.

Technical Solutions for Impact Analysis This theme focuses on the availability of tools, required for evaluating the impact of vulnerabilities. It emphasizes the need for analysis before decisions regarding vulnerability management can be made.

Need for impact analysis before making a decision (P1, P4, P8) Interviewees talked about the necessity of performing the analysis of two things: the impact of vulnerability itself, and the impact of its potential mitigation. P8 emphasized that engineering and security departments need a good justification whenever discussing disruptive actions with business departments. P1 mentioned using distributed tracing to collect information about how often the vulnerable part of the system is being used. P4 brought up an example of checking what would be the effect of turning off the vulnerable component.

Need for a tool that monitors runtime behavior of vulnerabilities (P3) Taking down system components is a highly disruptive operation, which is often not allowed by the business departments. P3 talked about the necessity to monitor system telemetry to make sure that

the discovered vulnerabilities are not being exploited, especially in cases where short-term mitigation is not possible.

Interviewees talked about the importance of an analytical approach whenever emergencies like highly severe vulnerabilities occur. The impact analysis methods, used by them, range from empirical experiments like disabling system components to informed analysis using existing telemetry measures. Regardless of the method, they emphasized the need for observability in making decisions and discussing the next steps with business departments.

Conclusions The answers to the analyzed question underscore the need for analyzing the vulnerability impact, both from the perspective of exploiting the issue and its effect on other system components whenever mitigation is put in place. They also show how business priorities, rather than purely technical factors, can significantly impact how vulnerabilities are addressed and potentially halt the mitigation process entirely.

Therefore, it would be good to have a monitoring measure that could allow for monitoring of the transient behavior of vulnerabilities in the systems. It would help in determining if the vulnerability is potentially being exploited, based on the amount of vulnerable method's triggers. What's more, combining this kind of information with highly contextual telemetry measures like distributed traces could help to verify the impact of the potential mitigation.

Moreover, introducing a short-term mitigation option that could be applied by operators would decrease the burden on the developers. It would also increase overall actionability whenever ongoing business priorities render developers unavailable to tackle the problem promptly.

4.2.8 System usability questionnaire

Participants' answers to the System Usability Scale [12] questionnaire were positive. The mean SUS score is 87.25, with an average standard deviation of the answers equal to 0.71. This score is vastly above the common average value of 68, earning the highest usability grade [37] and being universally considered to be a sign of a good usability design [2] [3]. For example, the statements "*I think that I would like to use this system frequently*" (question 1) and "*I felt very confident using the system*" (question 9) both scored 4.6/5.0 with a standard deviation of 0.52. However, the answer to the statement "*I found the system unnecessarily complex*" (question 2) shows that there is space for improvement in terms of how the data is displayed. The distribution of the answers to ten standard SUS questions can be seen in figure 4.2.

4.2.9 Wrap-up questions

After the initial questions, system walkthrough, and SUS questionnaire, we asked several wrap-up questions before finishing the interview. The goal of these questions was to assess how the interviewees' perception of vulnerability handling has changed after presenting them with the envisioned system. We focused on the personal aspect, by asking how the system would affect their awareness, actionability, and impact assessment regarding vulnerabilities in their operated solutions.

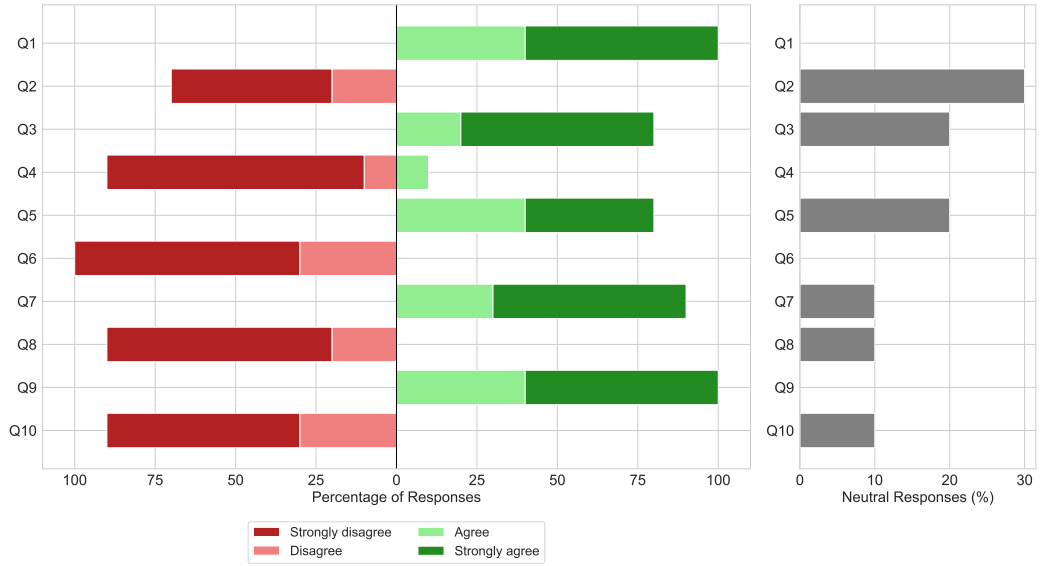


Figure 4.2: Distribution of answers to the SUS questions

Assuming you would have access to such a tool, how would it affect your awareness about existing vulnerabilities?

All participants agreed that introducing the envisioned system into their workflow would have a positive effect on awareness. P2 and P6 emphasized the usefulness of the system in pinpointing if a particular vulnerability is being triggered and poses a real risk. P5 mentioned that the introduction of the system would be particularly beneficial in the case of the older systems in maintenance mode, which are not actively developed. In such systems, the continuous integration and delivery pipelines are not executed often, which means that if vulnerability detection is part of the pipeline, it will not be executed until a bug fix or new change is developed. P7 talked about the usefulness of the envisioned system in testing environments, allowing the team to monitor if the normal operation of developed microservices triggers vulnerable code before changes are deployed to the production environment. They also mentioned how it could provide additional insights during penetration tests.

Assuming you would have access to such a tool, how would it affect your impact analysis of the vulnerabilities?

Participants unanimously stated that the envisioned solution would help them in measuring the impact of vulnerabilities in their developed systems. Some interviewees pointed out that they have not seen a tool that provides a comparable level of granularity in tracking the transient behavior of vulnerabilities in live systems. They highlighted the usefulness of embedding trigger information in distributed traces, allowing them to determine not only the vulnerable service but also its vulnerable endpoints. When describing how the solution would help them, interviewees focused on the granular assessment of counting vulnerable

method triggers, as well as the holistic overview, provided by the impact graph. P1 mentioned that "*seeing which services exactly are affected, how much traffic goes through them, and how many trigger counts there are would be very helpful in making decisions on how to mitigate the issue*". P4 highlighted the potential of the solution to serve as a learning tool that shows developers how vulnerabilities can affect the deployed system. P7 and P8 spoke about the usefulness of the impact graph in prioritizing mitigation actions. P9 and P10 emphasized how the provided solution would vastly shorten the time, required to determine the vulnerability's behavior and impact on the entire system.

Assuming you would have access to such a tool, how would it affect your vulnerability mitigation workflow?

Interviewees collectively underlined the novelty of method-level vulnerability mitigation during application runtime, and how it would be a valuable addition to their current mitigation pipelines. They emphasized that it would substantially decrease the reaction time whenever a new vulnerability is discovered. P10 pointed out that in some cases it would even allow to mitigate the vulnerability without having to consult developers and assess the business impact. P3 described the situation, where discovering vulnerability on Friday could mean that the first mitigation attempts would happen only on Monday, giving potential attackers an entire weekend to try and exploit the vulnerability. They pointed out that having access to the immediate, fine-grained mitigation capability eliminates that risk and allows developers to work on fixing the issue in the code base without the extra stress of knowing that the code could be exploited on production.

Would you use such a tool?

All participants said they would like to use the envisioned solution with their projects. The biggest mentioned benefit is shortening the initial response time, achieved through quick mitigation capability which, according to P3, is "*incomparable to anything available on the market right now*". Another discussed advantage is the aggregation of all the information about vulnerabilities and their behavior in the operated software into one place. P4 mentioned that the solution would be very valuable for small companies and startups that do not have enough resources to hire security specialists, and whose software engineers are usually too busy developing new functionalities to perform security audits.

P5 emphasized the solution's ease of use, and how it could be used by more people than just developers and system operators if they have enough technical background. Participants mentioned how it would be beneficial to integrate our solution with DevOps gradual deployment strategies like A/B testing or canary deployments, allowing for testing the applied mitigation before it gets applied globally.

Do you think there are organizations that couldn't use it?

We asked interviewees if they could foresee any organizations that could not incorporate the proposed solution into their vulnerability handling pipeline. Some participants indicated the legal or compliance issues, likely related to the determinism of the executed software. P2

and P7 talked about the problem of trust, where product stakeholders could oppose the idea of introducing a system that alters the execution of their services.

However, most participants said that they did not see any obstacles to using it. P8 emphasized that even without the granular mitigation capabilities, our solution still provides valuable and actionable information.

How do you think it would affect the vulnerability mitigation done by the developers and operators in your team? Would it affect the responsibilities of the people? Would it affect the actionability of the findings?

All interviewees agreed that the introduction of the envisioned system into their teams would have profound effects on their vulnerability handling procedures. They emphasized the beneficial impact of the system on the team's actionability whenever a new vulnerability is disclosed or exploited, highlighting how applying mitigations through instrumentation allows for avoiding the execution of lengthy tests and building pipelines in emergencies. P2 talked about the benefits of real-time monitoring of vulnerability exploits and how it helps in prioritizing the fixes and avoiding the situation, where excess resources are assigned to mitigating the vulnerability which is not being actively exploited. Instead, these resources could be allocated to ongoing business tasks or preparing a long-term mitigation.

Participants mentioned that empowering the operator with the possibility to apply short-time vulnerability mitigations would allow them to act without having to rely on developers. P3 and P4 emphasized how shifting the responsibility to neutralize the risk of vulnerability exploitation from developers to operators gives developers additional time to investigate the vulnerability and prepare a high-quality fix that goes through all the quality assurance and testing pipelines.

P6 discussed how the introduction of the system could alter the team composition, by decreasing the number of on-call security specialists required to monitor the security of the system and respond to the alerts.

4.2.10 Evaluation debriefing

During the interviews, we collected several aspects that were raised by the participants during discussion, regarding the production-grade application of the envisioned solution.

Integration with external systems Interviewees emphasized the need for the envisioned solution to be able to integrate with already existing tools. P2 talked about the possibility of using distributed traces with vulnerability events as the input for the company's internal machine learning models for threat detection. P8 indicated the potential of integrating vulnerability detection with project management software like Jira[7], where a ticket could be automatically created and assigned whenever a new vulnerability is detected or exploited.

Security P6 expressed their concern about the instrumentation capabilities of the agent being driven by the content of the database, saying that "*with great power comes great*

[7] https://www.atlassian.com/software/jira

responsibility". They noticed that malicious actors who are sufficiently informed about the structure of the code base could modify the content of the system's database and apply mitigation to critical components of the instrumented system. Creating a production-ready solution requires a thorough investigation of that aspect to mitigate the risk of unauthorized instrumentations.

Neglecting long-term mitigation Participants pointed out that shifting short-term mitigation responsibility from software developers to operators decreases the urgency of fixing the problem in the code base. Some of the interviewees raised a concern that once the mitigations are applied by our solution, there will be no incentive to schedule the long-term fix implementation. Ensuring the short-term life of the mitigation is therefore crucial for the quality of developed software. It could be accomplished for example by periodically providing operators with a report, showcasing all the enabled mitigations and for how long they have been in place. This would allow for identifying stale mitigations and help to prioritize fixing them.

Dependency quality control Some interviewees voiced their worry about the potential impact of the envisioned system on the software development process. P4 said that the possibility of mitigating vulnerabilities during runtime by system operators could make developers careless in terms of choosing libraries and their versions. Whenever a new dependency is about to be included in the developed service, it should be inspected in terms of how well it is maintained and how often it receives security updates. Knowing that any vulnerability could be mitigated by the operator can lead the developer to choose an arbitrary version of the library, compatible with the remaining components of the service. Moreover, if taken further it could lead to neglecting periodic dependency updates. Such an attitude would directly lead to a decrease in the quality of the code base. Therefore, it is crucial to stress the difference between short-term and long-term mitigation options.

4.3 Results and reflections

The evaluation of our vision showed that challenges, identified by us in the previous chapter, are relevant and acknowledged by industry practitioners. The discussions highlighted that while system operators are often the first line of response to security alerts, their capacity to act is generally limited to notifying other team members, or performing coarse-grained disruptive operations like taking down vulnerable components. This further confirms the need for empowering operators with additional capabilities to mitigate discovered vulnerabilities.

There is a notable interest among practitioners in terms of real-time callable-level vulnerability monitoring, which helps to ensure that mitigation efforts are commensurate with the actual risk, posed by a vulnerability. This approach has been acknowledged by the interviewees to be beneficial in prioritizing patches during project planning, and in monitoring components that are not actively developed. Interviews revealed that our envisioned tool has the potential to streamline the vulnerability handling workflow not just for the operators, but also for the remaining members of the team. The usage of a dashboard provides

a common ground for users with varying levels of expertise, provided that they have a sufficient technical background. Although the developer's actionability is higher due to their access to the source code, they also have been indicated as the party, who could benefit from our tool's precise and fast mitigation capabilities. Moreover, its envisioned impact assessment capabilities have been endorsed as a novel way of determining vulnerability's transient behavior, further helping to plan the issue mitigation.

All the participants unanimously concluded that it would be worthwhile to develop the envisioned system into a working prototype. In the next chapter, we will describe the implementation details of the envisioned prototype.

Chapter 5

Implementation

In this chapter, we will discuss how we implemented our vision into a functional prototype. We start by describing how the insights provided by the FASTEN research project allowed us to create a base for starting our research. We will discuss why we chose distributed traces for storing triggers of vulnerable callables. Next, we present the architecture of our solution and the role of its components. We also cover the implementation of an instrumentation agent, which enables embedding vulnerability triggers within traces and applying callable-level mitigations in Java applications. Further, we delve into the implementation of the instrumentation coordinator module, which acts as the source of vulnerable callables and mitigation strategies for the agent. We discuss how our prototype collects and processes distributed traces to enable counting vulnerability triggers. Finally, we will introduce the concept of trace impact graphs, how we use processed traces to build them, as well as how they allow us to build visualization and calculate metrics for impact assessment.

5.1 Structural primitives

Before we start with the prototype implementation, we have to determine the building blocks of our solution. To detect executions of vulnerable callables, we need to determine which ones are triggering faulty code. Afterwards, we have to find the most suitable way to store vulnerability executions, in a way that provides the most context information about the circumstances of the trigger. Finally, we will derive a framework to help in assessing the impact of vulnerability on the operated system.

5.1.1 Determining vulnerable methods

The process of disclosing the presence of software vulnerability consists of several steps. The newly identified vulnerability is classified according to its type (*e.g.* buffer overflow, SQL injection) and severity. Severity assessment measures like the Common Vulnerability Scoring System (CVSS) are commonly used to rate how critical a vulnerability is based on factors like its exploitability and impact. Then, the vulnerability is documented with a clear description, affected software versions, steps to reproduce it, and so on. Each vulnerability is assigned a unique identifier, such as a CVE (Common Vulnerabilities and Exposures)

number, to ensure consistent referencing across different platforms. The vulnerability is then reported to the relevant stakeholders, like the internal teams, responsible for the development of the affected software. Finally, once the fix is developed and the patch is available, the vulnerability is disclosed to a wider audience.

The problem with this approach is the granularity. The vulnerability catalogs contain the names and versions of the affected libraries, but no structured information about vulnerable code. This means that relying solely on the information from these catalogs would require any solution that uses them to treat all invocations of the vulnerable library as potentially problematic. Such an approach would likely lead to many false positives, as empirical research has discovered that a very small amount of APIs, exposed by the libraries, is used in projects that use them as dependencies [43]. Therefore, to make our solution usable, a more granular approach was needed.

To uniquely identify a method in Java's classloader, it is necessary to know the name of the method's enclosing class, the package of that class, as well as the method's signature. The signature consists of the method's name and the number, type, and order of its parameters. Although other solutions like Eclipse Steady[1] also provide this information, we decided to use the API provided by the FASTEN project [1] due to its ease of integration. FASTEN system exposes an API, which can be used to query for vulnerabilities in the provided Java libraries by using their Maven coordinates: group identifier, artifact identifier, and version. Each vulnerability can be then queried for vulnerable callables, returning all aforementioned method metadata. This opens a way to utilize metaprogramming for executing custom logic whenever the vulnerable callable is executed

5.1.2 Persisting vulnerability triggers

The next step is determining the most suitable way of persisting vulnerability triggers. An optimal way to approach that would be to extend the existing observability stack with the capability to store information about the vulnerability triggers, in a way that provides the most information about the context of every individual trigger of the faulty code. It's also important to consider the extensibility potential of the chosen observability measure, so that additional domain-specific extensions can be added in the later stages, allowing for more precise mitigation impact analysis.

Logs Logs could be used for that purpose, by adding a new entry whenever a vulnerable callable is executed. This creates an opportunity to embed metadata like the callable's signature, vulnerability identifier, severity, and so on. However, due to the inherently non-structured nature of logging, the output format would have to be standardized to allow parsing logs during data analysis. Applications often rely on logging frameworks (in the case of Java these are Log4j, `java.util.logging.Logger`, and so on), which format the output in a structured way. However, this does not guarantee the stability of the output – even though every application can use the same logging framework, they can have different output formats defined, often depending on the higher-level framework that uses the logging

[1] https://projects.eclipse.org/projects/technology.steady

one (for instance, Spring Framework) and the business use case (for instance, a session identifier). This means that the mitigation impact analysis software that parses the logs would have to be configured to handle the output formats of all the microservices within the mesh, resulting in a huge maintainability burden. What's more, using logs for reporting limits the analyzed context to the scope of individual service instances, which makes logs unsuitable for the service mesh-level impact analysis.

Metrics We can rule out using metrics for this purpose as well. Applications provide their metrics either by sending them to the central aggregator, or by exposing them on the endpoint, which is periodically scraped by the aggregator. Each metric is stored as a stream of timestamped numerical values. Metrics can have labels, which are used to store additional metadata in a key-value format. Each metric and its combination of labels creates a separate stream of data. For example, there might be a single metric for memory usage of the Java virtual machine (JVM), with separate labels for each memory space. However, if the label values are not from the constant pool, this poses a risk of cardinality explosion. Unlike the metric values, the labels are cached, and cardinality spikes can lead to memory errors and crashes on the aggregator's side. In the case of JVM memory spaces, the labels always take the same values. However, if the label were to be something like the vulnerability identifier, there is no limit to how many identifiers could be reported by the system. What's more, the context scope would be even more narrow than in the case of logs. The quantitative nature of metrics would allow only for counting the triggers, without information about the events, preceding and following the trigger occurrence within the scope of a single thread.

Distributed traces Distributed tracing was developed to tackle the problem of tracking the flow of downstream calls within the microservice system. A *trace* is a record of a unique transaction, handled by the system's entrypoint, triggering a certain amount of downstream services as the transaction is being executed. In the case of the system that exposes a REST API, the transaction would be a single call to the API gateway. Each trace consists of *spans*, which are single operations, performed in the context of a trace. These can be downstream service calls, individual method executions, database queries, and so on.

Tracking of a request within a microservice system is done by propagating a unique tracing identifier, assigned to the request on the entrypoint – usually by the service mesh sidecar. In the case of REST services, an identifier is injected into the request's headers.[2] Each service within the system must be programmed to propagate this header value to all the downstream services.

Since service mesh sidecar proxies intercept all the ingress and egress traffic of the service instances, they report encountered requests as spans to the trace aggregator like Grafana Tempo.[3] The aggregator is then responsible for combining all spans with the same tracing identifier into one trace. Applications are also able to report spans on their own, for example by creating spans for individual method executions.

[2] https://github.com/openzipkin/b3-propagation
[3] https://grafana.com/oss/tempo/

Tracing provides mesh-level analysis capabilities, by allowing to track downstream propagation of individual ingress requests within the microservice system. It means that by embedding information about vulnerability triggers into a trace, it is possible to assess the impact on the entire system – for example, by determining the API endpoint of the very first service that started processing the request, as well as the services triggered on the way to the vulnerable one.

Moreover, trace spans can have events – recorded points of interest within the span. Each event has a timestamp and set of key-value attributes as its payload. This makes them a good candidate for tracking triggered vulnerabilities within the running system, by adding new events to spans whenever a faulty callable is executed. Event's attributes can be used as a metadata store, and contain information like the triggered vulnerability's identifier, callable name, dependency name, and so on.

All things considered, distributed traces are the most fitting observability measure to be extended for our research purposes, thanks to their broad context scope and extensibility through the events.

5.1.3 Using distributed traces for impact assessment

Service meshes provide visualization of communication between individual services thanks to the telemetry, collected from application sidecars which monitor all ingress and egress traffic of each application instance. Dashboards like Kiali represent these relationships as a directed graph, with nodes as services and edges as downstream calls.

Trace graphs In distributed traces, the spans can be nested within other spans. Each span has an attribute that determines the service it came from. This allows to represent traces as directed graphs as well, by connecting each nested span to its parent. The nesting and service name attributes allow such a graph to visualize the communication between services in the mesh. Moreover, if the span has a vulnerability trigger event, we mark it as vulnerable. We call this representation a *trace graph*.

Impact graphs This intuitive idea led us to envision visualizing the impact of vulnerabilities by building and displaying an *impact graph*. We define an impact graph as a subset of the service mesh one, with vulnerable service nodes clearly labeled. We determine if a service is vulnerable by collecting trace graphs from a specified time interval and checking if any graph had the span of that service marked as vulnerable. To connect the nodes, we build the impact graph edges by aggregating trace graph ones. Each edge of the trace graph can be seen as being part of the path to the vulnerable code invocation or not. If the edge is part of the path from the root span to the vulnerable one, we consider it to be vulnerable. Therefore, by aggregating the trace graph edges between services s_1 and s_2, we can determine how many traces are involved in communication between s_1 and s_2, ended up triggering vulnerable logic.

The idea behind the impact graph is to visualize which services in the mesh are vulnerable and to show which ones trigger vulnerable logic indirectly by making downstream calls. Graph edges can provide an additional telemetry measure, helping to determine the number

of requests that trigger vulnerability. The aggregate can be visualized by showing a ratio of how many traces on the edge between two services ended up triggering the vulnerability out of all the traces on that edge. This ratio can be also represented as a percentage. The provided aggregation strategy and metrics are examples of what can be done with collected and annotated trace graphs. We do not claim these metrics to be effective in determining vulnerability impact in every scenario.

5.2 Prototype description

Being able to collect vulnerable callables, having a way to store them in distributed traces, and knowing how to work with traces to determine the vulnerability's impact allows us to proceed with implementing our vision.

5.2.1 System architecture overview

Our prototype is a multiservice system, consisting of multiple components. The component diagram is depicted in figure 5.1.

Agent The agent is responsible for the instrumentation of workloads. It registers the service with the coordinator and receives the list of vulnerable callables from it. Each callable is then instrumented to create an event in the currently processed span whenever it's triggered. Depending on the configured mitigation strategy for the callable, the agent can also replace its logic to avoid executing faulty code.

Coordinator The instrumentation coordinator is used to register services and provide their agents with the list of vulnerable callables for each service's dependencies. The callables are fetched from repositories like FASTEN.

Manager This component provides a dashboard website and allows the operator to explore the behavior of vulnerabilities, detected in instrumented services. It also allows the operator to apply mitigation strategies to individual callables within services.

Trace collector The collector component fetches trace identifiers for each service that has been registered in the coordinator and persists them in the database.

Trace processor The processor component queries the trace aggregator to fetch the spans and events of the traces, whose identifiers have been obtained earlier by the collector. Then, it uses the fetched events and span hierarchy to build trace graphs and persist them in the database.

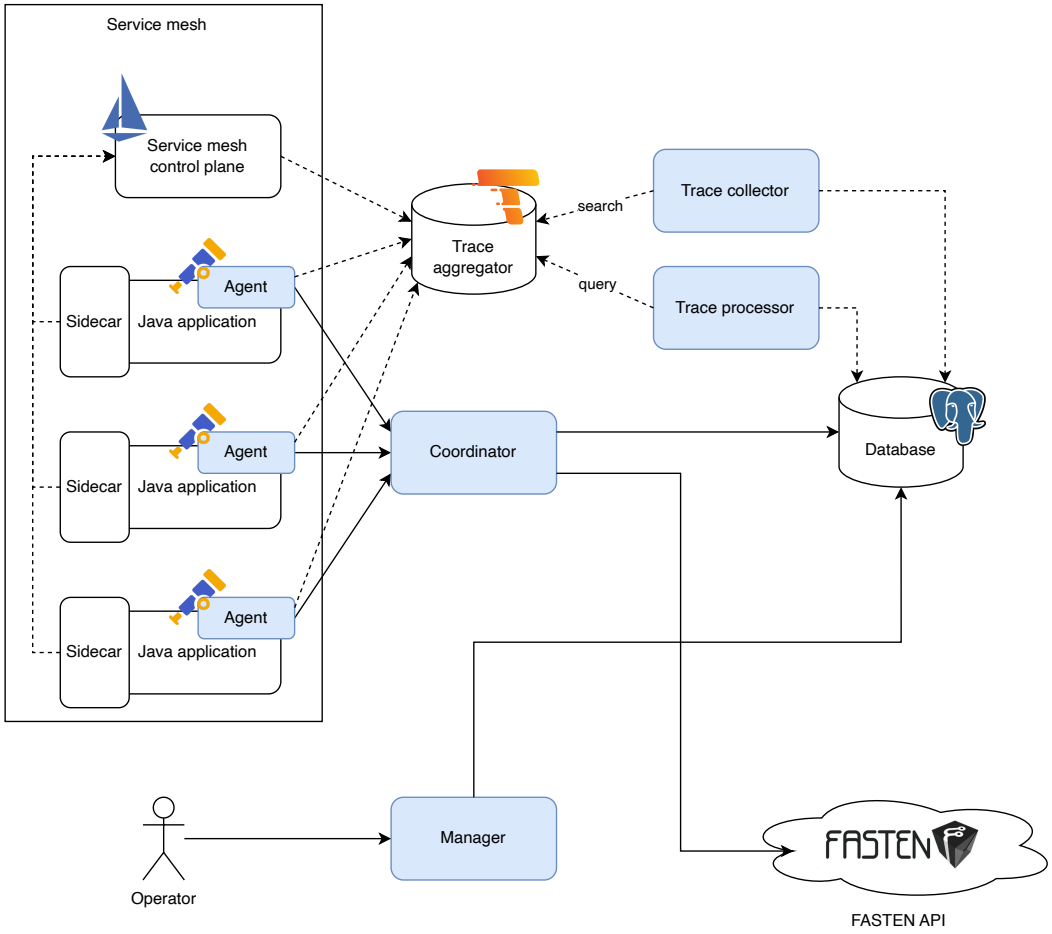


Figure 5.1: Component diagram of our prototype. Components implemented by us have been marked with blue color. Requests executed due to an event (like fetching vulnerable callables, or querying services by operator) are marked with a solid line. Requests executed periodically in constant intervals (like trace reporting, collection, and processing) are marked with a dashed line.

5.2.2 Build system plugin

The information about the application's dependencies is not explicitly available during the runtime of a Java application. The way how the dependencies are loaded by JVM depends on the way the application has been built and packaged – they can be included in the final artifact as JARs or standalone classes, they can be extracted to separate directories, and so on. In all these cases, there might be not enough information to reconstruct full Maven coordinates of the dependencies, which is necessary to obtain a list of vulnerable callables from repositories like FASTEN.

Build systems like Maven and Gradle are responsible for resolving and downloading application dependencies for build purposes. This means that they have access to their coordinates, which are necessary to download dependencies from the repositories. Since both of the build systems are extensible, it is possible to develop a plugin that will resolve the dependencies and save the list of them to the file.

Therefore, we have created a Gradle plugin, which resolves the coordinates of application compile dependencies during the build and stores them in a JSON file. This file can be bundled with the application as a part of its artifact, for instance inside the JAR archive or in the working directory of the Docker container. The agent can then load it and use the coordinates inside to obtain the list of vulnerable callables.

5.2.3 Instrumentation agent

Java applications are compiled into bytecode, which is then executed by the Java Virtual Machine (JVM). The language's ecosystem provides an instrumentation API, which allows for metaprogramming through the bytecode manipulation. What's more, the API allows for defining agents, which are programs capable of being launched side by side with JVM applications, the bytecode of which they can modify. Agents can be compiled and packaged just like normal applications, and are provided to the application during startup.

OpenTelemetry instrumentation agent framework OpenTelemetry[4], a project under the governorship of the Cloud Native Computing Foundation (CNCF), aims to standardize the way how observability data is collected from a variety of applications, written in different languages and technologies. One of the components of the OpenTelemetry project is a set of instrumentation agents, which automatically add recording of logs, metrics, and traces to existing applications. In the case of Java, these agents provide auto-instrumentation of applications, built using commonly used frameworks[5] like Spring, Grails, Hibernate and so on. Tasks performed by auto-instrumentation include modifying the behavior of downstream service calls so that trace identifier and context are propagated, creating spans whenever database transactions are executed, and so on.

[4] https://opentelemetry.io/

[5] https://github.com/open-telemetry/opentelemetry-java-instrumentation/blob/main/docs/supported-libraries.md

Agent extension OpenTelemetry agent provides a framework for creating custom instrumentation logic using the underlying ByteBuddy[6] library, which extends Java's default instrumentation API. Since ByteBuddy manipulates the bytecode, not only the application's logic but also its dependencies logic can be instrumented, since both have their classes loaded to the same classloader.

We used these facts to develop an extension, which fetches the information about the application's vulnerable callables from the coordinator component of our system (described in section 5.2.4) and applies instrumentation to them in the aspect-oriented[7] fashion.

Creating vulnerability trigger events After fetching the list of service's vulnerable callables from the coordinator, our agent extension instruments each of them by defining the pointcut[8] at the beginning of vulnerable callable. Whenever the vulnerable callable is executed, the code injected by the agent creates an event in the current span of the processed trace. It also includes all the metadata about the triggered callable and its vulnerability in the event's attributes. The OpenTelemetry base agent is then responsible for asynchronously sending the events to the trace aggregator in a separate thread.

Applying mitigations strategies In addition to the list of vulnerable callables of the application's dependencies, the agent also fetches the mitigation strategies, applied to each callable by the system operator. For demonstration purposes, we have implemented two vulnerability mitigation strategies: error and no-op. The first strategy is achieved by replacing the callable's logic by throwing a runtime exception. Unlike normal exceptions, runtime exceptions do not have to be declared in the method definition, which allows them to be thrown from arbitrary places in the code. The no-op strategy depends on the callable's type. For methods that return a value, their implementation is replaced by a single statement, returning a default value of the returned type. For the number types (*e.g.* `int`, `float`) the returned value is zero, for `boolean` it is false, and for `char` it is the null character. In the case of objects, the returned value is null. For procedures (methods returning `void`), the instrumentation performs a direct stack manipulation by loading a `null` pointer into the operand stack, followed by the `void` return handler, which effectively turns it into an empty method.

5.2.4 Instrumentation coordinator

Our agent is capable of extending distributed traces with vulnerability trigger events, and of applying mitigation strategies to them. However, to apply instrumentation to the vulnerable callables, it needs to know which ones should be instrumented, and if there should be any mitigation applied to avoid execution of vulnerable code. This could be achieved by allow-

[6] https://github.com/raphw/byte-buddy

[7] Aspect-oriented paradigm extends the existing code with custom behavior, declared separately from the extended code base

[8] Set of the control flow points that can be reached from multiple places in the program's code base, like execution of a method

ing the agent to communicate directly with the callable repositories like FASTEN and by fetching information about mitigations from the database.

This approach has several drawbacks. Since every application startup requires the agent to retrieve the list of callables again, restarting multiple instances at the same time would lead to swarming the callable repository with requests. This could trigger anti-DoS[9] safety mechanisms of repositories, rendering them unusable. Any unavailability of repositories poses a problem since restarted services would not be able to obtain the callables. They would have to either wait for the repositories to become available, or to start without applying any instrumentation. Using a local cache in the form of a file, created by the agent to be read when repositories are not available, is not feasible for implementation in Kubernetes cluster environments due to the ephemeral nature of the containers.

Registering services and its dependencies To tackle that issue, we decided to introduce a dedicated component called the instrumentation coordinator. The coordinator serves as the communication endpoint for the agents. During its startup, the agent sends a request to the coordinator, with the payload consisting of the name of the instrumented service, and the list of its dependencies from the file generated by the build plugin (as described in section 5.2.2). The coordinator registers the service in its database, along with the list of its dependencies. Then, it starts querying the callable repository to obtain information about vulnerabilities in the service's dependencies, and the vulnerable callables in them. Queries for each dependency are sent in parallel to ensure performance and throttled with the use of semaphore to avoid overloading the repositories. The vulnerabilities and vulnerable callables are persisted in the coordinator's database and bound to their dependencies. Finally, the coordinator aggregates all the callables and returns them to the agent, along with the mitigations configured by the system operator in the dashboard (described in section 5.2.6).

Coordinator as a local cache Persisting information about vulnerabilities and callables in its database has several advantages. Instead of querying remote repositories whenever a service is restarted and requests the list of callables and mitigations, the data can be served directly from the local database. This allows the coordinator to avoid querying repositories unnecessarily and accelerates the startup time of the agents. Moreover, whenever repositories are unavailable, the coordinator is still able to provide the data.

Mocking appearance of new vulnerabilities In the real-world scenario, the application dependencies should be continuously monitored for the appearance of new vulnerabilities. This could be done by a separate component that would periodically query callable repositories and add new vulnerabilities with their callables to the database whenever a new one is detected. Then, the operator should be informed about the necessity of restarting instrumented services, so that the instrumentation agent can fetch the new callables from the coordinator.

[9]The goal of denial-of-service (DoS) attack is to oversaturate resources of the server, usually by flooding it with requests

For the experiment purposes, we have extended the coordinator module with mocking capability. It allows us to create a dependency with the severity of choice, attach it to arbitrary Maven coordinates, and provide information about the vulnerable callable. The callable does not have to be part of the dependency – the agent will instrument an arbitrary method, as long as the application has a dependency with the same coordinates as provided. This simulates discovering new vulnerabilities that would be normally done by the aforementioned query component.

5.2.5 Collecting and processing traces

As for the time of writing, the trace aggregator used in our experimental setup – Grafana Tempo – does not provide the capability to perform complex aggregate queries on the stored traces. To enable that, we decided to collect the traces, process them, and store the results in the relational database.

The process of turning traces into trace graphs consists of two steps, which are performed by two separate components of the system – trace collector and processor. The former one is responsible for determining which traces should be processed, while the latter one does the actual processing and trace graph building. This split was done due to the nature of Tempo, which exposes two endpoints for fetching traces:

- search – retrieves basic metadata (like identifier) of traces from the provided time range, based on the span attributes. Since the service name is a span attribute, it can return the traces, which contain spans from the given service.
- query – retrieves all spans and metadata of a single trace

During our experiments, we observed a rapid increase in memory usage in the querier and ingester components in Tempo whenever too many requests were sent to the query endpoint in a short period, which resulted in crashes of these components. In the normal use case, Tempo's search and query endpoints are used respectively to search for the traces and to visualize a single one of them in the Grafana dashboard. It could be that the logic behind the query endpoint was not optimized for being called repeatedly. Splitting search and query operations to be done by separate services allowed us to granularly control the amount of requests, sent to both endpoints. By modifying the intervals between collection and processing, as well as the batch sizes, we managed to achieve equilibrium in our experimental setup, where traces are collected and processed continuously without crashing Tempo components.

Collecting traces to process In the first step, the trace collector determines traces that should be processed later. It monitors the database for service registrations and deregistrations that are done by the coordinator module. For each monitored service, the collector spawns a goroutine[10] which periodically calls Tempo's search endpoint. The search operation is issued every 5 seconds for a fixed interval after a fixed delay. In our experimental

[10]Goroutines in Go programming language are lightweight threads, managed by Go runtime. Unlike the operating system threads, goroutines have a flexible stack size and can be spawned in much larger quantities than the system threads

setup, the delay is set to 5 minutes and the size of the search interval is 3 minutes. This means that for each run of the collector goroutine, it fetches traces that were started between 5 and 8 minutes before the run. It gives the services and service mesh components enough time to report all the spans to the aggregator. After fetching trace identifiers from Tempo, the collector persists them in the database as unprocessed.

The provided search interval length combined with the relatively frequent issuing of the search endpoint means that the subsequent intervals are overlapping. As a result, there is a large chance that the next run of the collector will return the same trace identifiers that were persisted in the database during the previous run. However, it does not lead to multiple instances of the same trace being collected into the database. The database uniquely indexes trace identifiers and ignores duplicates during batch inserts.

Processing traces In the second step, the trace processor queries the database periodically to fetch batches of the oldest unprocessed traces. For each trace identifier in the batch, it executes the query endpoint of Tempo to fetch all of its spans and events. Each trace is processed into the trace graph, which is then persisted into the database, along with vulnerability trigger events that occurred in one of its nodes. We will now describe the process of building the raw graph from the trace, followed by the reduction that has to be done to obtain a valid trace graph.

Building initial trace graph First, the graph is built directly from the span hierarchy, obtained from Tempo. Each span is represented as a node, with its events as the node payload. A directed edge from the child to the parent span node is created whenever the child span has a parent span. However, as seen in figure 5.2, each service can report more than one span. This leads to the situation where the obtained graph is full of chains of consecutive nodes, marked as the same service. Moreover, whenever a downstream request from s_1 to the instrumented service s_2 is made, it leads to the creation of three sibling spans.[11] Two of them were reported by the service mesh and represent outbound and inbound connections of s_1 and s_2 respectively. The third one was reported by the called service s_2 and its children represent the remaining hierarchy of spans. If s_2 was not instrumented, the spans from the service mesh would be essential to determine the path of downstream calls. However, since s_2 is instrumented and reports its own spans, the remaining two coming from the service mesh do not provide additional information. Leaving them would be also problematic from the perspective of deriving impact graph metrics since during aggregation they would be seen as additional requests, made between the services.

Reducing trace graph To turn the raw graph into a valid trace graph that shows the communication between services within the mesh, we have to reduce it. Reduction consists of two steps:

- merging – removes consecutive node chains, caused by instrumented services reporting multiple spans

[11] two spans are siblings when they have the same parent span

- pruning – removes sibling nodes without children

First, the raw trace graph has its consecutive nodes merged. The merged graph is then passed to pruning, resulting in a reduced trace graph which can be persisted in the database.

The process of merging involves a loop that executes until there are no nodes to merge in the graph. First, the algorithm performs a breadth-first search (BFS) on the raw graph. Whenever a successor node is found such that it has the same service name as its predecessor, the predecessor is then merged into that node. Merging means redirecting all incoming edges of the predecessor to the successor and redirecting all the predecessor's outgoing edges to come out of the successor. If the predecessor has any vulnerability trigger events, they are transferred to the successor. Finally, the edge connecting predecessor and successor is removed and the successor node is removed as well. The pseudocode of the merge procedure is presented in algorithm 1.

Algorithm 1 Merge consecutive node chains in the raw trace graph

```
1:  Function merge(rawGraph)
2:  Clone the rawGraph to create a mergedGraph
3:  repeat
4:      Perform a BFS starting from the root node to identify nodes to merge:
5:      for each node in BFS do
6:          Check if any node predecessor has the same service name
7:          if found then
8:              Mark the current node and the predecessor for merging and exit BFS
9:          end if
10:     end for
11:     if no nodes were identified for merging then
12:         break
13:     else if predecessor node was identified for merging then
14:         Redirect all incoming edges from the predecessor to its successor
15:         Redirect all outgoing edges from the predecessor to its successor
16:         if the predecessor node has any events then
17:             Transfer these to the successor node
18:         end if
19:         Remove the edge between predecessor and successor
20:         Remove the predecessor from the graph
21:     end if
22: until no more nodes can be merged
23: return the merged graph
```

Similarly, as in the case of node merging, pruning also involves looping until there are no more nodes to prune. The algorithm performs a BFS and checks the successors of each node. If a node has more than one sibling successor with the same service name, each sibling is checked for how many successors it has. Upon identifying a sibling without successors, it is marked for removal. The node is then detached from its predecessor, has

its events transferred to the predecessor, and is removed from the graph. The pseudocode of the merge procedure is presented in algorithm 2.

Algorithm 2 Prune Leaves from a Graph

```
1: Function prune(mergedGraph)
2: Clone the mergedGraph to create prunedGraph
3: repeat
4:    Perform a BFS from the root node to identify leaf nodes for pruning:
5:    for each node visited do
6:       Map each successor's service name to its count and track successors' successor
         counts
7:       for each successor do
8:          if successor's service name appears more than once and it has no successors
            then
9:             Mark this successor for pruning
10:            Find a sibling node with the same service name
11:            Return true {Indicates a node was found for pruning}
12:         end if
13:      end for
14:   end for
15:   if no pruning candidate is found then
16:      break
17:   end if
18:   Disconnect the pruned node from its predecessor
19:   if the pruned node has events then
20:      Append events to its sibling's event list and update the sibling node
21:   end if
22:   Remove the pruned node from the graph
23: until no more nodes can be pruned
24: return the pruned graph
```

5.2.6 Dashboard

The components described so far are the backend of the prototype. They deal with technical aspects of collecting information about triggers of vulnerable code, instrumenting applications to avoid vulnerability execution, and processing telemetry data for further visualization. Once the operator deploys all components and attaches agents to the workloads, the vulnerability trigger collection and trace graph processing will work. However, to allow for impact assessment and applying mitigation strategies, it is necessary to provide the operator with a dashboard.

The implemented prototype dashboard consists of four views, described below.

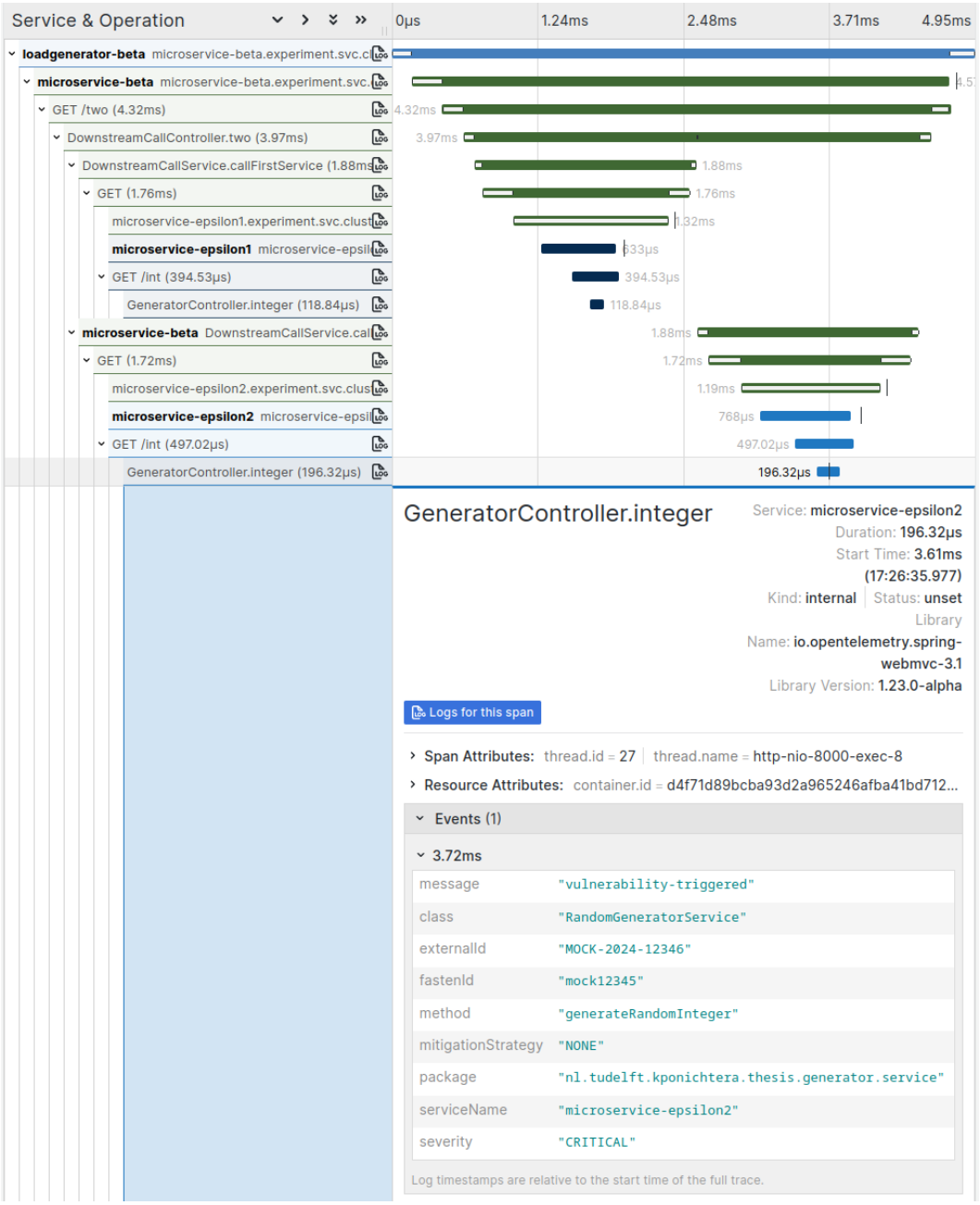


Figure 5.2: Visualization of a distributed trace from Tempo. The last span, reported by the *epsilon*$_2$ service, contains an event with a vulnerability trigger event.

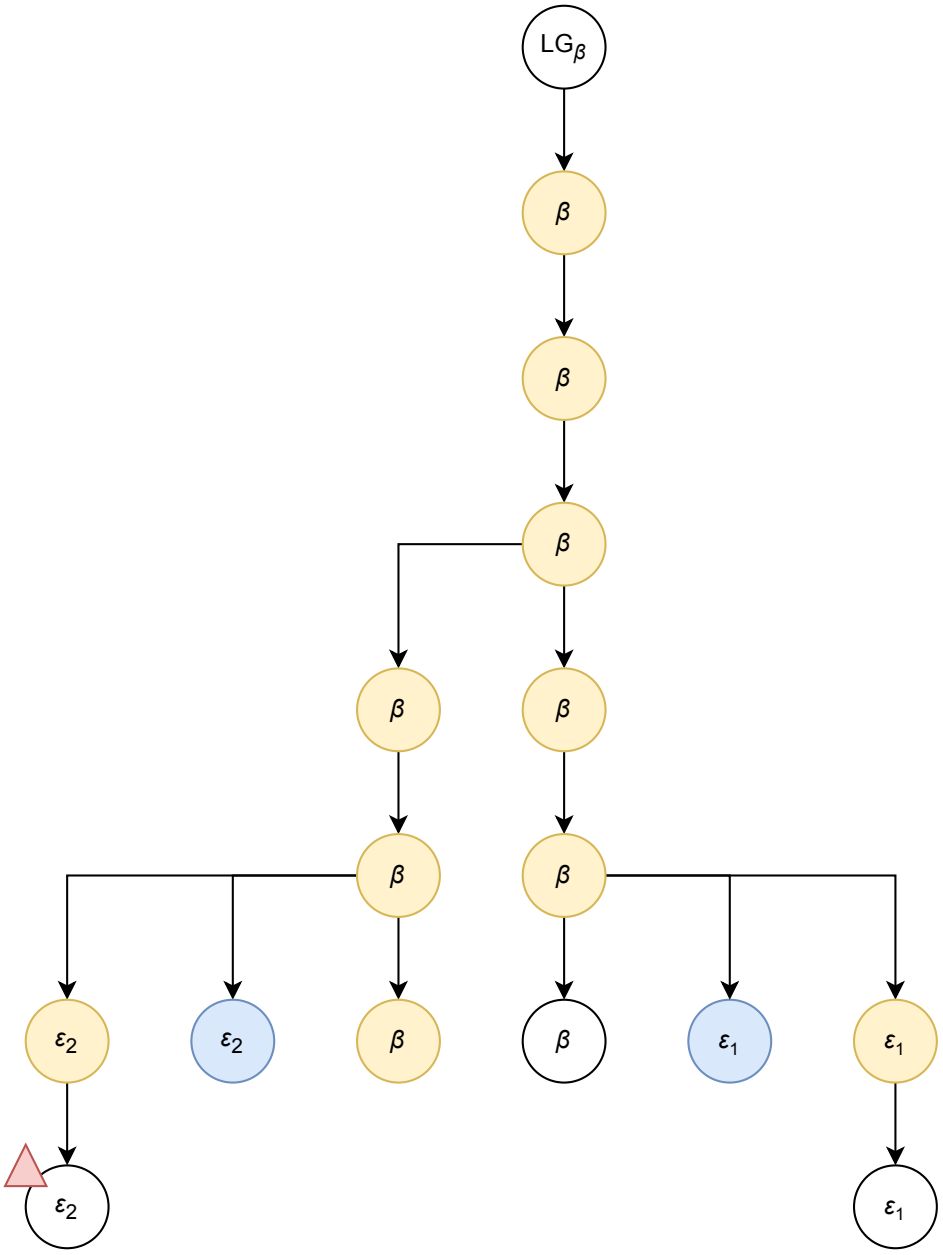


Figure 5.3: Trace graph before reduction, built from the trace depicted in figure 5.2. The node with the vulnerability trigger event has been marked with a red triangle. Yellow nodes are candidates for merging, while the blue ones are the candidates for pruning.

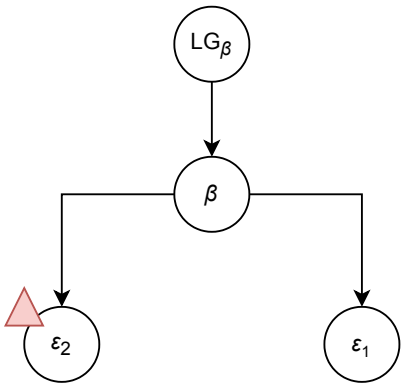


Figure 5.4: Trace graph from figure 5.3 after reduction.

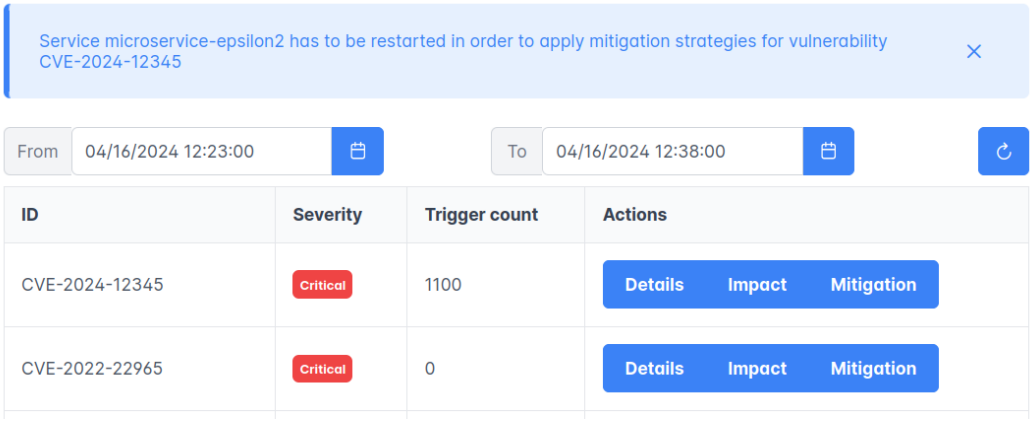


Figure 5.5: Main dashboard page, showing vulnerabilities and their trigger count in the context of the whole service mesh. The notification informs the operator about the necessity to restart the service to apply the newly chosen mitigation strategy.

Main dashboard The main page of the dashboard shows a list of vulnerabilities, present in all registered services. It shows their identifiers, severity, and the number of triggers. Counting the triggers is done by aggregating the vulnerability execution events from the trace graphs in the given time interval. The dashboard also shows notifications about events that require the operator's manual intervention, like restarting the application to start monitoring newly discovered callables or applying a mitigation strategy. Each vulnerability in the list has dedicated buttons, allowing the operator to navigate to the remaining views of the dashboard.

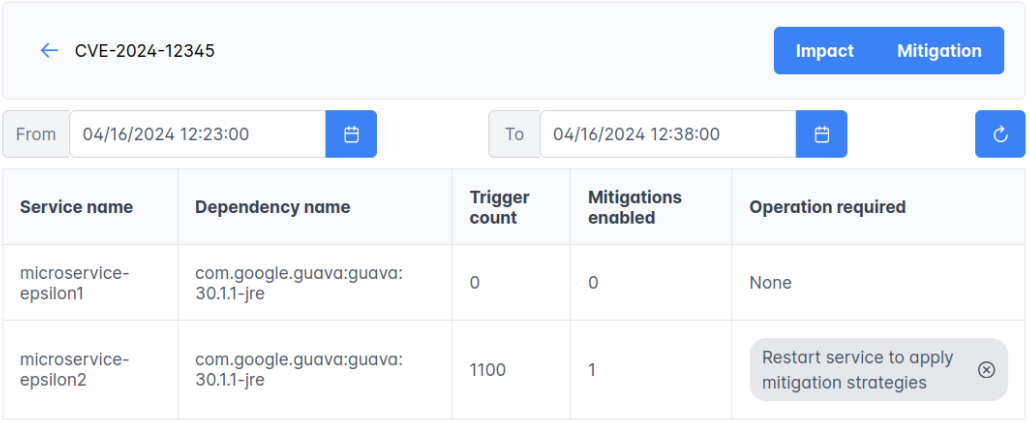


Figure 5.6: Details screen, showing the list of services affected by the vulnerability.

Vulnerability details Clicking on the *Details* button in the vulnerability list navigates to the page with the details about the behavior of the chosen vulnerability in the system. It shows services, affected by the particular vulnerability, as well as the Maven coordinates of the dependency that causes the problem. The vulnerability triggers are aggregated for each service individually in the provided time interval. The list also shows the number of mitigations, applied to vulnerable callables within a particular service, and the operations that have to be performed manually by the operator. The dashboard user is also presented with two buttons, allowing them to navigate to the impact assessment and mitigation pages.

Mitigations Clicking the *Mitigation* button brings the user to the screen, used to apply mitigation strategies to vulnerable callables within affected services. The page shows a tabbed view, where each tab represents a service, affected by the given vulnerability. Choosing a tab reveals the list of vulnerable callables within that service, complete with information about Maven coordinates of responsible dependency, the callable's signature, the number of mitigated and non-mitigated triggers, as well as the mitigation strategy for each callable. Each vulnerable callable has no mitigation enabled (option *None*) by default, meaning that its behavior is not changed, and the applied instrumentation is responsible only for tracking the triggers. The operator can choose an error or no-op mitigation strategy for each callable individually. Changing the mitigation strategy creates a pending notification for the operator, reminding them to restart the service so that the agent can fetch the newly-applied mitigation for the callable.

Impact assessment Choosing the *Impact* button navigates to the impact assessment page. It shows the impact graph, generated for a specific vulnerability from the graphs of traces that had their first spans reported to the aggregator in the given time interval. Nodes represent individual services in the service mesh. The color is used to distinguish services directly affected by the vulnerability from the ones that do not have its faulty code – vulner-

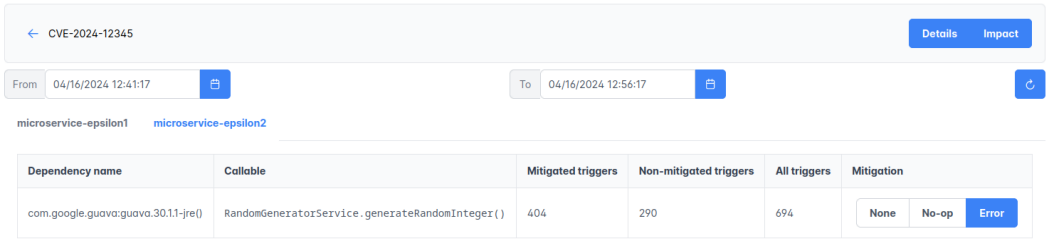


Figure 5.7: Mitigation screen, showing the list of vulnerable callables within the service, the amount of mitigated and non-mitigated triggers, and the chosen mitigation strategy.

able services are red, while the rest are blue. Edges show the direction of the downstream calls made between services. Whenever there is a trace from the given time interval that ends up triggering vulnerable code, all the affected edges of its trace graph are represented as red edges in the impact graph. This allows the operator to determine what other services are transitively calling the vulnerable code by making downstream calls to the vulnerable service. As a result, the operator can derive the following information from the impact graph:

1. impact of the vulnerability – which services in the mesh are potentially affected by the execution of the vulnerable code in the downstream service, for example by having the affected service return an undesirable response
2. impact of the vulnerability mitigation – which services could be affected if the vulnerable code were to be mitigated by the agent, for example by propagating the error

The graph is created by fetching all the trace graph nodes in which at least one node is vulnerable – i.e. has a vulnerability trigger event registered. Then we connect these nodes by querying the database for the edges. We distinguish two types of edges – the vulnerable and non-vulnerable ones. Vulnerable edges form the path within the trace graph that leads to the vulnerable node. They are obtained by querying for the vulnerable nodes of trace graphs and traversing their incoming edges. The remaining edges of the trace graphs from the provided time interval are treated as non-vulnerable. The obtained trace graph edges are then categorized according to their source and target nodes. For each pair of a source and target node of the impact graph that has at least one edge collected, we create an edge in the impact graph that joins these two nodes. We also keep track of the amount of vulnerable and non-vulnerable edges in the newly created edge, which allows us to define the edge metrics.

Impact graph edge metrics Since each impact graph edge has information about the number of vulnerable and non-vulnerable trace graph edges that were used to build it, we can derive quantitative metrics from it. In the most basic setting, the metric shows how many inbound requests to the service mesh ended up invoking particular downstream calls, and how many of these calls resulted in executing vulnerable logic. This can be represented by two numbers: the amount of vulnerable calls and the amount of all calls on the particular edge. For the edges without any vulnerable calls, the latter number is sufficient. The

operator can use that information to establish how much traffic within the service mesh is affected. Looking at the outgoing edges of the nodes without any incoming edges (the "roots") allows for determining how much of the ingress traffic entering the service mesh is affected. Meanwhile, inspecting the edges further down the graph allows for finding out how much of the communication between individual services is affected. In the case where the numbers are too large to provide meaningful information at a glance, the operator can also switch the metric to percentage. The impact graph in figure 5.8 shows an example where edges marked in red show the ratio of vulnerable traces out of all the traces that involve communication between the involved services. Meanwhile, the gray edges do not cover any vulnerable traces and display only the total number of covering traces.

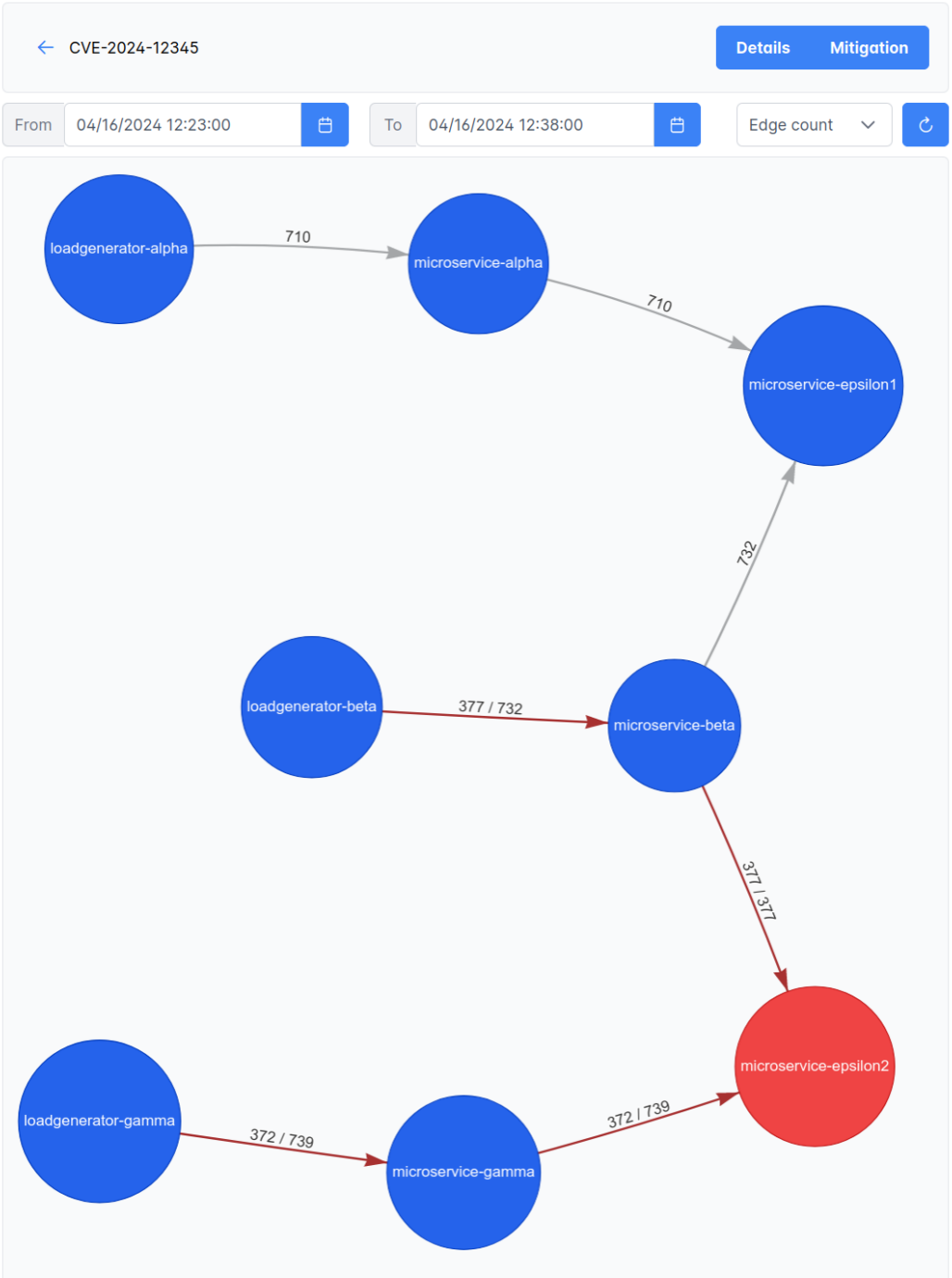


Figure 5.8: Impact assessment screen with the impact graph. The affected service is marked in red. The chosen edge metric is the vulnerable trace count, showing the number of affected traces out of all that cover the given edge in the provided time interval.

Chapter 6

Prototype evaluation

Our solution revolves around three pillars: improving the awareness of the vulnerabilities triggered during the system's runtime, providing a way to perform impact analysis of the vulnerability and its mitigation, and allowing for fast, short-term mitigation. In this chapter, we will evaluate our developed solution from the perspective of tackling these three issues by defining use case scenarios and checking if our prototype can solve the challenges as foreseen. We will also validate the feasibility of applying it in the real world, by measuring resource usage and checking the additional latency, introduced by instrumenting the applications.

6.1 Methodology

To evaluate our prototype, we created a simple microservice system, consisting of two types of services. Each service is a Java application, developed with the use of Spring Boot and Spring Framework.[1] The first type is the downstream caller service, which exposes two REST API endpoints: `one` and `two`. Each service has a load generator, which calls both endpoints every second to simulate the load. Sending a request to the endpoint `one` and `two` makes the service send a request to one or two downstream services respectively. The second type is a generator service, exposing two endpoints: `int` and `uuid`. Once one of them is triggered, it can generate a random integer number or a random UUID[2], and then log their values to their standard output. Generation can be enabled or disabled in the service's configuration. Disabling generation makes the service still handle the request but does not make it call the generation logic.

For three deployed downstream caller services – *alpha*, *beta* and *gamma* – triggering endpoints `one` and `two` makes the service send a request to one or two generator services $epsilon_1$ and $epsilon_2$, which have their generation logic disabled and enabled respectively. The configuration of the downstream call targets will differ, depending on the scenario.

In each experiment run, the method responsible for generating integer numbers is registered in our coordinator module to be part of the Google Guava library, which is a depen-

[1] https://spring.io/projects/spring-framework

[2] universally unique identifier

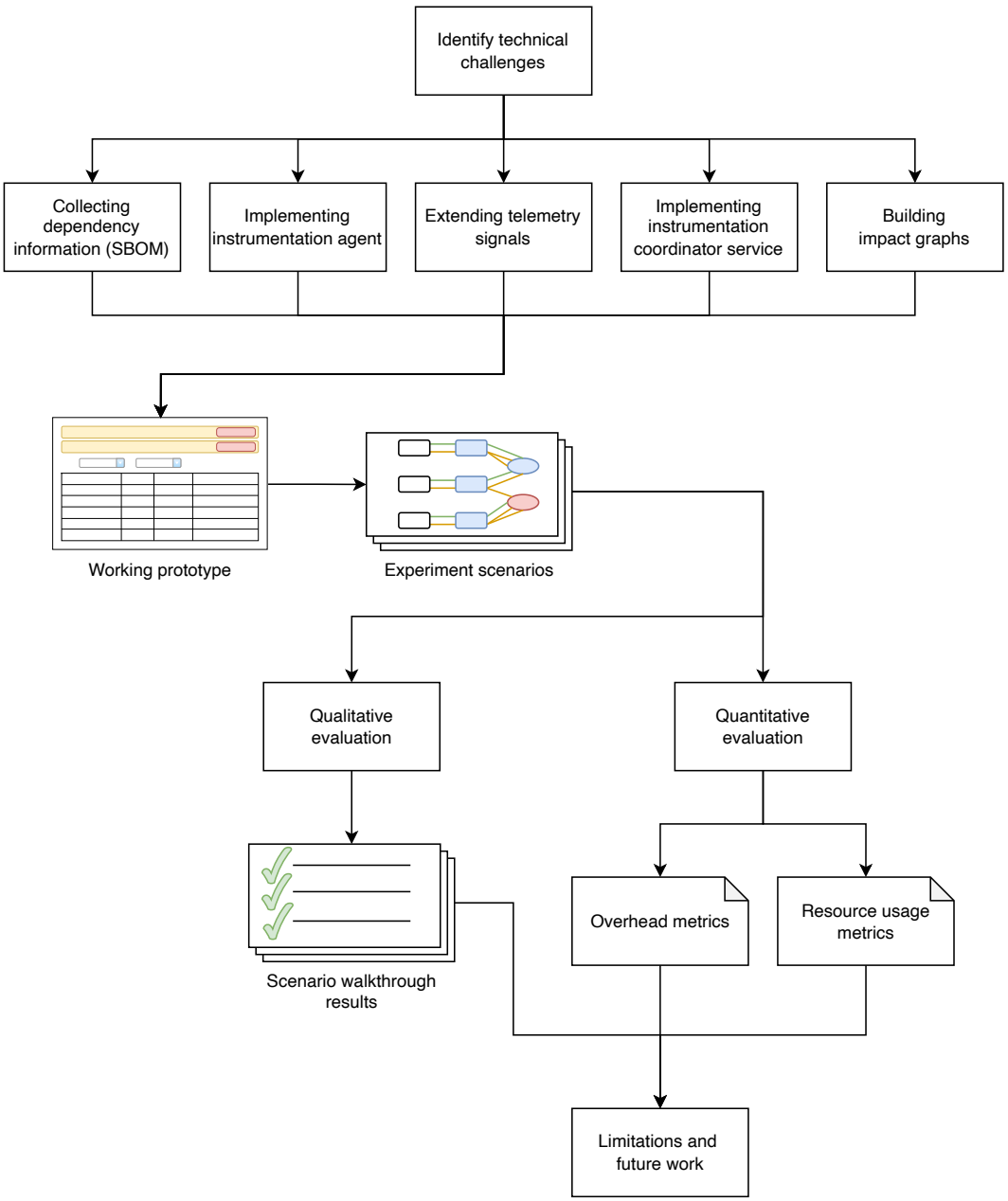


Figure 6.1: Prototype development and evaluation methodology

dency of the generator services. Then, the method will be marked as vulnerable under a mock critical vulnerability *CVE-2024-12345*. In the real-world case, registering new vulnerabilities in the coordinator module could happen by periodically scanning vulnerable callable storage services like FASTEN for new vulnerabilities in dependencies of registered services. In each scenario, the operator has access to the prototype dashboard, the system's observability stack (log, metric, and trace aggregators), the service mesh management system and the application deploy management system. As it was in the case of the scenario, presented to interview participants, the operator is not able to modify the code and application artifacts.

All of the services have the instrumentation agent attached to them and have registered themselves with the coordinator component of our prototype. As a result, the trace collector is configured to collect traces that go through them, while the trace processor builds and persists trace graphs in the database.

6.2 Qualitative evaluation

During the evaluation of our vision, we discussed with the interview participants how the operators get informed about the vulnerability's appearance, what they do with that information, and how they can mitigate the problem. According to interviewees, the operators learn about the appearance of new vulnerabilities from the results of continuous integration pipelines or periodic scans. The impact of the vulnerability is generally assessed as the number of services with the vulnerable version of a dependency. Whenever a new vulnerability appears, the operator forwards that information to other teams. While developers work on the long-term mitigation solution, the operator might be tasked with modifying the configuration of the application and its deployment to avoid executing vulnerable code in the short term. We have envisioned that the developed solution would help improve awareness about vulnerability behavior in the deployed microservice systems, evaluate the vulnerability's impact on the system, and allow for their granular mitigation without having to rebuild or shut down system components.

Ideally, the qualitative evaluation of the prototype would be done by conducting interviews with industry practitioners, similar to how it was done with evaluating the vision. However, since the prototype is the implementation of the vision, we decided to simplify its evaluation by defining three scenarios. By following these scenarios, we will validate if the prototype can tackle the challenges of improving the operator's awareness, impact assessment, and actionability in terms of vulnerability handling. Through the interviews, we have determined how these improvements could be validated empirically from the perspective of an operator. Improving awareness can be achieved by providing a way for the operator to know about the appearance of the vulnerability in one of the system's components, without running manual or periodic scans and facilitating early response to the issue. To improve impact assessment, it is necessary to deliver a possibility to monitor the runtime behavior of vulnerability – namely, if the vulnerable code is being executed during system operation and what other services triggered it by doing downstream calls. Actionability can be enhanced by allowing the operator to mitigate the vulnerability, based on the insights obtained from

impact assessment and external sources (*e.g.* by discussing the matter with developers or the business department), but without modifying the application's code or artifact.

We have created three scenarios to validate if our prototype can tackle the aforementioned challenges:

- Scenario 1 – vulnerability is discovered in the service's dependency, but it's not triggered during normal execution of the system
- Scenario 2 – vulnerability is discovered in the service's dependency, is triggered during system execution, and can be mitigated with the built-in strategies
- Scenario 3 – vulnerability is discovered in the service's dependency, is triggered during system execution, but the built-in strategies are not suitable to mitigate it

We will describe them in detail, discuss how they would be handled by the operator without access to our envisioned system, and how the typical walkthrough would be done with the use of the prototype. Finally, we will discuss the differences and improvements the prototype brings over state of the art.

6.2.1 Scenario 1

Setup In this scenario, the vulnerability appears in one of the service's dependencies but is not triggered during normal execution of the system. Its purpose is to show the prototype's usability in cases where vulnerability is not actively exploited. The downstream call targets have been configured as follows:

- *alpha* – triggering endpoint `one` sends a downstream request to `uuid` endpoint of service *epsilon*$_1$. Triggering endpoint `two` sends two requests to `uuid` and `int` endpoints of the same generator service.
- *beta* – triggering endpoint `one` sends a downstream request to `int` endpoint of service *epsilon*$_1$. Triggering endpoint `two` sends a request to `int` endpoint of service *epsilon*$_1$ and another one to `uuid` of *epsilon*$_2$.
- *gamma* – triggering endpoint `one` sends a downstream request to `uuid` of service *epsilon*$_2$. Triggering endpoint `two` sends two requests to the `uuid` endpoint of the same generator service.

Although service *epsilon*$_2$ exposes its vulnerable `int` endpoint, it's not being executed by any downstream call service. The architecture diagram of the experimental setup that shows the downstream calls, made by the services, is depicted in figure 6.2.

State of the art In the provided scenario, the operator would usually get informed about the vulnerability's appearance during the next execution of the scanning pipeline. This would be done during the next development iteration or periodic vulnerability scan, whichever gets executed first. The impact assessment would be done by checking which services have the vulnerable version of the dependency. However, further assessment would have to be done by the developer, by looking into the code or probing the application in an attempt to trigger the vulnerability.

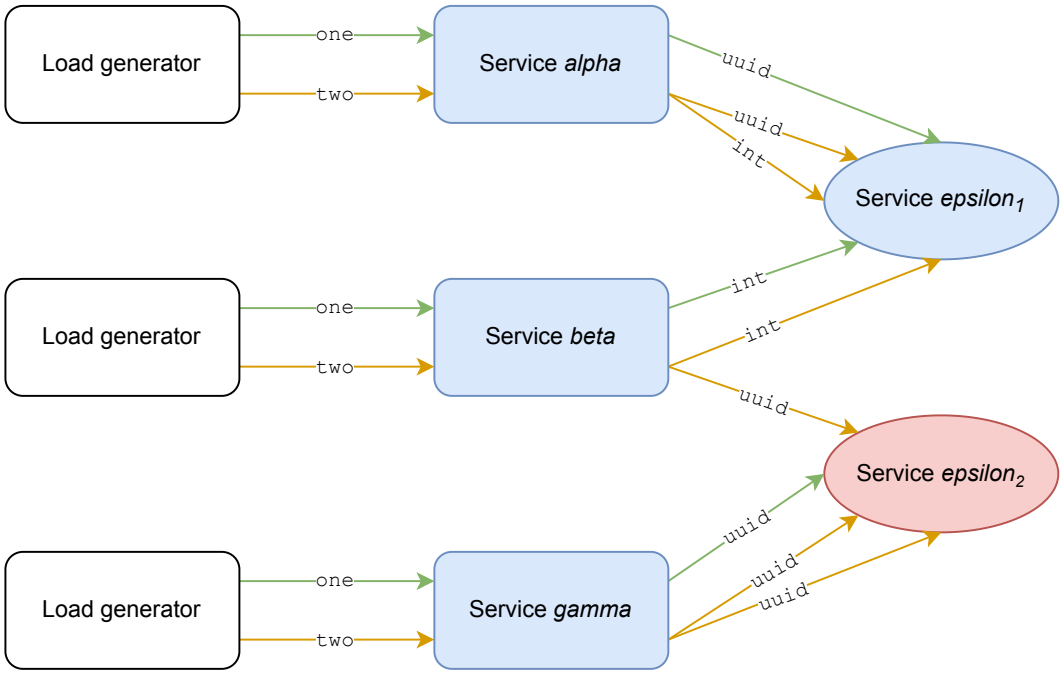


Figure 6.2: Architecture diagram of the experimental setup for the first scenario. Service *epsilon*$_2$ with reachable vulnerable code has been marked as red. However, in this scenario, its vulnerable endpoint `int` is not being executed.

Prototype walkthrough With the envisioned system in place, a new vulnerability gets registered in the coordinator and the dashboard shows notifications to the operator, as depicted in figure 6.3. Entering the vulnerability details screen reveals which services are vulnerable (figure 6.4). Notifications instruct the operator to restart service instances so that agents can fetch newly discovered vulnerable callables and instrument them to provide information whenever vulnerabilities are triggered. The operator can dismiss the notifications by clicking the close button or by restarting at least one service instance. Once services are restarted and new vulnerable callables get instrumented, the operator notices in the details screen (figure 6.5) that even though vulnerable callables are present in the services, they are not being called in any of them during the system's execution.

Discussion and comparison As a result, using our system helped to improve awareness, by providing the operator with information about new vulnerabilities being present in the system immediately after they were discovered and reported to the coordinator. The vulnerability was therefore discovered proactively, as a result of its disclosure, and not reactively by running periodic security scans. It was also independent of the development stage of the application and the frequency of running security scans during pipeline execution. We also managed to improve the impact assessment – in addition to informing the operator about vulnerable services and their dependency versions, we provided information that the newly

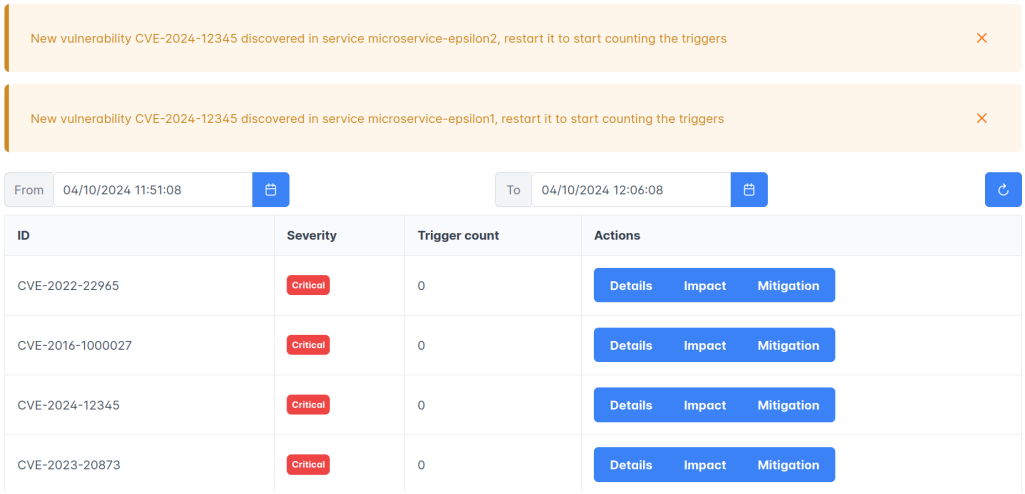


Figure 6.3: Screenshot of a dashboard, showing notifications about a newly discovered vulnerability

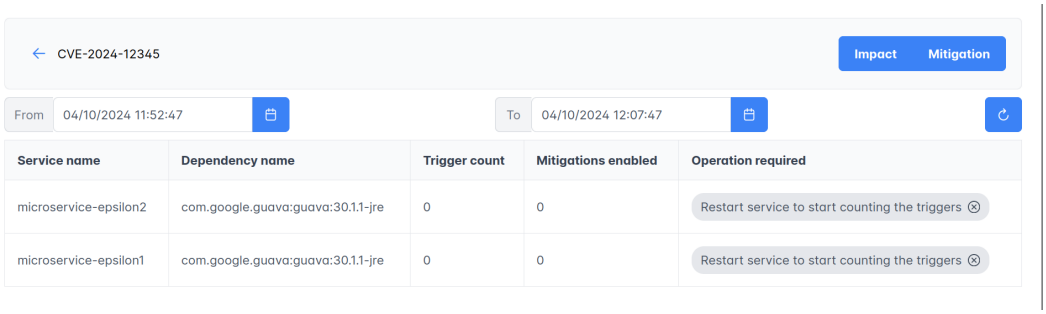


Figure 6.4: Screenshot of a vulnerability details screen, showing services in which the vulnerability has been discovered. The operator is also informed which services have to be restarted to start counting the triggers.

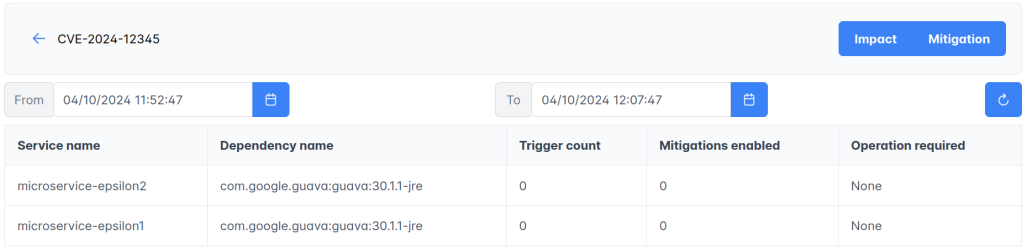


Figure 6.5: Screenshot of a vulnerability details screen after restarting services to instrument vulnerable callables, showing that the newly discovered vulnerability is not being triggered.

discovered vulnerability is not being triggered during the system's operation.

6.2.2 Scenario 2

Setup In this scenario, the vulnerability appears in one of the service's dependencies and is triggered during normal execution of the system. Its purpose is to show the prototype's usability in cases where vulnerability is actively exploited and it is possible to mitigate it granularly. The downstream call targets have been configured as follows:

- *alpha* – triggering endpoint `one` sends a downstream request to `uuid` endpoint of service *epsilon*$_1$. Triggering endpoint `two` sends two requests to `uuid` and `int` endpoints of the same generator service.
- *beta* – triggering endpoint `one` sends a downstream request to `int` endpoint of service *epsilon*$_1$. Triggering endpoint `two` sends a request to `int` endpoint of service *epsilon*$_1$ and another one to `int` of *epsilon*$_2$.
- *gamma* – triggering endpoint `one` sends a downstream request to `uuid` of service *epsilon*$_2$. Triggering endpoint `two` sends two requests to the `uuid` and `int` endpoint of the same generator service.

Here, the service *epsilon*$_2$ exposes its vulnerable `int` endpoint, which ends up being triggered by *beta* and *gamma* services. The architecture diagram of the experimental setup that shows the downstream calls, made by the services, is depicted in figure 6.6.

State of the art Similarly to scenario 1, the operator would normally get informed about the vulnerability's appearance through the execution of a pipeline or a periodic security scan. The impact assessment is limited to checking which services are vulnerable. To determine if the vulnerability is exploitable, it is necessary to assign such a task to the developer or a security specialist. If they determine which part of the service is vulnerable, the operator might be tasked with disabling part of the system's functionality through configuration or asked to take the affected services down. However, there is no possibility of checking if the vulnerability is being triggered in the environment. Therefore, unless the exploit is obvious and disturbs the system's operation, it is impossible to determine if the mitigation was just a preventive measure or if malicious actors used it.

Prototype walthrough With the envisioned system in place, the operator is informed about the new vulnerability's appearance just like in scenario 1. After restarting services, they notice that there is a rise in the amount of vulnerability triggers. This can be seen both on the dashboard (figure 6.7) and the screen with vulnerability details (figure 6.8), which also shows that vulnerability *CVE-2024-12345* is being executed in the *epsilon*$_2$ service. Switching to the impact assessment screen reveals the impact graph with the vulnerable service marked in red (figure 6.9). The impact graph analysis allows the operator to deduce that:

- service *alpha* is not affected by the vulnerability, since no path in the graph leads from it to the vulnerable service.

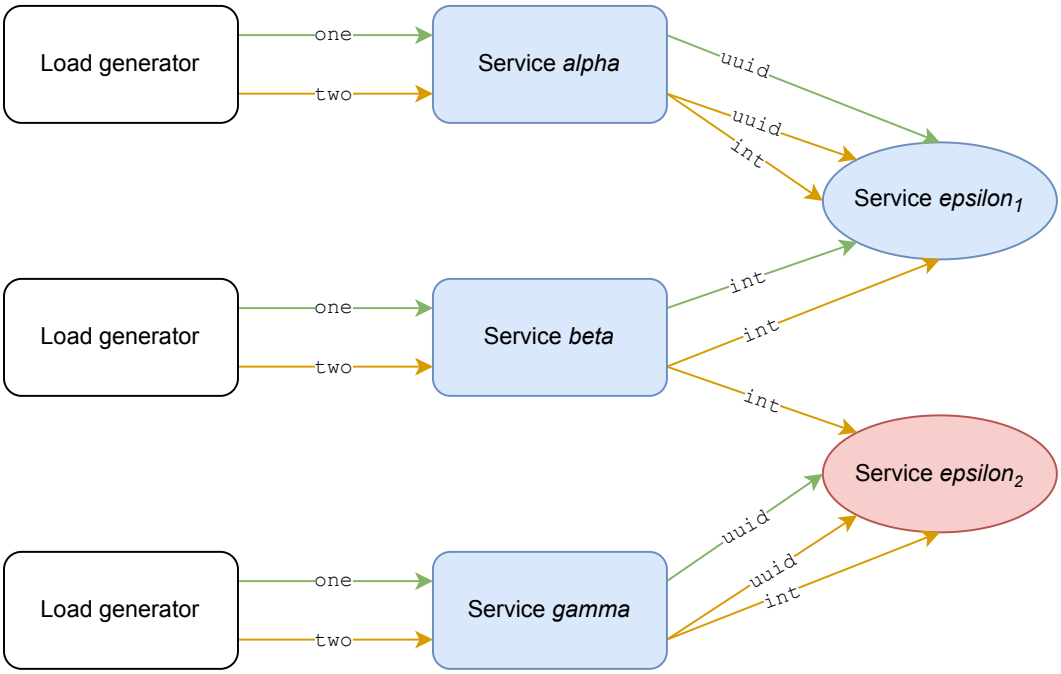


Figure 6.6: Architecture diagram of the experimental setup for the second scenario. Service *epsilon*$_2$ with reachable vulnerable code has been marked as red. Its vulnerable endpoint `int` is executed by *beta* and *gamma* services.

- service *beta* is affected by the vulnerability – about 50% of traces from its load generator, and 100% of calls between it and *epsilon*$_2$ end up calling the vulnerable code. This makes sense since the load balancer of *beta* sends two types of requests – the first one results in the downstream service requesting integer generation from *epsilon*$_1$, while the second one makes *beta* request integer generation from both *epsilon*$_1$ and *epsilon*$_2$. In the graph, we see that all traces with spans between *beta* and *epsilon*$_2$ call vulnerable code, since the only type of request that *beta* sends there is one for integer generation.
- service *gamma* is also affected by the vulnerability – approximately half of the traces from its load generator and half the traces between it and *epsilon*$_2$ result in calling vulnerable code. Calls to the endpoint `one` of the service execute endpoint `uuid` of *epsilon*$_2$, which is not vulnerable. Although a third of requests from *gamma* is made to the vulnerable `int` endpoint, they are made as a result of calling endpoint `two` of the service. This makes requests to `uuid` a part of the same trace that also calls the `int` endpoint, which is vulnerable.

The operator can then use the trace aggregation system to check which endpoints of *epsilon*$_2$ service are being executed. Upon deciding to mitigate the vulnerability in a granular way, the operator navigates to the mitigation screen (figure 6.10), which shows them a list of vulnerable callables with detailed information about the triggers of callables in the context of all affected services. The operator chooses the error mitigation strategy, which replaces

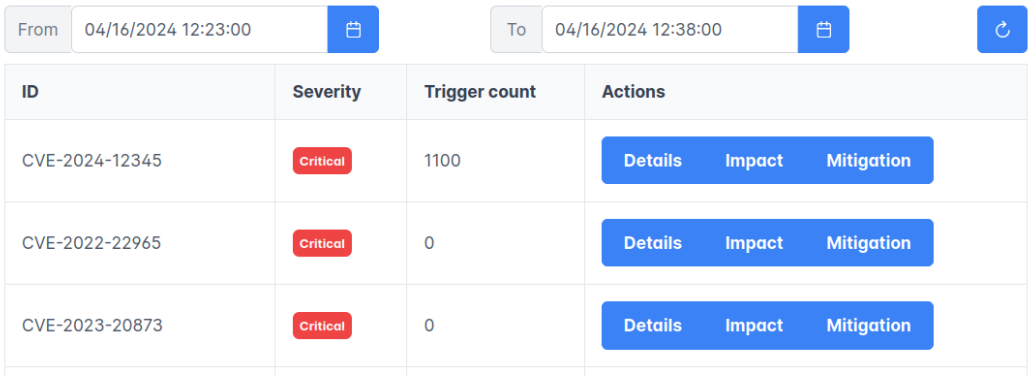


Figure 6.7: Screenshot of a dashboard, showing a rise of *CVE-2024-12345* vulnerability triggers.

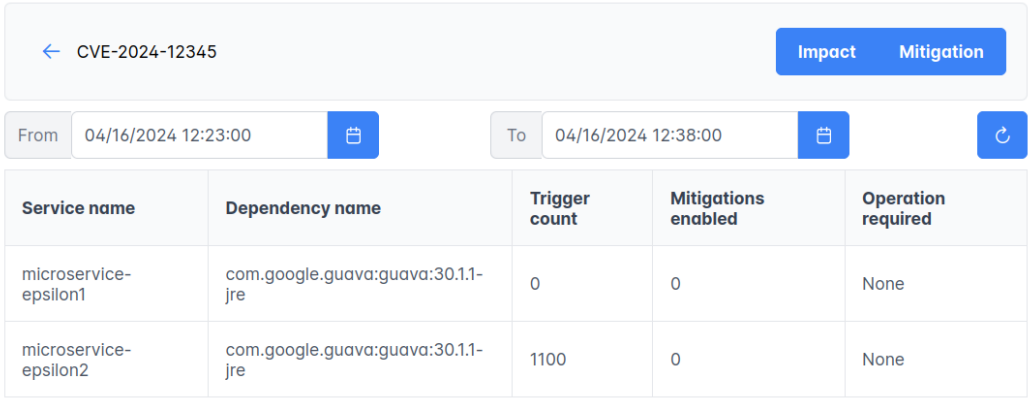


Figure 6.8: Screenshot of details screen, showing a rise of *CVE-2024-12345* vulnerability triggers in the service $epsilon_2$.

the integer generation logic by raising an exception whenever the method is called. They are immediately informed about the necessity to restart the service instances so that their agent can fetch information about the new mitigation strategy for the callable and perform instrumentation during startup. In case the operator misses the notification, it will also be displayed in the dashboard and vulnerability details screen (figure 6.11). The notification can be dismissed by the operator manually, it will be also dismissed automatically upon restarting the service. Upon restarting the service, the operator can verify in the mitigation screen that the amount of mitigated triggers increases over time (figure 6.12).

Choosing an error mitigation strategy by the operator means that every call to the `int` endpoint of $epsilon_2$ will result in an error – the raised exception will be caught by the web server's exception handler and transformed into error response. This can be verified by looking into the service mesh management system like Kiali. Upon checking the service graph there, the operator can confirm that all the requests along vulnerable paths from the

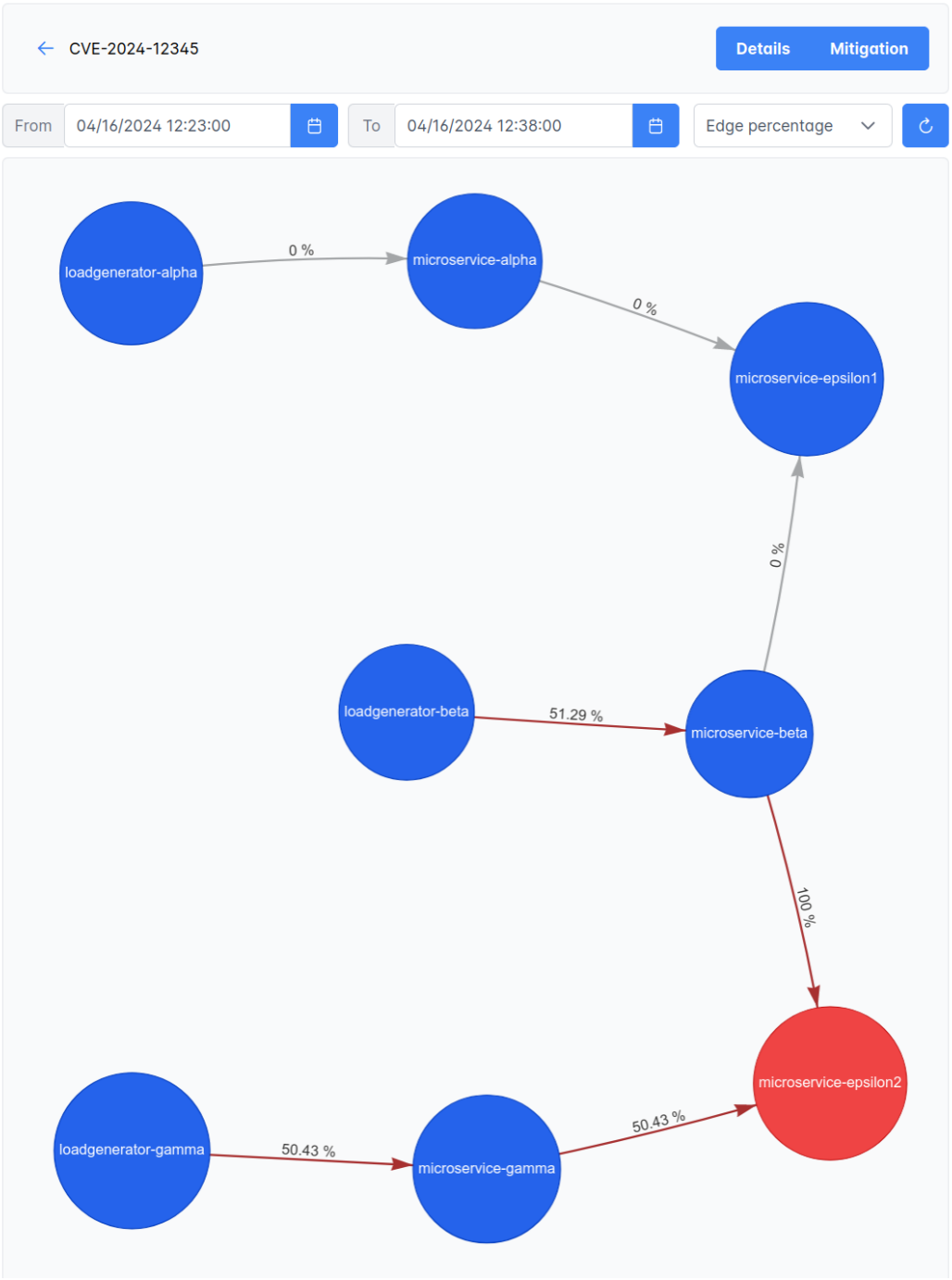


Figure 6.9: Screenshot of the impact assessment screen, showing an impact graph of *CVE-2024-12345* vulnerability on the services.

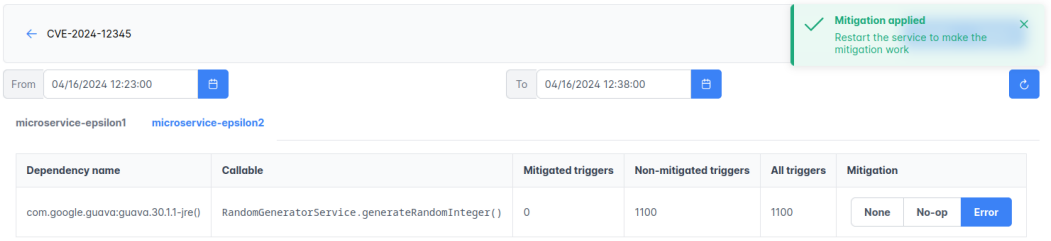


Figure 6.10: Screenshot of the mitigation screen, showing information about vulnerable callables, how often they have been triggered in the given timespan, their mitigation options, and the notification displayed upon choosing the mitigation strategy for the service's callable.

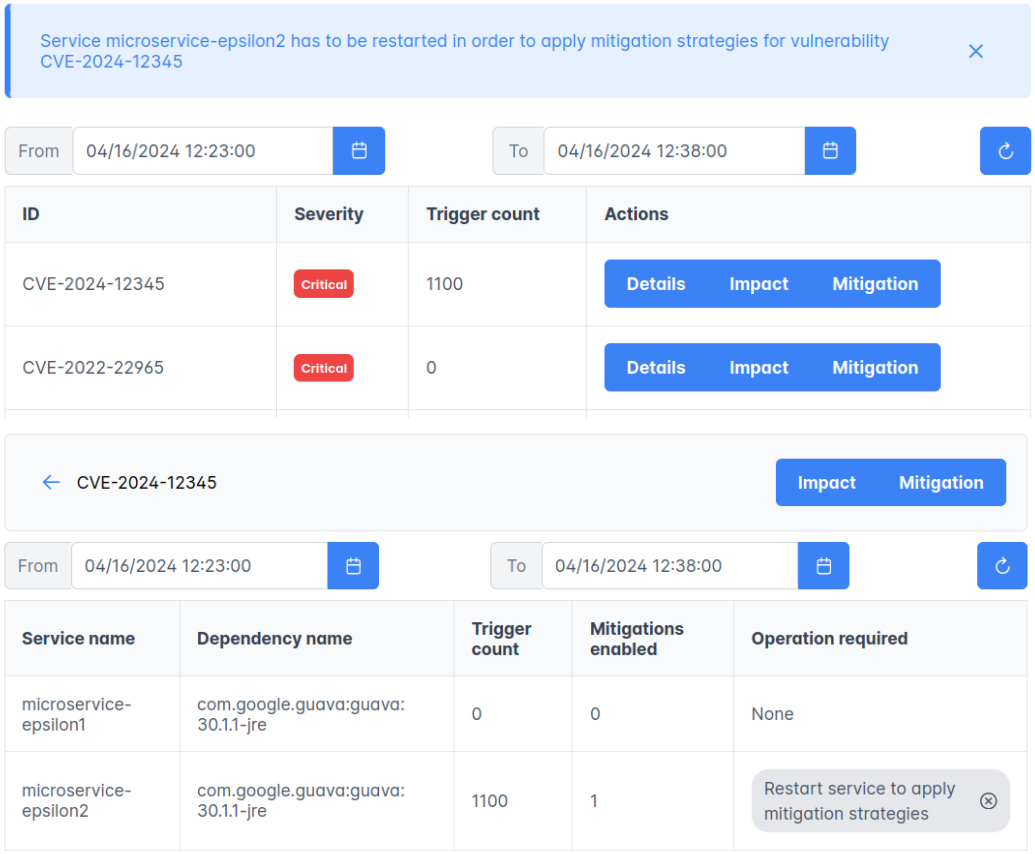


Figure 6.11: Screenshots of the dashboard and the vulnerability details screens, showing a notification about the necessity to restart the service.

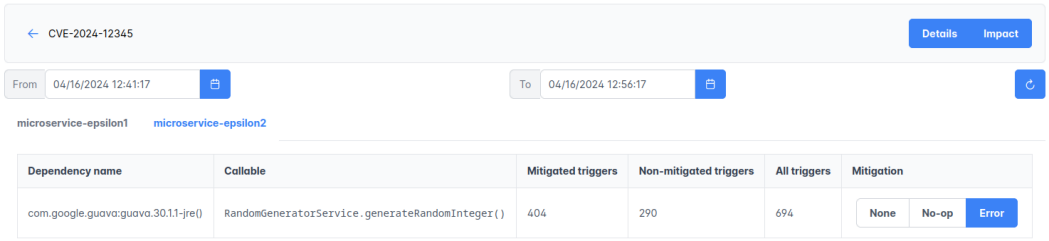


Figure 6.12: Screenshot of the mitigation screen, showing an increase of mitigated triggers of the vulnerable callable.

previously analyzed impact graph (figure 6.9) are now returning errors to their load generators. Inspection of the service mesh sidecar statistics for the service *beta* reveals that 33% of the requests, made by *beta* to $epsilon_2$, result in an error being returned by the generator service. Meanwhile, 50% of the inbound requests to service *beta* from its load generator return an error response. The same applies to the service *gamma*. The downstream call configuration, described earlier and depicted in figure 6.6, confirms the validity of the mitigation scope.

Discussion and comparison The introduction of the prototype brought an improvement over the state of the art in every aspect of the scenario. In addition to the information about the new vulnerability and dependencies that caused it, the operator has been provided with the fact that the faulty code is being executed during the system's operation. Thanks to the impact graph, they could inspect which other services are calling the vulnerable one and how much traffic between them is affected, enabled by the new metric of the trace percentage on the graph's edges. Thus, we managed to improve both the awareness and impact assessment capabilities of the operator, allowing them to assess the vulnerability's behavior within the system without involving developers or security specialists. We also improved the actionability by allowing the operator to mitigate the vulnerability and avoid calling vulnerable code. Moreover, the mitigation could be performed in a granular way, without affecting the remaining functionality of the service, without relying on feature flags to be prepared in advance, and without modifying the service's source code or its artifact. It is a big improvement over the state of the art, where the operator's actionability is limited to informing other team members and performing coarse-grained operations like taking down the service.

6.2.3 Scenario 3

Setup In this scenario, the vulnerability appears in one of the service's dependencies and is triggered during normal execution of the system. Its purpose is to show the prototype's usability in cases where vulnerability is actively exploited, but there is no possibility of applying a mitigation strategy from the pre-defined set. The downstream call targets have been configured as follows:

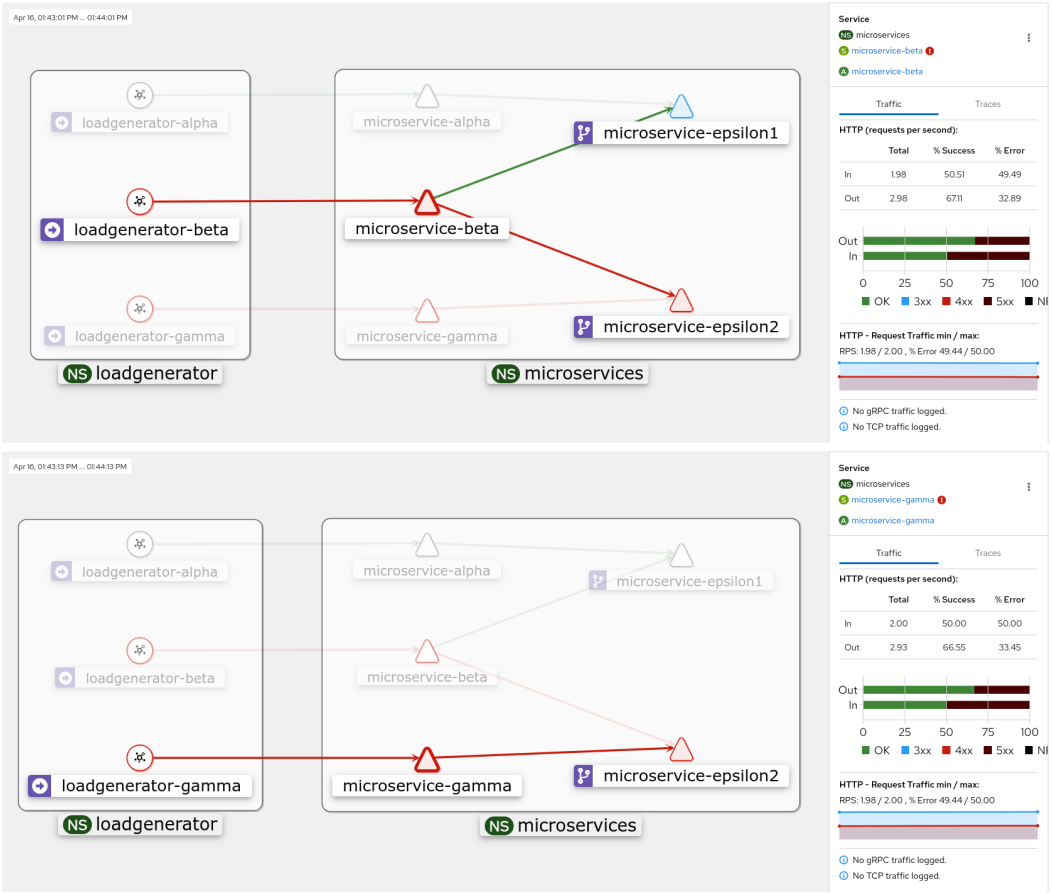


Figure 6.13: Screenshot of the service graph in Kiali, showing how error propagation from *epsilon*$_2$ service affects inbound and outbound requests of services *beta* and *gamma*.

- *alpha* – triggering endpoint `one` sends a downstream request to `uuid` endpoint of service *epsilon*$_1$. Triggering endpoint `two` sends two requests to `uuid` and `int` endpoints of the same generator service.
- *beta* – triggering endpoint `one` sends a downstream request to `int` endpoint of service *epsilon*$_1$. Triggering endpoint `two` sends two requests to `int` endpoint of the same generator service.
- *gamma* – triggering endpoint `one` sends a downstream request to `int` of service *epsilon*$_2$. Triggering endpoint `two` sends two requests to the `int` and `uuid` endpoint of the same generator service.

Here, the vulnerable endpoint `int` of service *epsilon*$_2$ is triggered only by service *gamma*. The architecture diagram of the experimental setup that shows the downstream calls, made by the services, is depicted in figure 6.14.

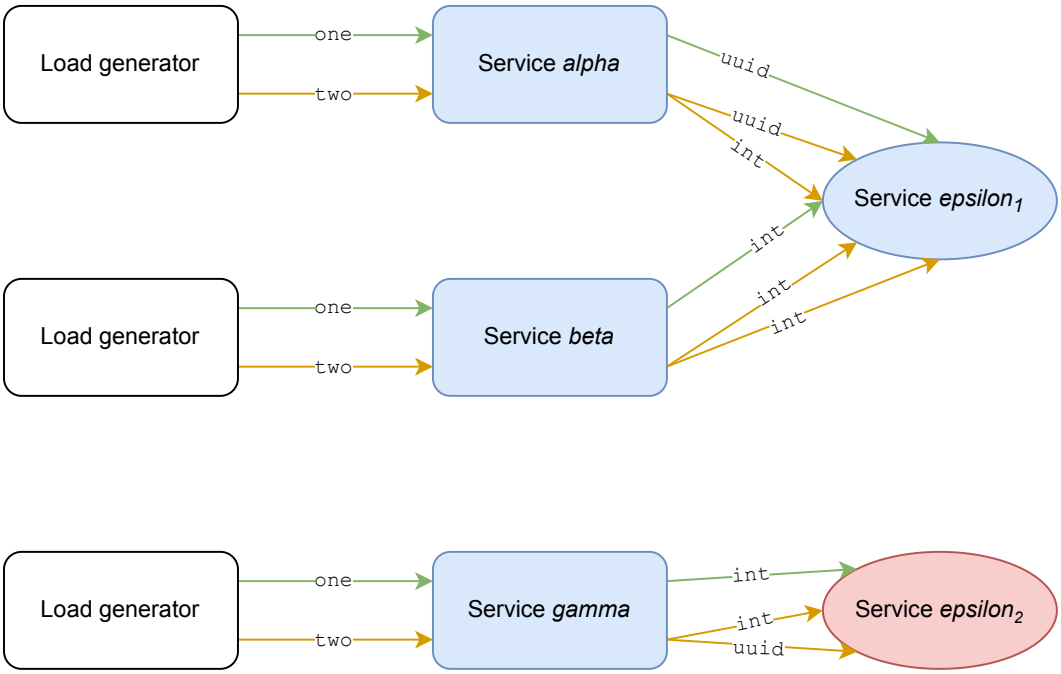


Figure 6.14: Architecture diagram of the experimental setup for the third scenario. Service *epsilon*$_2$ with reachable vulnerable code has been marked as red. Its vulnerable endpoint `int` is executed by the *gamma* service.

State of the art Just like in the previous scenarios, the operator would get informed about the new vulnerability through the pipeline or periodic security scan execution. The impact assessment would be most likely done by a developer or a security specialist, to determine the real severity of the vulnerability. Similarly to previous scenarios, there is no way to determine if the vulnerability is executed in the environment unless a traceable disturbance in the system is discovered. If the assessment reveals that the vulnerability is exploitable, the operator will be likely tasked with either disabling part of the system's functionality through configuration, or with taking the affected service down.

Prototype walkthrough The operator is informed about the new vulnerability's appearance like in the previous scenarios. After restarting services, the operator noticed a rise in the number of vulnerability triggers, which can be verified in both the dashboard and the details screen. Switching to the impact assessment screen reveals the impact graph with the vulnerable service marked in red (figure 6.15). The impact graph analysis allows the operator to deduce that:

- services *alpha* and *beta* are not affected by the vulnerability, since no path in the graph leads from them to the vulnerable service.
- service *gamma* is affected by the vulnerability – all the traces from its load generator and all the traces between it and *epsilon*$_2$ result in calling vulnerable code. Calls to the

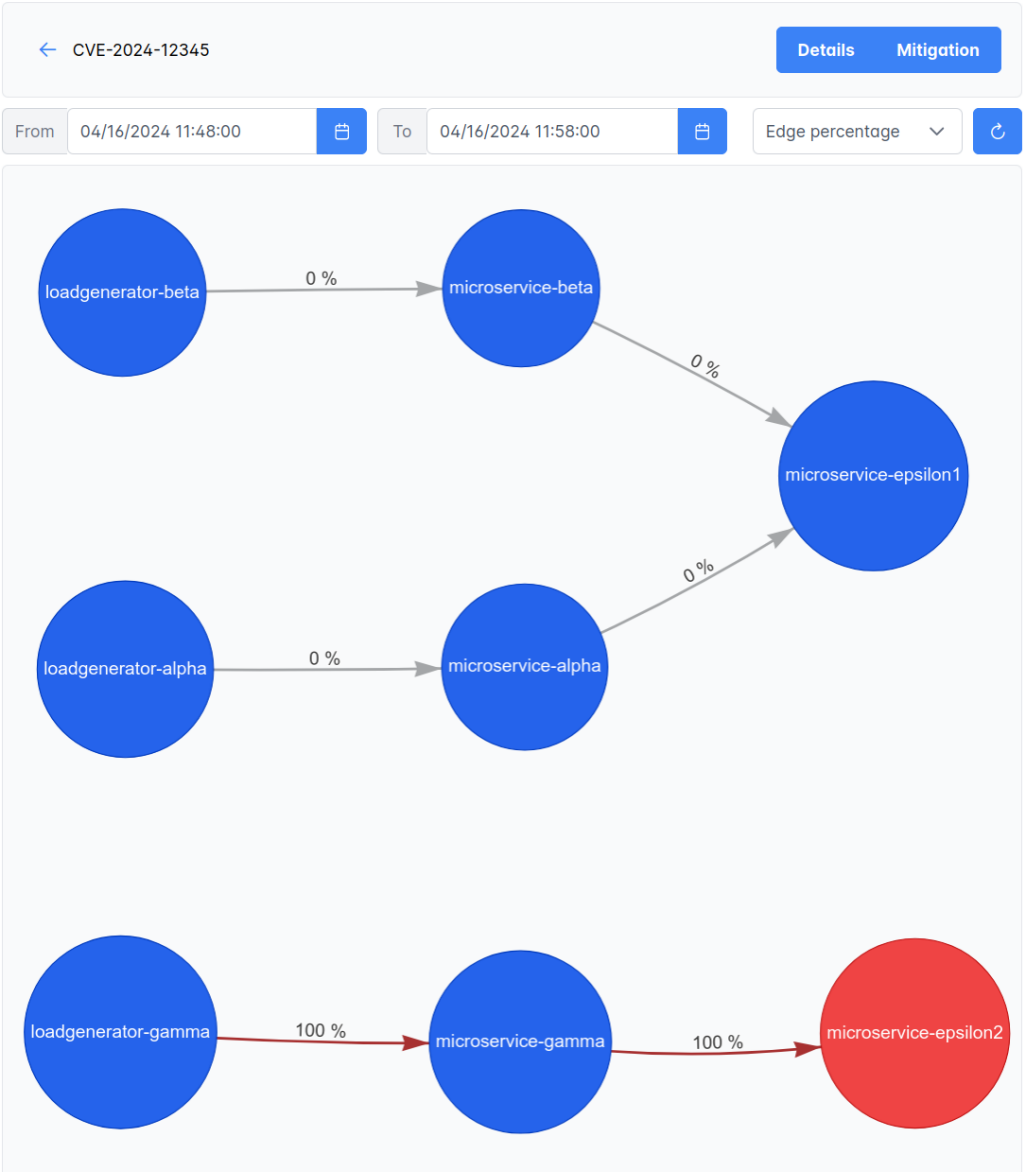


Figure 6.15: Screenshot of the impact assessment screen in scenario 3, showing an impact graph of *CVE-2024-12345* vulnerability on the services.

endpoint `one` of *gamma* trigger the vulnerability, since they make the service call the vulnerable `int` endpoint of $epsilon_2$. One-third of requests from *gamma* are made to the non-vulnerable `uuid` endpoint, as a result of calling the endpoint `two` of the service. However, since it's done in the scope of handling the request to endpoint `two` of *gamma*, it means that it's part of the same trace as another call to the vulnerable `int` endpoint.

The operator has been informed that values, generated by *gamma*, are used in other services.

Instead of applying mitigation in the production environment, they decide to do so in the test environment first. The operator chooses the no-op mitigation strategy, which replaces the result of the vulnerable callable with the default value of its returned type. For integers in Java, the default value is zero. After checking the logs in the log aggregator (figure 6.16), the operator confirms that the reported integers are not random anymore and are always reported to be equal to zero.

However, the end-to-end tests executed on the test environment after applying the mitigation have reported the incorrect behavior of the other services in the mesh. After receiving feedback from other teams, responsible for the service development and operation, the operator concludes that the error strategy is not suitable either since the amount of affected traces is too high, and error propagation would return non-meaningful generic server errors to every request from the clients. Since *epsilon*$_2$ is used only by *gamma* and no rapid mitigation options are suitable, the operator discusses the matter with other teams, decides to take down both *gamma* and *epsilon*$_2$, and display a maintenance page whenever somebody tries to access *gamma*.

Discussion and comparison Similarly to previous scenarios, introducing the prototype into this one allowed us to improve every aspect of the vulnerability handling procedure. It helped to improve awareness and impact assessment, by informing the operator about new vulnerabilities and providing a way to determine which services and how much traffic between them is affected – all without requiring in-depth analysis from the other teams. Although the provided mitigation capability has proven itself to be insufficient, it can be argued that the actionability of the operator has improved nevertheless. By the definition of actionability improvement that we provided at the beginning of section 6.2, the ability to experiment with the granular mitigation options in the testing environment helped to make a more educated decision to shut down the service on production. Without our prototype in place, the operator would be required to wait for the developer or security teams before performing any change in the environment. They would not be able to communicate the information about the runtime behavior of the vulnerability to the other teams either.

6.3 Quantitative evaluation

To determine the feasibility of applying our solution in practice, we have conducted several experiments to measure the instrumentation overhead and resource usage. The experimental setup consists of two service types: load generators and number generator services *epsilon*. The load generators repeatedly call the `int` endpoint of *epsilon* services, which execute vulnerable logic. Requests of each load generator are distributed between *epsilon* services in a round-robin fashion. The architecture diagram of the experimental setup of the services and their downstream call configuration is depicted in figure 6.17. The impact graph, generated by our prototype, can be seen in figure 6.18. All experiments were executed on a single node Kubernetes cluster, deployed on a machine with Intel Core i7-10875H CPU (8 cores, 16 threads, base clock of 2.30 GHz, and turbo clock of 5.10 GHz), 64 GB of RAM and NVMe SSD drive.

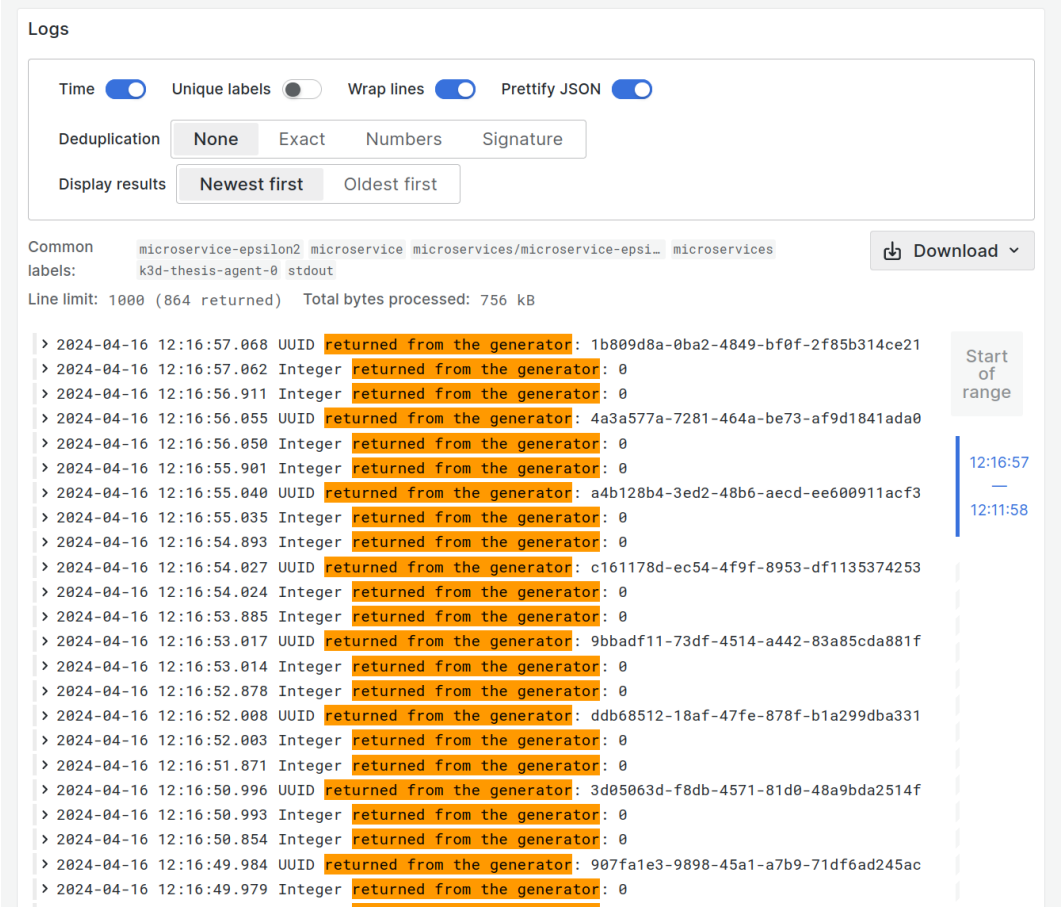


Figure 6.16: Screenshot of Grafana Loki, showing the logs of *epsilon*$_2$ service after applying the no-op mitigation to the integer generation callable. The integers, returned from the generator logic, are always logged as zero. Meanwhile, UUID generation is not affected.

We let our system run for a few minutes in five different scenarios:

1. no agent – without any instrumentation agent attached to the services
2. default agent – with OpenTelemetry agent and its default instrumentation, but without collecting vulnerable method triggers
3. collecting agent – with OpenTelemetry agent, its default instrumentation, and collecting vulnerable method triggers
4. collecting agent with mitigation – with OpenTelemetry agent, its default instrumentation, collecting vulnerable method triggers, and no-op mitigation enabled on the vulnerable integer generation method
5. collecting agent with no generation – with OpenTelemetry agent, its default instrumentation, and collecting vulnerable method triggers. The vulnerable generation logic in the

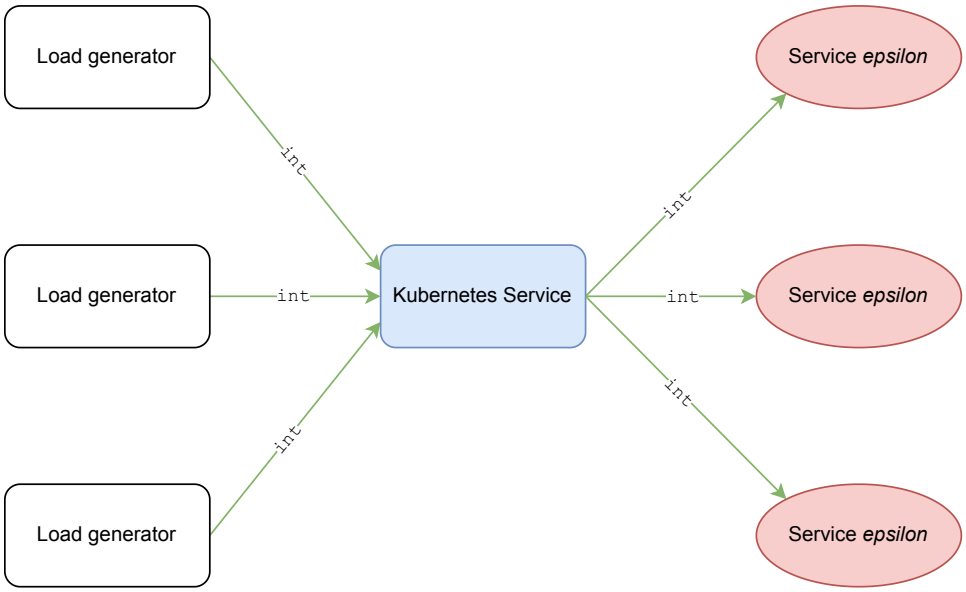


Figure 6.17: Architecture diagram of the experimental setup for quantitative evaluation. The vulnerable `int` endpoint of *epsilon* services is executed by the load generators in a round-robin fashion by using Kubernetes ClusterIP Service.

generator service is disabled. As a result, although the endpoint pf *epsilon* is called, the vulnerable method is not executed.

After each run, we collected 5000 request duration entries from load generator logs. We treat the run with default OpenTelemetry instrumentation to be the base for further comparisons, since it's a solution, commonly used in the industry to collect telemetry from the workloads. Therefore, it means that if our prototype solution were to be implemented in practice, it would most likely work together with instrumentations, provided by the default agent.

6.3.1 Instrumentation overhead

To measure instrumentation overhead, we collected the request durations from the load generators, which calculate it from the moment of sending the request until they finish receiving the response. It is important to note that the node hosted not only the instrumented microservices but also components of the prototype system, its database, and the observability stack – including log, metric, and trace aggregators – with an object storage cluster for retention. This would negatively impact the performance results due to the increased resource utilization by components such as the trace aggregator, trace graph builder, and database. These components consume more resources when the agent is active, which can affect the overall system performance and skew the observed outcomes. To mitigate that impact, we



Figure 6.18: Impact graph, generated by our prototype for the quantitative evaluation scenarios.

temporarily disabled the components that are not involved in the instrumentation overhead evaluation process, like the metric collector and aggregator, trace collector and aggregator, and our prototype's trace collector and processor. Instrumented workloads still create spans and events, even if they can't deliver them due to the trace collector being disabled.

The latency percentiles for all five scenarios can be found in table 6.1. The distributions of request durations for the first four scenarios are depicted in figure 6.19. The figure 6.20 shows a comparison between the scenario where the *epsilon* service has no-op mitigation applied to the number generation method, and the scenario where the mitigation is not applied, but number generation has been disabled.

Base latency When running services without any instrumentation agent attached, we measured the median response time to be around 2.09 milliseconds. Attaching the OpenTelemetry agent resulted in the median response time rising to 2.34. This is expected, as the agent applies the default suite of instrumentations to the frameworks, detected within the application. For example, it instruments HTTP request libraries to include trace propagation headers in the downstream calls to allow for tracking requests within the service mesh.

Scenario	25th percentile	50th percentile	75th percentile
No agent	1.76100	2.092	2.59175
Basic agent	1.97500	2.341	2.94100
Collecting agent, no mitigation	1.83000	2.203	2.85475
Collecting agent, mitigation	1.85575	2.187	2.69400
Collecting agent, no generation	1.89400	2.240	2.80600

Table 6.1: Table of request duration percentiles

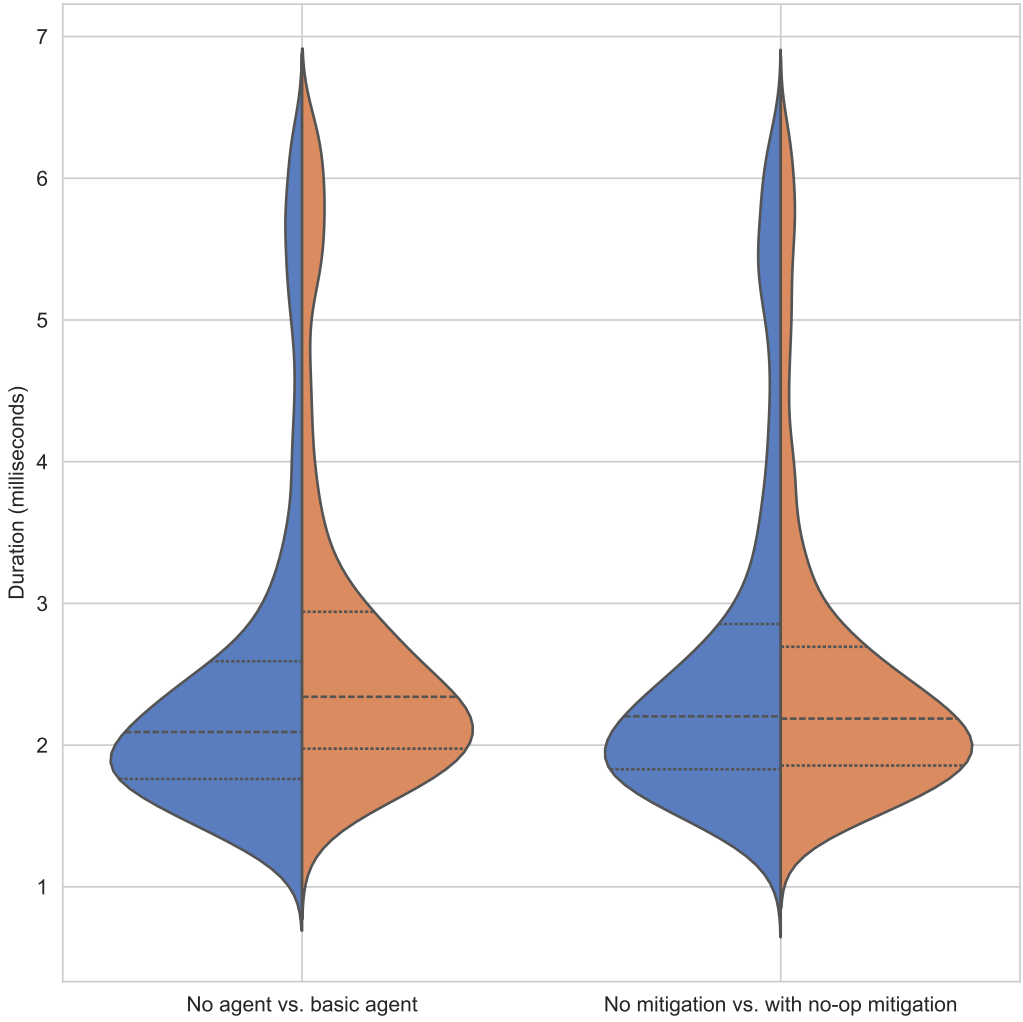


Figure 6.19: Violin plot of request duration distributions for four overhead measuring scenarios, as collected from load balancers.

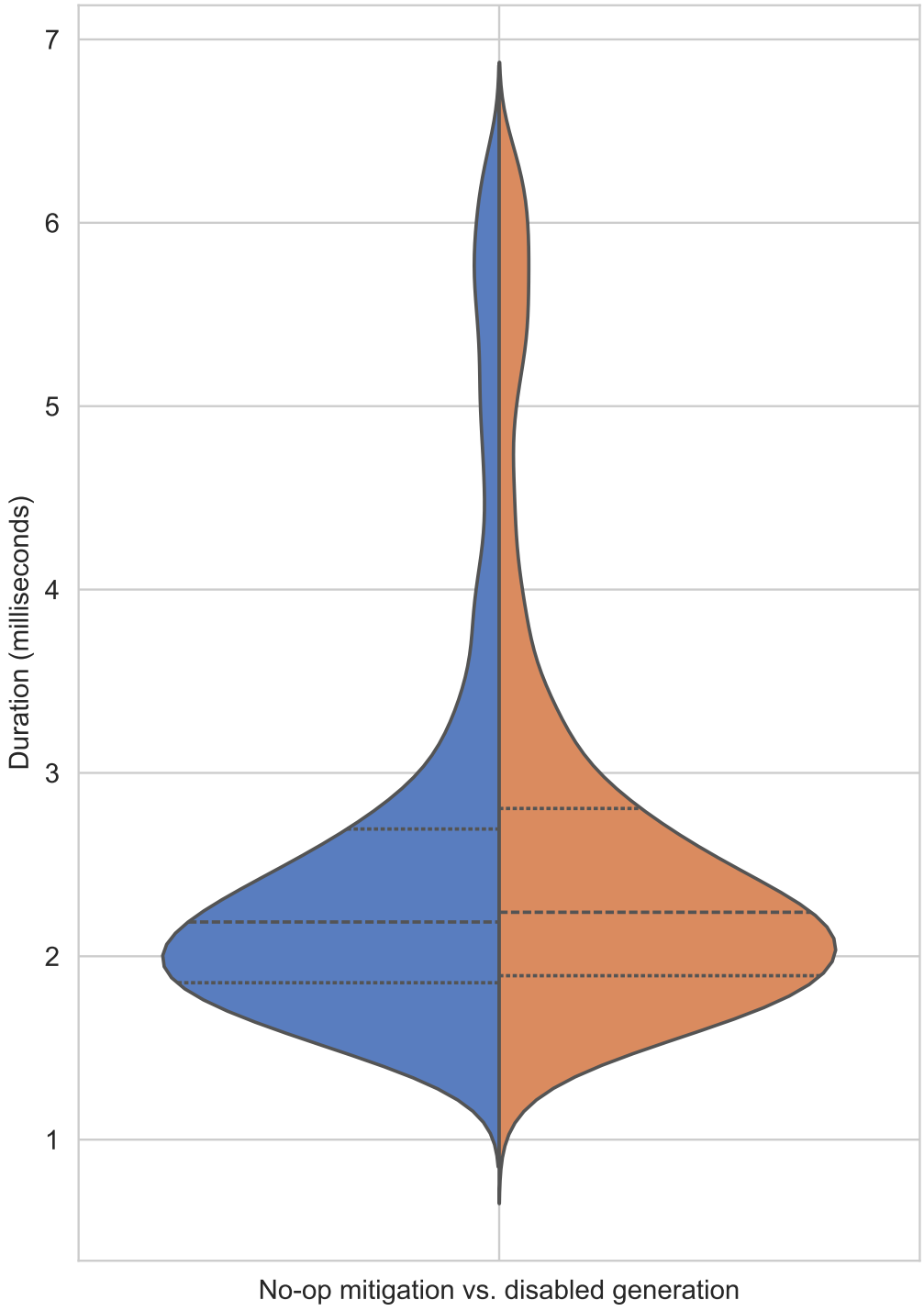


Figure 6.20: Violin plot of request duration distributions, comparing the scenario with no-op mitigation, and the scenario where the number generation has been disabled.

Latency introduced by collecting method triggers Enabling the agent extension of our prototype did not lead to a further increase in latency. Instead, the median response time with the collecting extension enabled revolves around 2.2 milliseconds. This indicates that the impact of collecting the triggers of vulnerable callables is negligible, and the range of latencies largely overlaps with one of the basic OpenTelemetry instrumentation suite. Enabling the no-op mitigation strategy did not cause an increase in latency. It stayed roughly the same, with a median request duration of around 2.19 milliseconds. The instrumentation agent applies a mitigation strategy by performing bytecode manipulation during startup, which means that the executions of vulnerable methods are as performant as if their mitigation were implemented in the code. In the case of a no-op mitigation strategy on a method that returns an integer, the only statement of that method is returning zero. After disabling the number generation logic, we observed no substantial change in latency. Disabling the number generation logic did not bring any substantial change in performance – the median latency increased a little, from 2.19 to 2.24 milliseconds, as shown in the figure 6.20. The similarity of scenarios without mitigation, with mitigation, and disabled generation logic allows us to conclude that applying simple mitigation strategies like the no-op through the startup code transformation does not introduce significant performance overhead.

Result verification To verify our claims and determine if the introduced performance overhead is indeed negligible, we conducted several statistical tests. First, we processed the experiment results and removed the outliers using the interquartile range method. Then, we checked the distribution type of our results using Shapiro-Wilk's test of normality, by calculating the statistic W and the p-value. The effect size has been calculated using the formula for Cohen's d:

$$d = \frac{W - \mu_W}{\sigma_W}$$

where W is the Shapiro-Wilk test statistic, μ_W is the mean of the Shapiro-Wilk test statistic under the null hypothesis, and σ_W is the standard deviation of the Shapiro-Wilk test statistic under the null hypothesis. For the large datasets, we can assume $\mu_W \approx 1$. To further simplify calculations, we can estimate an effect size through relative deviation from 1 and assume that $\sigma_W = 1$, which leaves us with the formula for the effect size:

$$d \approx W - 1$$

The results can be seen in table 6.2. In the Shapiro-Wilk test, the null hypothesis is that data is normally distributed. The obtained p-values were low enough to allow us to reject that hypothesis (the largest value was 1.03×10^{-59}). However, we also obtained relatively low absolute values of Cohen's d, as suggested by Cohen and further discussed by Sawlilowsky [38]. This indicates a small effect size, suggesting that while the data deviates from normality, the deviation is not substantial.

We can validate that claim further by looking at the frequency histogram, kernel density estimate (KDE) plot, and the quantile-quantile (Q-Q) plot obtained from the results of the scenario 3, depicted in figure 6.21. The frequency histogram reveals a frequency peak

Scenario	W	Cohen's d
No agent	0.80	−0.20
Basic agent	0.77	−0.23
Collecting agent, no mitigation	0.79	−0.21
Collecting agent, mitigation	0.80	−0.20
Collecting agent, no generation	0.79	−0.21

Table 6.2: Table of results from the Shapiro-Wilk test of normality, executed on the data from five scenarios.

around the median of 2 milliseconds. The KDE plot in that spot closely resembles a normal distribution curve. However, there is a non-negligible local maximum between 5 and 6 milliseconds. The Q-Q plot shows a shift of the data points from the normal reference line, further confirming the deviation from the normal distribution. The latencies, collected by running the remaining four scenarios, exhibit the same behavior.

After determining that our experiment data is not normally distributed, we paired the scenarios to compare their distributions. The pairs look as follows:

- no agent (scenario 1) and default instrumentation suite (scenario 2)
- default instrumentation suite without collecting vulnerable triggers (scenario 2) and with collecting vulnerable triggers (scenario 3)
- collecting vulnerable triggers without mitigation (scenario 3) and with no-op mitigation applied (scenario 4)
- with no-op mitigation applied (scenario 4) and with vulnerable logic not being executed (scenario 5)

To compare distributions, we used the Mann-Whitney U test. For each pair, we calculated the statistic U and the p-value. We measured the effect size by using two metrics – rank-biserial correlation r_{rb} and probability of superiority PS, defined as follows:

$$r_{rb} = 1 - \frac{2U}{n_1 n_2}$$

$$PS = \frac{U}{n_1 n_2}$$

where U is the Mann-Whitney statistic, and n_1 and n_2 are the sample sizes of the two groups. The value of the rank-biserial correlation metric can be interpreted as the association between the sample sets. The effects around 0.1 are considered small, meaning that the results are more or less equally associated between the groups. Meanwhile, the probability of superiority shows the probability that a randomly selected value from one group will be greater than a randomly selected value from the other group. In other words, the PS values around 0.5 show that the difference between the groups is very small.

The results can be seen in table 6.3. In the Mann-Whitney U test, the null hypothesis is that data from two tested sample sets comes from the same distribution. The low p-

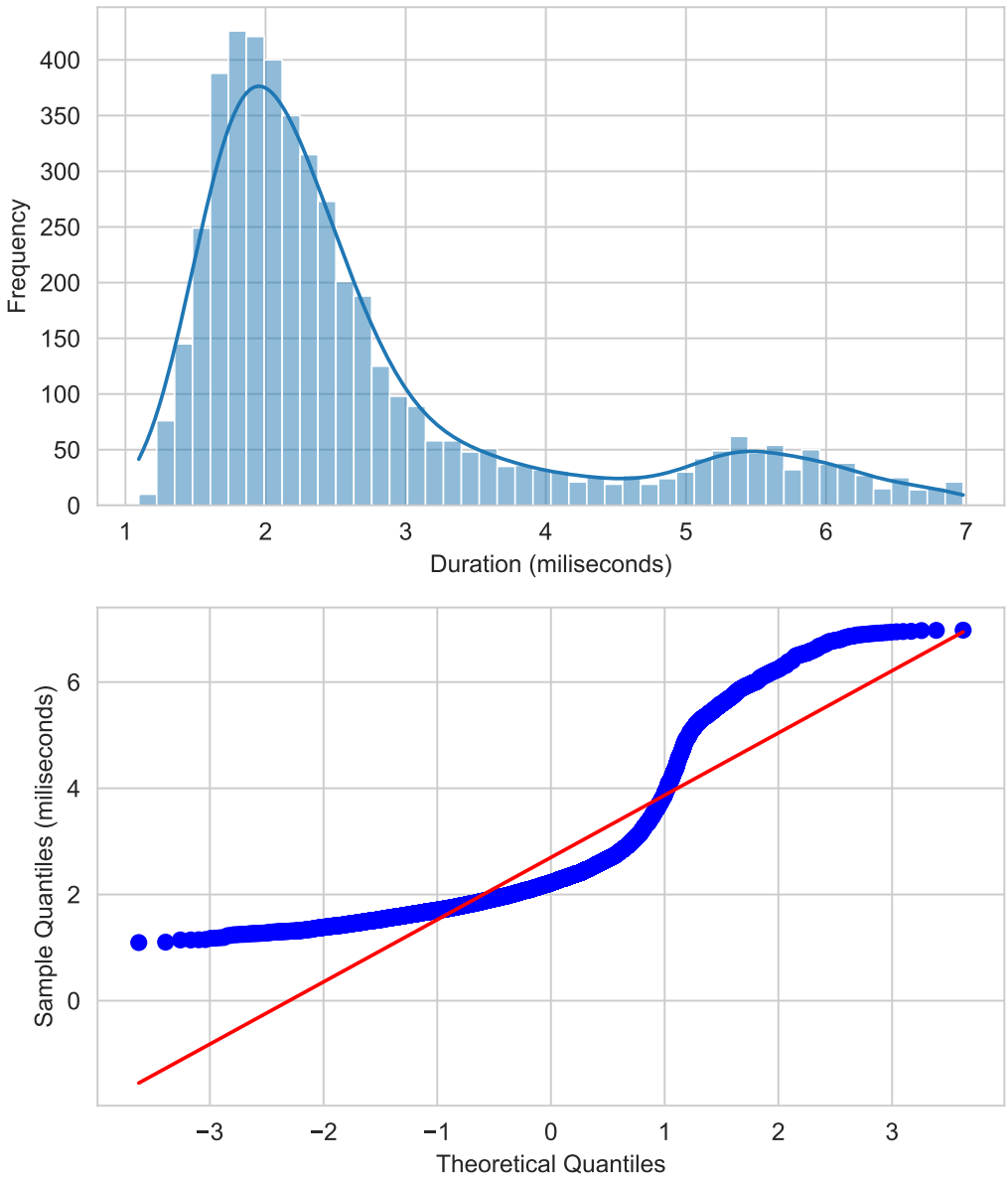


Figure 6.21: Plots from normality tests of the latencies from the scenario 3 (application with default suite of instrumentations and our custom instrumentation). The upper plot shows the frequency histogram and the kernel density estimate plot of the underlying distribution, while the lower one is a Q-Q plot, comparing the underlying distribution with the normal one.

Scenario pair	U	r_{rb}	PS
1 and 2	7 508 329	0.32	0.34
2 and 3	13 347 574	−0.15	0.42
3 and 4	11 688 150	−0.07	0.47
4 and 5	9 890 359	0.08	0.46

Table 6.3: Table of results from the Mann-Whitney U test, executed on the pairs of sample sets from five scenarios.

values allow us to reject that hypothesis for all the scenario pairs (the largest obtained value was 5.81×10^{-8}). The difference between scenarios 1 and 2 was proven to be statistically significant, resulting in a very small value of the p-value and with a relatively large effect size, backed by $r_{rb} = 0.32$ and $PS = 0.34$. This indicates that the introduction of the default OpenTelemetry instrumentation suite brings a noticeably large overhead. The difference between scenarios 2 and 3, although significant, is not as large as in the previous case. The small value of the p-value allows us to reject the null hypothesis, but the effect size is smaller in magnitude, with $r_{rb} = -0.15$ and $PS = 0.42$. Although enabling new instrumentation led to a change in request latency distribution, the difference is not as substantial as when the default instrumentation suite was first introduced. This makes sense since although we observed lower request duration percentiles after introducing our instrumentation, adding new program logic should not lead to a decrease in overhead – if anything, the overhead should either increase or be undetectable. The small effect size allows us to conclude that the observed difference after introducing our instrumentation is not significant from a practical perspective. The difference is even less apparent between scenarios 3 and 4, where the small effect size confirms that applying mitigation has negligible overhead. The small effect size for the test between scenarios 4 and 5 allows us to say that there is no practically significant overhead in our scenario even if the vulnerable method and the additional instrumentation logic are not executed.

In conclusion, while the distributions are significantly different, the actual difference in the central tendency of the distributions is not large, as indicated by the small effect sizes. This means that although the change in latencies between different scenario pairs is statistically detectable, it may not have a substantial impact in practical terms.

Additional startup time The main overhead, caused by our prototype's instrumentation agent, is its necessity to communicate with the coordinator module during startup to fetch vulnerable callables and their mitigation strategies. This depends on the performance of the network between the application and the coordinator, the CPU load of the machine where the coordinator is deployed, the performance of its database, and the performance of vulnerable callable sources. The synchronous nature of this process means that the additional latency, introduced on communication with vulnerable callable sources like FASTEN, will make the application take longer to start. In such a case, the application has to wait for the instrumentation agent, which then has to wait for the coordinator module that waits for the callable source. The coordinator module aims to mitigate that risk by caching information

about vulnerable dependencies and their callables in its database during the first registration of the service. This means that application restarts will not cause the coordinator to communicate with a callable source. Instead, the service will be provided with the list of vulnerable callables that are stored in the database. The database can be then updated asynchronously, along with a notification for the operator about the necessity to restart the services. The new callables will be then provided to the agent after the service is restarted.

6.3.2 Resource usage

To further determine the feasibility of our solution, we measured the resource usage in two areas: the memory and the storage usage.

Memory To measure the additional memory usage, introduced by our solution, we deployed five instances of the *epsilon* generator service, as described in the beginning of section 6.3. All instances had their vulnerable `int` endpoint executed by load generators. We measured the memory usage of our applications by collecting Java virtual machine memory metrics, reported by applications, and summed up all the memory types (non-heap and heap for eden and survivor spaces[3]) and presented them in figure 6.22. We also calculated the per-second average rate of increase of all the memory types over 5 minutes and summed up all the increases. The results can be seen in figure 6.23.

Services with a default suite of OpenTelemetry instrumentations have roughly the same memory usage increase rate as the ones without any instrumentation applied. The difference in memory usage, measured by comparing the usage after garbage collection, shows a negligible difference between scenarios with no agent and with the default OpenTelemetry suite. The introduction of our vulnerability tracing instrumentation leads to higher memory usage and a higher usage increase rate, with a significant rise when vulnerable callables are executed. However, although more memory gets allocated faster, by looking at memory usage after garbage collection we can deduce that the agent requires approximately 75 MB of memory regardless of the fact if vulnerable callables are executed or not. This shows that although the memory usage stays deterministic, there is room for improvement and optimization of our solution, especially considering the negligible impact of the default suite of OpenTelemetry instrumentations.

Storage The backend components of our prototype store data in a relational database, while the instrumentation agent appends events to distributed traces, stored in Grafana Tempo. The events have negligible effect on the size of stored trace data. However, our prototype collects traces, processes them into trace graphs, and stores them in the database alongside the vulnerable method trigger events. It means that the size growth of the database is directly correlated with the amount of traces generated by the service mesh. We started executing our experiments on an empty database and an empty Tempo object storage bucket.

[3]In Java, the eden space is the region in the heap where objects are initially allocated, while survivor spaces are the areas where objects are moved from the eden space if they survive garbage collection, aiding in efficiently managing memory by filtering out short-lived objects

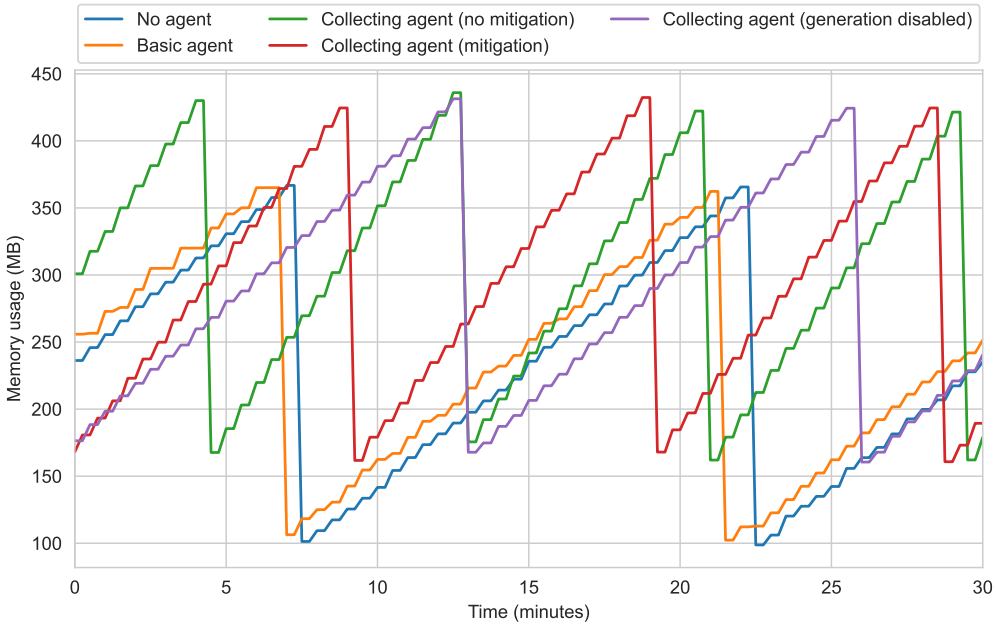


Figure 6.22: Memory usage of two *epsilon* service instances. The green plot shows the memory usage of the service with the basic OpenTelemetry instrumentation suite. The yellow one shows the memory usage of the service with the basic instrumentation suite and our prototype's instrumentation logic.

Then, we measured the rate of size growth for both the database and Tempo's bucket when running the scenario with our instrumentation agent attached.

As we can see in figure 6.24, the steady usage of the system leads to linear growth in storage usage. This can be validated by looking at the usage of the coordinator module's PostgreSQL database, which allocates space when needed. The usage of Tempo grows in steps, as it allocates new chunks in its object storage. This is a behavior, specific to the Parquet format used by Tempo.

Scalability The negligible overhead of our solution's instrumentation, combined with stable memory usage means that our solution is fairly transparent to the workloads. As a result, scaling the system up will not lead to a rapid increase in resource usage on the machines where workloads are deployed. When all the services in the mesh are instrumented, the introduced latency will compound for each chain of downstream calls. However, our experiments have shown that the additional performance overhead of our solution is negligible when compared with the basic OpenTelemetry instrumentation suite.

As the increase in traffic within the system will lead to more spans being created, the trace graph generator components of our prototype will process and persist more data into the database, increasing the storage requirements of the system. However, the generated

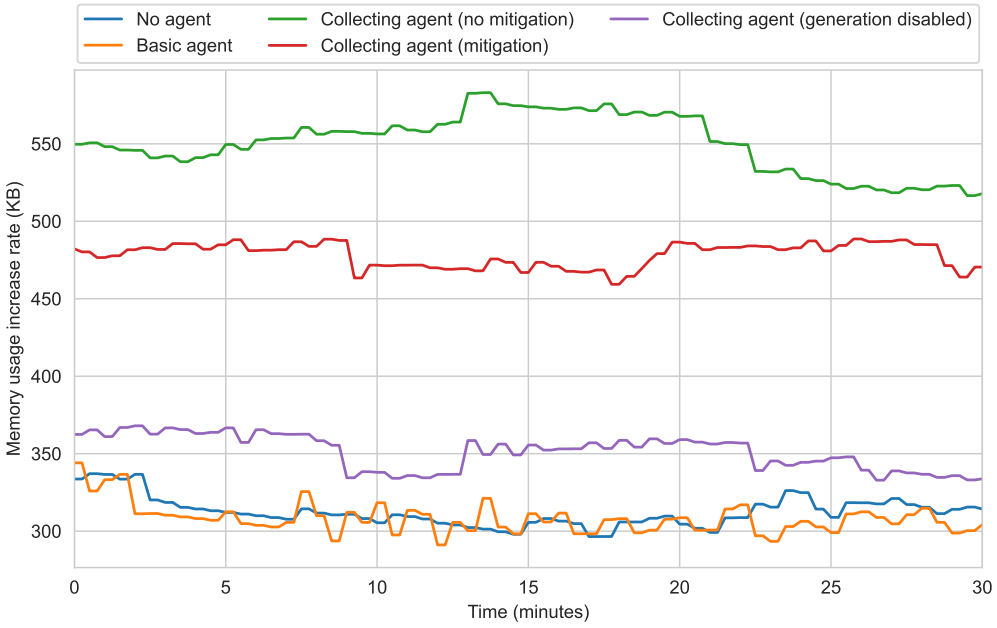


Figure 6.23: Memory usage of two *epsilon* service instances. The green plot shows the memory usage of the service with the basic OpenTelemetry instrumentation suite. The yellow one shows the memory usage of the service with the basic instrumentation suite and our prototype's instrumentation logic.

trace graphs will not take more space than traces in the trace aggregator do, since the graphs are generated from the subset of traces from the aggregator.

The trace graph generation is based on periodic collection and processing of the traces, obtained from the trace aggregator. Since the collection is done by fetching a fixed amount of trace identifiers from the fixed interval sliding windows, it might lead to a situation where the number of traces in the aggregator is too large to be handled by a single instance of trace collector and processor. If the procedure of trace identifier collection starts to fall behind, some traces might be lost and not processed into the trace graphs. If the trace processing starts to fall behind, it will take longer for the new vulnerability triggers to be reported to the operator. These issues can be solved by deploying additional collector and processor instances for building trace graphs. It might be necessary to also scale up the trace aggregation system to sufficient querying capacity.

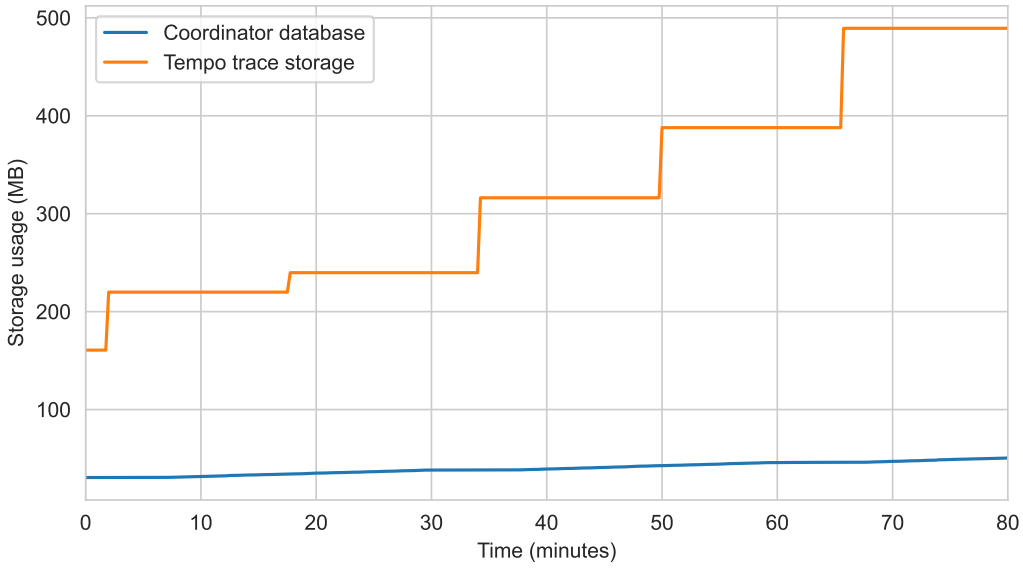


Figure 6.24: Storage usage comparison of coordinator's database and Grafana Tempo trace aggregator.

Chapter 7

Discussion

This chapter discusses the implications of the findings obtained from evaluating our vision and prototype. We will acknowledge the limitations of the current approach and threats to the validity of the evaluation results. Finally, based on the vision and prototype evaluation results, we will determine areas that could be explored further in future research.

Vision evaluation findings We validated our vision by conducting interviews with ten software engineers who have experience in system administration and operation. Employing a combination of interviews and grounded theory methods proved instrumental in determining that the envisioned system has the potential to optimize vulnerability handling pipelines currently used in the industry. The awareness improvement is achieved by shortening the time between vulnerability disclosure and informing the team about their software being affected by it. Interviewees also pointed out the novelty of collecting vulnerable code triggers during runtime and overlaying them on the service mesh graph, which allows for a more accurate assessment of the vulnerability's impact. The versatility of mitigating vulnerable code on the callable level was acknowledged as a vast improvement in the actionability for all the parties, involved in the vulnerability handling process. Currently, the disclosure of a high-risk vulnerability in the operated system often requires undertaking disruptive measures. Some of them like shutting down the entire affected component are often rejected by the management departments, which deem them to be too risky from the business perspective. Being able to granularly avoid executing vulnerable code without interfering with other executions of the component brings multiple benefits from both security and planning perspectives. The obvious advantage is that the vulnerable code cannot be exploited anymore. Since our solution does not require code or artifact modifications to apply the mitigation, it can be done in a matter of minutes. This is much faster than the usual time indicated by interviewees. For most of them, it takes between a few hours and a few days before they can prepare and deploy any short-term mitigation to production. Since the short-term solution can be quickly introduced to the affected environment, developers can focus on mitigating the issue in the code base, without the additional stress of knowing that vulnerability can be exploited in the live system. Moreover, if the ongoing business priorities do not allow for developing long-term mitigation straight away, its development can be postponed without sacrificing security. All interviewees stated they would like our envisioned tool to be

part of their vulnerability handling pipeline. They pointed out its usefulness in lowering the risk of vulnerability exploits, by providing a short-term solution while developers work on long-term fixes. They also highlighted how having access to information about actual vulnerability triggers helps to prioritize fixes and avoid alert fatigue.

Prototype evaluation findings Consistent with the theoretical predictions, the prototype's empirical evaluation affirmed our vision by showing how to optimize vulnerability handling pipelines in practice. The scenarios of the qualitative evaluation show that the prototype helps in improving awareness, actionability, and impact assessment capabilities as envisioned. It informs users about the vulnerability in the components of the operated system immediately after information about new vulnerabilities gets obtained from the source of vulnerable callables. It also allows users to determine if the vulnerability is being triggered during the system's operation. The impact graph shows which components of the service mesh are affected and how many traces between them execute vulnerable logic. Our scenarios also demonstrated the usefulness of the impact graph's edge metrics in determining the most suitable course of action. Finally, being able to choose different mitigation strategies for vulnerable callables allowed for neutralizing the exploit risk or guiding towards undertaking other measures if deemed unsuitable. Further research could aim to define other impact graph edge metrics and explore their usability in scenarios where they could be beneficial in determining the impact of vulnerability and the most suitable mitigation strategy. Another field to explore is the way how the callable-level mitigation strategies themselves are defined. One could look for ways of defining granular mitigations more generically and evaluate their usability through the walkthrough scenarios, or by surveying industry practitioners, similar to how we did it when evaluating our vision. Moreover, the research could explore methods for defining mitigation strategies that are more specific and tailored, while remaining reasonably technology-agnostic and not requiring the system's user to write code.

Feasibility of implementation Quantitative evaluation of instrumentation overhead and resource usage showed that the envisioned solution can be implemented without requiring excessive amounts of resources and without slowing the applications down. The additional latency introduced by our extension of the OpenTelemetry agent to collect vulnerability triggers is negligible and remains within the performance bounds set by the base OpenTelemetry agent's default suite of instrumentations. The agent implementation has a deterministic impact on the resource usage of instrumented workloads and does not cause memory explosion when used. Meanwhile, the volume of data it generates is comparable to and correlates with the data produced by other observability measures.

Usefulness for other parties than the operator Contrary to the initial hypothesis that the envisioned tool would be used mainly by the system operator, the interviews revealed that it could be helpful to other members of the team as well. As the DevOps practices evolve, many software engineers are assigned both development and operational tasks, meaning that they could also benefit from the shorter response time, provided by the system. The

high usability of the envisioned system, reported by participants in the SUS questionnaire (as discussed in section 4.2.8), further confirms that it could be used not only by technical people like developers and operators but also by testers and project managers – assuming that their technical background is sufficient enough.

Distribution of interview participants A potential external threat to the validity of these findings could be the bias in the distribution of the interview participants. Due to the small scale of the survey, most of the participants were software engineers, whose main task at work is software development. We attempted to mitigate that by inviting people with operational backgrounds, who either used to work as system operators or still perform operator-specific tasks like deployments, monitoring, infrastructure administration, and server maintenance. Future studies should aim to survey more industry professionals and make sure that the range of their professions is evenly represented.

Evaluation scenario extremity It is necessary to point out that our quantitative evaluation experiments were run with a 100% trace sampling ratio, where each executed trace triggered vulnerability at least once. In real-world scenarios, the sampling ratio for tracing would typically be lower and only increase temporarily during specific periods. Furthermore, traces that trigger vulnerabilities would constitute a minority of all collected traces. It is up to the system operators and the rest of the team to determine the sampling ratio that will provide the most representative results for the operated system, and to configure data retention policies. However, low sampling also means that some vulnerability triggers might not be detected if traces of their events do not get reported. This can be mitigated by having our instrumentation agent report its traces not only to the aggregator but also to a separate component, which would react to the detection of vulnerability triggers and temporarily increase trace sampling within the service mesh to detect the ongoing behavior more accurately.

Impact of experimental setup components on the latency Moreover, since our experimental setup was deployed in the same single-node Kubernetes cluster as the prototype system and its dependencies (as discussed in more detail at the beginning of section 6.3), the results of the experiments represent an unfavorable scenario where the latencies of non-instrumented workloads are noticeably lower than instrumented ones, even without our custom extension enabled. This is why we focused on comparing the overhead in two scenarios – with the OpenTelemetry agent and its suite of default instrumentations, and the same agent with our instrumentation enabled on top of the default ones – to obtain the most accurate representation of the impact of our solution on the workloads. In the real system where workloads have dedicated cluster nodes, the computed impact would be lower. We attempted to mitigate the impact of running everything on a single machine by temporarily disabling components that were not necessary for the scenario. For example, when measuring the overhead of the agent, we collected request durations from the load generator's logs. This allowed us to disable trace and metric collectors and aggregators, as well as trace graph collectors and processors of our solution.

Procedure of building trace graphs The presented algorithm for building trace graphs has been tested on a limited number of experimental scenarios and might not apply to all trace topologies. There is a risk that in certain corner case scenarios, the reduction steps of merging and pruning span nodes are not sufficient to turn the trace span hierarchy into a graph that can be used in building impact graphs. Further research could look into defining an exhaustive test suite, consisting of more service mesh topologies. These tests could be used to identify scenarios in which the current reduction procedure fails to generate a correct trace graph. Then, the results could be used to refine the procedure to cover these cases.

Verifiable mitigation strategy applying During our interview study, participants expressed their concern regarding the way how instrumentation agents are provided with the mitigation strategy. Agents fetch information about vulnerable callables and their mitigations from the coordinator service, which stores them in the relational database. Obtaining read-write access to the database would provide a malicious actor with the possibility to modify the behavior of arbitrary callables in instrumented workloads. For example, applying no-op mitigation to callables, responsible for authentication and authorization, could allow anybody to log into the instrumented system without providing valid credentials. This creates a new attack area, where a correct code could be turned vulnerable without access to the code base or even build and deployment interfaces. It could be alleviated by designing a security protocol, allowing agents to verify the authenticity of applied mitigation strategies. A straightforward way to do this would be by creating a dedicated public key infrastructure, where a root certificate authority is used to issue certificates for system operators. Operators can then use their private keys to sign each applied mitigation. Providing agents with the root certificate allows them to verify the signatures of received mitigations before applying them. Provided that the private keys are stored securely (*e.g.* on a hardware key), and the malicious actor does not have access to the agent's configuration (to change the root certificate), it provides a framework to reject malicious mitigations, inserted into the database. Future research could look into other risks, introduced by providing agents with instrumentation logic from the external source, as well as reinforcing or redesigning the provided protocol.

Real-world quantitative evaluation of the prototype The quantitative evaluation of the developed prototype has been performed in an experimental setup, designed around an unrealistically pessimistic scenario where every ingress request triggers vulnerable code. It would be beneficial to embed the prototype agent in the real-world microservice system and collect latency and resource usage metrics from it.

Asynchonous processing As for the time of writing, the OpenTelemetry community is still standardizing the semantics of handling distributed traces in the systems with asynchronous processing. Currently, the preferred way of doing so is by utilizing span links, which can define a causal relation between spans in the same or different traces. This creates a question if the trace graph processing pipeline should follow the span links and merge the graphs from the different spans or leave them as separate ones. Moreover, the nature of

asynchronously processed requests means that it might be difficult to determine the time, required to complete handling the ingress request. This could lead to some traces being fetched too soon by the collector, resulting in incomplete trace graphs. To handle asynchronous cases it might be necessary to utilize additional metadata like the span statuses, set explicitly by the application whenever request processing is fully finished. The collector could then use this information to postpone processing traces that are not ready yet.

Defining new vulnerability handling methodologies Having a way to get informed immediately after the vulnerability in software is disclosed or exploited creates fresh ground for defining new methodologies and pipelines for handling vulnerabilities in developed software. Future work could investigate leveraging the integration of the envisioned system with external project management tools like Jira to establish scenarios and procedures that could be followed in enterprise settings to efficiently mitigate vulnerabilities in the short and long term.

Chapter 8

Related work

In this chapter, we will discuss the state of research in the fields, covered by our study.

Processing telemetry signals Observability measures like logs, metrics, and traces were designed to be explored and browsed directly by developers and operators through dedicated dashboards like Grafana or Kibana. However, as the systems get more complex, the need for aggregating and alternative representation of the signals becomes more apparent. Major et al. [17] suggest that the high cardinality and high dimensionality nature of raw telemetry data makes runbooks[1] and dashboards ineffective in handling modern systems, saying that they "*rarely fail in precisely the same way twice*". A literature study by Cândido et al. [14] revealed that log analysis is a mature field. Researchers attribute that to the ease of data collection in the world of observability and the large size of data sets, which is paramount in creating machine learning-based analysis solutions. For instance, processing logs has proven to be beneficial in the cybersecurity area, especially for anomaly detection. Abad et al. have established a proof of concept [4] of correlating logs from different sources to detect unusual behavior within systems through the use of data mining. Instead of treating logs as events, the approach proposed by Lv et al. [26] aims to utilize the context information within the log by utilizing natural language processing methods. Viola et al. tackled the problem of extending anomaly detection context by combining logs with metrics [41]. Aharon et al. [7] developed an algorithm for summarizing complex system logs with the use of metric-like graphs, facilitating efficient visualization of logs and simplifying system debugging.

Processing distributed traces Compared with logs and metrics, distributed tracing is a fairly new observability measure, dating back to 2010 and the introduction of Dapper by Google [39]. Its creators admitted that initially it was supposed to be a self-contained tracing tool. However, it evolved into a monitoring platform that led to the creation of other analysis utilities. As the popularity of tracing increases, so does the amount of research that explores its applications in large-scale system management and security. In their study, Zvara et al. [47] designed a framework that utilizes traces to optimize data stream processing in applications using Apache Spark. Cassé et al. [15] used traces to model microservices and

[1] standardized routine procedures and operations, performed repetitively by the system operator

their dependencies as graphs and used them to pinpoint inefficient resource composition in cloud environments. Al Maruf et al. [8] used logs, metrics, and traces to model the standard behavior of the system and detect anomalies, effectively extending the already-established anomaly detection algorithms with data from distributed traces. Despite that, distributed tracing remains a relatively new area of interest among researchers. Bento et al. [11] concluded their literature review by saying that "*there is an underused wealth of actionable information that can be extracted from both metric and morphological aspects derived from tracing*".

Qualitative research in software engineering The establishment of grounded theory in the late 1960s [22] provided a systematic methodology for generating theories based on empirical data. Instead of testing existing theories, this approach allowed performing research grounded in real-world observations and experiences. Researchers use it to study software engineering methodologies and their human aspects. Its data-driven approach made grounded theory and coding a popular qualitative research method in computer science, which leads to studies that focus on the specificities of using it in the context of computer science and optimizing the data analysis procedures. Adolph et al. [5] provided a model to help computer scientists apply classical grounded theory in the software engineering setting. In another paper [6], they established the four-stage process of software development and empirically evaluated how its first two parts depend on the interaction with other team members, while the last two require decreasing interaction to focus on implementation.

Researchers combine grounded theory aspects with other qualitative study methodologies to broaden the scope of research. Razali et al. [35] shared their experience of combining grounded theory with phenomenology and discussed how they can be used to qualitatively study phenomena in software engineering. They discuss that phenomenology helps to understand the individuals' experience regarding a phenomenon, while grounded theory allows one to explore it in depth with the help of these individuals.

Guzzi et al. [23] envisioned a tool to improve developer collaboration within integrated development environments (IDEs). The study consisted of brainstorming and semi-structured interviews, which helped to determine the problems with collaboration in large development teams. Identifying improvement opportunities through the card sort method led to designing a prototype in the form of a mockup slide deck. It was then submitted to the second round of interviews to evaluate the improvements qualitatively. Their research and interview outcome analysis methodology was an inspiration for evaluating our envisioned prototype qualitatively.

Davidson et al. [18] qualitatively evaluated the user experience of distributed trace visualization tools. In their study, they conducted interviews among industry practitioners and applied coding to the outcomes, allowing for the identification of shortcomings in the commonly used tools. Their findings led to deriving guidelines for improving trace visualization, facilitating more effective usage of collected trace data.

System usability evaluation The usability of the system in software engineering can be described as its ability to be used by the consumer or user in a way that is both effective

and satisfying for the user. Researchers study different aspects where usability evaluation can be applied in the software engineering process. Ardito et al. [9] focus on the practical application of usability engineering in organizations and how it helps to deliver better quality products and maintain user satisfaction. Horsky et al. [24] talked about how different methods of usability evaluation should be applied in the different stages of software design and development. Deriving a framework allowing for evaluating usability quantitatively has been a common topic of many research studies as well. Paz et al. [32] identified over a thousand publications that provide frameworks for evaluating the system's usability. According to their analysis, most of these frameworks utilize questionnaires as the assessment tool.

To evaluate the usability of our envisioned system, we used the system usability scale (SUS) [12] questionnaire-based method. Its wide applicability, relative simplicity, and the possibility of aggregating the results into a single score number allowed for performing comparative studies that empirically evaluate the score's relevance in the industry context. Bangor et al. [2] analyzed a large number of SUS scores, collected from many products in all phases of their development. They established that SUS questionnaires succeeded in establishing a subjective rating of the product's usability. In another paper [3], they evaluated the correlation of the aggregated SUS score with the absolute judgment of the evaluated system's usability. They extended the standard SUS questionnaire by adding a seven-point adjective rating scale as an eleventh question. By performing 1000 modified surveys, they concluded that the scores of the auxiliary question correlate well with SUS scores. Sauro [37] discussed interpretations of the SUS scores and derived usability classes, providing more descriptive meaning to the questionnaire results.

Vulnerabilities in application dependencies The introduction of vulnerabilities into developed software through the usage of external libraries and dependencies is an active research field. Most of the modeling solutions revolve around representing dependency relations between projects in the form of a directed graph. This approach makes it easy to determine if software is vulnerable, by performing graph traversal along the edges and determining vulnerable dependencies. Li et al. [25] used graph representation to develop a tool for analyzing publicly known security vulnerabilities in project dependencies. They used it to perform a large-scale empirical study involving over 300000 projects and relations between them. The study revealed the limited remediation of vulnerabilities among the observed projects – only around 2% of insecure dependency relationships have been fixed during an eight-month observation period. Márquez et al. [21] propose an approach to constructing dependency graph networks and analyzing them using satisfiability modulo theories (SMT) to identify vulnerability-free configurations of dependencies.

The current state-of-the-art in vulnerability detection revolves around marking dependencies as vulnerable whenever there is a disclosed vulnerability that affects the particular version. Several studies highlight that this approach is coarse-grained and usage of the vulnerable library version does not mean that the faulty code is being executed. Molin et al. [29] proposed a software component analysis approach, which determines the prevalence of vulnerabilities within the system and helps to assess their impact. Their solution helps in prioritizing long-term vulnerability mitigation so that the most pressing issues are tackled first.

The problem largely stems from the fact that vulnerability catalogs do not contain information about vulnerable callables within vulnerable dependency versions. Overcoming that limitation is another active research field. In their studies, researchers aim to derive new methodologies that increase the detection granularity, allowing for detecting dependencies that are not deployed and do not pose a risk. Paschenko et al. [30] created a method that avoids inflation of numbers in vulnerability counting. They discovered that around 20% of dependencies affected by the vulnerability do not have their code deployed and cannot be exploited. In another project [31], they propose a methodology to further decrease the number of false alerts for dependencies incorrectly flagged as vulnerable. Their solution also helps to predict when the dependency's maintenance becomes unsatisfactory, which might pose a risk of new vulnerabilities not getting fixed after the disclosure.

Another common approach is to determine vulnerable callables through the usage of machine learning. Russel et al. [36] trained a model to detect vulnerable functions in C/C++ by using publicly available code repositories. They used a labeled vulnerability dataset and created their own by annotating the functions with information from static analyzers. Vo [42] created a framework for detecting vulnerable commits in source code repositories. They represent each commit as a graph and use a graph neural network model to capture their specifics. The statements of vulnerable commits are then assigned a suspiciousness score. Fu et al. [19] chose to use natural language processing methods and developed a transformer-based vulnerability detector, increasing the granularity of vulnerability detection to the level of individual statements.

In 2020 the FASTEN project [1] has been started. It aims to develop software package management methods that improve the security of deployed systems. Its goals are achieved in large part through building call graphs of software libraries, and connecting them whenever a particular library calls another one. Moreover, it combines scanning vulnerability catalogs and source code repositories to determine the parts of the libraries, which are problematic, down to the level of individual callables. Mir et al. [28] explain how using FASTEN and increasing the granularity of vulnerability detection allowed them to determine that from 3 million Maven packages only 1% of them has a reachable call path to vulnerable code, even though about 30% of these packages were marked as potentially vulnerable during naive dependency analysis.

Chapter 9

Summary

In conclusion, the presented study opens up a new area of research on high-granularity vulnerability impact assessment. Through our investigation of modern software engineering methodologies, we identified limitations that system operators face in the context of vulnerability handling. We proposed a novel solution, aimed at improving the capabilities of system operators in terms of awareness, actionability, and impact assessment of vulnerabilities in dependencies of the operated software. The evidence presented supports the notion that having access to the transient runtime behavior of vulnerabilities in deployed applications is a novel approach, with a real potential to improve security in developed systems. Surprisingly, the interviewees highlighted that the envisioned tool could be used not just by the system operators, but also by other team members with sufficient technical background, resulting in further democratization of granular vulnerability impact analysis and mitigation. While this study provides a new look into handling vulnerabilities, it also acknowledges certain limitations as its reliance on the trace sampling rate and potential non-exhaustiveness of the trace graph generation procedure. We hope that the results of our study and the released artifacts will contribute to enhancing systems security across the software industry by inspiring further research into innovative vulnerability management strategies.

Bibliography

[1] FASTEN project. URL https://www.fasten-project.eu.

[2] James T. Miller Aaron Bangor, Philip T. Kortum. The system usability scale (sus): an empirical evaluation. *International Journal of Human-Computer Interaction*, 24: 574–, 08 2008. doi: 10.1080/10447310802205776.

[3] James T. Miller Aaron Bangor, Philip T. Kortum. Determining what individual sus scores mean: Adding an adjective rating scale. *J. Usability Stud.*, 4:114–123, 04 2009.

[4] C. Abad, J. Taylor, C. Sengul, W. Yurcik, Y. Zhou, and K. Rowe. Log correlation for intrusion detection: a proof of concept. In *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, pages 255–264, 2003. doi: 10.1109/CSAC.2003.1254330.

[5] Steve Adolph, Wendy Hall, and Philippe Kruchten. Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16:487–513, 08 2011. doi: 10.1007/s10664-010-9152-6.

[6] Steve Adolph, Philippe Kruchten, and Wendy Hall. Reconciling perspectives: A grounded theory of how people manage the process of software development. *Journal of Systems and Software*, 85(6):1269–1286, 2012. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2012.01.059. URL https://www.sciencedirect.com/science/article/pii/S0164121212000386. Special Issue: Agile Development.

[7] Michal Aharon, Gilad Barash, Ira Cohen, and Eli Mordechai. One graph is worth a thousand logs: Uncovering hidden structures in massive system event logs. In Wray Buntine, Marko Grobelnik, Dunja Mladenić, and John Shawe-Taylor, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 227–243, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-04180-8.

[8] Abdullah Al Maruf, Alexander Bakhtin, Tomas Cerny, and Davide Taibi. Using microservice telemetry data for system dynamic analysis. In *2022 IEEE International*

Conference on Service-Oriented System Engineering (SOSE), pages 29–38, 2022. doi: 10.1109/SOSE55356.2022.00010.

[9] Carmelo Ardito, Paolo Buono, Danilo Caivano, Maria Costabile, Rosa Lanzilotti, Anders Bruun, and Jan Stage. Usability evaluation: A survey of software development organizations. pages 282–287, 01 2011.

[10] Brian Barrett. The next wave of Log4J attacks will be brutal, December 2021. URL https://www.wired.com/story/log4j-log4shell-vulnerability-ransomware-second-wave/.

[11] Andre Bento, Jaime Correia, Ricardo Filipe, Filipe Araujo, and Jorge Cardoso. Automated analysis of distributed tracing: Challenges and research directions. *Journal of Grid Computing*, 19(1), February 2021. doi: 10.1007/s10723-021-09551-5. URL https://doi.org/10.1007/s10723-021-09551-5.

[12] John Brooke. Sus: A quick and dirty usability scale. *Usability Eval. Ind.*, 189, 11 1995.

[13] Ian Buchanan. History of DevOps, Feb 2023. URL https://www.atlassian.com/devops/what-is-devops/history-of-devops.

[14] Jeanderson Cândido, Maurício Aniche, and Arie van Deursen. Log-based software monitoring: A systematic mapping study. *PeerJ Computer Science*, 7:1–38, 2021. ISSN 2376-5992. doi: 10.7717/PEERJ-CS.489.

[15] Clément Cassé, Pascal Berthou, Philippe Owezarski, and Sébastien Josset. Using distributed tracing to identify inefficient resources composition in cloud applications. In *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, pages 40–47, 2021. doi: 10.1109/CloudNet53349.2021.9657140.

[16] M Chapman and N Gatti. A model of a service life cycle. In *Proceedings of TINA*, volume 93, pages 205–215, 1993.

[17] George Miranda Charity Majors, Liz Fong-Jones. *Observability Engineering: achieving production excellence*. O'Reilly, 2022.

[18] Thomas Davidson, Emily Wall, and Jonathan Mace. A qualitative interview study of distributed tracing visualisation: A characterisation of challenges and opportunities. *IEEE Transactions on Visualization and Computer Graphics*, pages 1–12, 2023. doi: 10.1109/TVCG.2023.3241596.

[19] Michael Fu and Chakkrit Tantithamthavorn. Linevul: a transformer-based line-level vulnerability prediction. In *Proceedings of the 19th International Conference on Mining Software Repositories*, MSR '22, page 608–620, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393034. doi: 10.1145/3524842.3528452. URL https://doi.org/10.1145/3524842.3528452.

[20] Adrian Furnham. Response bias, social desirability and dissimulation. *Personality and Individual Differences*, 7(3):385–400, 1986. ISSN 0191-8869. doi: https://doi.org/10.1016/0191-8869(86)90014-0. URL https://www.sciencedirect.com/science/article/pii/0191886986900140.

[21] A. Germán Márquez, Ángel Jesús Varela-Vaca, María Teresa Gómez López, José A. Galindo, and David Benavides. Vulnerability impact analysis in software project dependencies based on satisfiability modulo theories (smt). *Computers & Security*, 139:103669, 2024. ISSN 0167-4048. doi: https://doi.org/10.1016/j.cose.2023.103669. URL https://www.sciencedirect.com/science/article/pii/S0167404823005795.

[22] B.G. Glaser and A.L. Strauss. *The Discovery of Grounded Theory: Strategies for Qualitative Research*. Observations (Chicago, Ill.). Aldine, 1967. ISBN 9780202302607. URL https://books.google.nl/books?id=oUxEAQAAIAAJ.

[23] Anja Guzzi, Alberto Bacchelli, Yann Riche, and Arie van Deursen. Supporting developers' coordination in the ide. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, page 518–532, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450329224. doi: 10.1145/2675133.2675177. URL https://doi.org/10.1145/2675133.2675177.

[24] Jan Horsky, Kerry McColgan, Justine E. Pang, Andrea J. Melnikas, Jeffrey A. Linder, Jeffrey L. Schnipper, and Blackford Middleton. Complementary methods of system usability evaluation: Surveys and observations during software design and development cycles. *Journal of Biomedical Informatics*, 43(5):782–790, 2010. ISSN 1532-0464. doi: https://doi.org/10.1016/j.jbi.2010.05.010. URL https://www.sciencedirect.com/science/article/pii/S1532046410000687.

[25] Qiang Li, Jinke Song, Dawei Tan, Haining Wang, and Jiqiang Liu. Pdgraph: A large-scale empirical study on project dependency of security vulnerabilities. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 161–173, 2021. doi: 10.1109/DSN48987.2021.00031.

[26] Dan Lv, Nurbol Luktarhan, and Yiyong Chen. ConAnomaly: Content-based anomaly detection for system logs. *Sensors*, 21(18), 2021. ISSN 1424-8220. doi: 10.3390/s21186125. URL https://www.mdpi.com/1424-8220/21/18/6125.

[27] Emily Mell; Alexander S. Gills Meredith Courtemanche. What is DevOps? the ultimate guide, January 2023. URL https://www.techtarget.com/searchitoperations/definition/DevOps.

[28] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem, 2023.

[29] Andy Molin, Andrei Mario Riviş, and Radu Marinescu. Assessing the real impact of open-source components in software systems. *IEEE Access*, 11:111226–111237, 2023. doi: 10.1109/ACCESS.2023.3322362.

[30] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358231. doi: 10.1145/3239235.3268920. URL https://doi.org/10.1145/3239235.3268920.

[31] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 48(5):1592–1609, 2022. doi: 10.1109/TSE.2020.3025443.

[32] Freddy Paz and Jose Pow-Sang. A systematic mapping review of usability evaluation methods for software development process. 10:165–178, 01 2016. doi: 10.14257/ijseia.2016.10.1.16.

[33] Konrad Ponichtera. Current state of vulnerability handling from the perspective of software operators and developers: semi-structured interviews, June 2024. URL https://doi.org/10.5281/zenodo.11621469.

[34] Konrad Ponichtera. Vulnerability Tracer: instrumentation suite for granular vulnerability trigger detection and impact evaluation, June 2024. URL https://doi.org/10.5281/zenodo.11809584.

[35] Rozilawati Razali and Humairath Abu Bakar. Combining phenomenology and grounded theory in software engineering: An experience. *European Conference on Research Methodology for Business and Management Studies*, 22:145–153, 08 2023. doi: 10.34190/ecrm.22.1.1456.

[36] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, 2018. doi: 10.1109/ICMLA.2018.00120.

[37] J. Sauro. *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices*. Measuring Usability LLC, 2011. ISBN 9781461062707. URL https://books.google.nl/books?id=BL0kKQEACAAJ.

[38] Shlomo Sawilowsky. New effect size rules of thumb. *Journal of Modern Applied Statistical Methods*, 8:597–599, 11 2009. doi: 10.22237/jmasm/1257035100.

[39] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010. URL https://research.google.com/archive/papers/dapper-2010-1.pdf.

[40] Anthony Tam and Behnam Shobiri. Vulnerability management: Best practices for patching CVEs, Aug 2023. URL https://thenewstack.io/vulnerability-management-best-practices-for-patching-cves/.

[41] Laura Viola, Elisabetta Ronchieri, and Claudia Cavallaro. Combining log files and monitoring data to detect anomaly patterns in a data center. *Computers*, 11(8), 2022. ISSN 2073-431X. doi: 10.3390/computers11080117. URL https://www.mdpi.com/2073-431X/11/8/117.

[42] Hieu Dinh Vo. Just-in-time vulnerability detection and localization. *Journal of Computer Science and Cybernetics*, 40(1):79–101, Mar. 2024. doi: 10.15625/1813-9663/19102. URL https://vjs.ac.vn/index.php/jcc/article/view/19102.

[43] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. An empirical study of usages, updates and risks of third-party libraries in Java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 35–45, 2020. doi: 10.1109/ICSME46990.2020.00014.

[44] Adam Wiggins. The Twelve-Factor App, 2017. URL https://12factor.net/.

[45] Ayesha Zainudeen. Testing ChatGPT as a tool for text data analysis: Is ChatGPT going to take my job, or make it a lot easier going forward?, Apr 2023. URL https://lirneasia.net/2023/04/testing-chatgpt-as-a-tool-for-text-data-analysis-is-chatgpt-going-to-take-my-job/.

[46] Andres Zambrano, Xiner Liu, Amanda Barany, Ryan Baker, Juhan Kim, and Nidhi Nasiar. *From nCoder to ChatGPT: From Automated Coding to Refining Human Coding*, pages 470–485. 10 2023. ISBN 978-3-031-47013-4. doi: 10.1007/978-3-031-47014-1_32.

[47] Zoltán Zvara, Péter G.N. Szabó, Barnabás Balázs, and András Benczúr. Optimizing distributed data stream processing by tracing. *Future Generation Computer Systems*, 90:578–591, 2019. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2018.06.047. URL https://www.sciencedirect.com/science/article/pii/S0167739X17325141.