ConfigWiz: Automating Privilege Configuration for Containerized Applications

Mohammad Kavousi[1][0000−0002−3378−2315], Hui Xue[2], Yan Chen[1][0000−0003−4103−1498], Xin Chen[2], Yunlong Xing[3][0000−0002−3844−2467], Kun Sun[3][0000−0003−4152−2107], Therese Schachner[1], and Zhiheng Tao[1]

[1] Northwestern University, Evanston, IL, USA
kavousi@u.northwestern.edu, ychen@northwestern.edu, thereseschachner2022@u.northwestern.edu, zhihengtao2023@u.northwestern.edu
[2] Zhejiang Lab, Hangzhou, China
hxue@zhejianglab.com, bage.cx@zhejianglab.com
[3] George Mason University, Fairfax, VA, USA
yxing4@gmu.edu, ksun3@gmu.edu

Abstract. Cloud-native applications have seen significant growth over the past decade due to their crucial role in bridging software development, deployment, and operations. The rapid pace of the container-based software supply chain, however, has introduced considerable security challenges. While Linux provides adequate privilege mechanisms that integrate with popular container runtimes like Docker, poor security practices such as granting excessive privileges to third-party containers remain prevalent. This issue persists largely because deriving appropriate privilege configurations for third-party container images is still an ad hoc process. To assign the correct privileges to a container, DevOps teams require precise knowledge of both the underlying system and the behavior of the containerized application.

In this paper, we present ConfigWiz, a practical tool designed to automate the privilege configuration process for containerized applications, with a focus on the Docker platform. ConfigWiz establishes a detailed mapping between system calls (syscalls) and capabilities, incorporating accuracy enhancements through analysis of the Linux kernel source code. To address privilege requirements for auxiliary operations not required by the image itself, ConfigWiz leverages Large Language Models (LLMs) to analyze discussions from online forums, providing a flexible and secure privilege configuration process.

Keywords: Container Security · Cloud Networking · Large Language Models · Docker · Operating systems

1 Introduction

Cloud-native applications have experienced steady growth over the past decade, with containers becoming a primary deployment unit due to their well-defined

life cycles (deployment, scaling, updating) and integration with tools such as Kubernetes. This ecosystem enables a fast-paced software supply chain where third-party images can be pulled and deployed with simple commands such as `"docker pull"` [9, 10].

Despite maintainers' quality control, running third-party images introduces significant risk. Reports show rising attacks and growing operational impact in cloud-native environments [5, 37]. Since containers share the host OS kernel, misconfigurations can amplify the consequences of kernel-level vulnerabilities [40, 23, 32]. Confining container behaviors crafted by unknown parties has thus become a critical yet challenging demand.

To correctly secure a freshly downloaded container, a DevOps team must (i) understand what the image does and (ii) be well versed in host OS security mechanisms to apply appropriate configurations. Both are difficult: image behavior is often under-documented, corner cases are common, and supply-chain compromise is a realistic threat. Recent empirical work confirms this usability challenge: a 2025 study found that developers struggled to determine which syscalls were used where when crafting policies, motivating automation for least-privilege mechanisms [16].

Problem Statement: Configuring container privileges often requires an in-depth understanding of both the container image's internals and the operating system's security mechanisms. Can this process be effectively automated to ensure accurate and secure privilege assignments?

Researchers have proposed mechanisms to improve container security against a variety of attacks [17, 38]. Lin et al. [29] studied 27 vulnerabilities that lead to escalation of privileges in container systems and found that 89% of these attacks could have been prevented with proper security measures at container launch. Several tools and best practices, such as `docker-bench` [6] and Matetti et al. [34], provide general hardening guidance, but they do not tailor privilege configurations to an individual image and user need.

Linux capabilities enable structured privilege management by grouping privileged operations under named capabilities and integrating with Docker via `-cap-add` and `-cap-drop`. As of kernel version 6.12, Linux has more than 300 syscalls but only 41 capabilities, simplifying privilege management compared to syscall-level policies. Docker applies a default set of 14 capabilities, which is convenient but often either excessive for simple workloads or insufficient for images with specialized requirements. In practice, users frequently misconfigure containers: some add ad-hoc capabilities until the image runs, while others resort to the `privileged` flag, which grants all capabilities and disables most isolation boundaries. These defaults and flags leave a wide gap between security and functionality. ConfigWiz addresses this gap by deriving container-specific least-privilege capability sets [24]. This urgency is reinforced by recent attacks such as the 2023 OverlayFS vulnerabilities ("GameOver(lay)"), where permissive settings enabled privilege escalation [45, 21].

To enable automation, it is essential to understand the mapping between system calls and capabilities, since this relationship dictates which privileges

are actually needed. Conceptually, this mapping should be documented within the Linux kernel; however, it is often unclear or incomplete. For example, the `setns` syscall should map to `SYS_PTRACE`, yet this relationship is not explicitly documented in either the syscall or capability descriptions. Moreover, a single capability (e.g., `SYS_PTRACE`) may gate multiple syscalls, and kernel conditions determine when the absence of a capability causes failure. This lack of clarity complicates correct capability configuration for containerized applications.

In addition to capability selection, correctly running a container often requires runtime environment settings (e.g., volume mounts or network parameters) determined before launch. While image analysis can infer many requirements, auxiliary needs arise ad hoc and are commonly discussed in online forums. Automating auxiliary support is thus essential to achieve practical least-privilege configuration.

In conclusion, the challenges addressed in this work can be summarized as follows:

- Misconfiguration of container isolation mechanisms is prevalent in practice.
- Establishing a clear connection between system configuration and application images requires understanding application internals that typical users may lack.
- Linux documentation and community guidance impose high user burden and latency for correct capability configuration.

To address these challenges, our solution uses a divide-and-conquer approach. First, we extract and refine the syscall–capability mapping by compiling Linux kernel code into an Intermediate Representation (IR) and analyzing mapping entries to determine which capabilities are necessary for syscall functionality. We then derive the required capabilities for a specific container image by analyzing its application binaries.

Second, to address auxiliary execution support, we leverage Large Language Models (LLMs) to interpret user specifications and recommend capability configurations when requirements are not implied by the image. We use prompt engineering and fine-tuning with documentation and filtered real-world forum data to improve reliability.

To the best of our knowledge, ConfigWiz is the first system to automatically generate capability configurations for container images by jointly considering both application-derived requirements and user-specified auxiliary requirements.

This paper also makes the following contributions:

- We analyze user-posted questions on popular forums (e.g., `stackoverflow.com`) to characterize misconfigurations and the prevalence of overprivileged container configurations suggested in community answers.
- We establish and refine a syscall–capability mapping via static analysis of the Linux kernel and documentation, enabling fine-grained capability configurations for container images.
- We apply prompt engineering and fine-tuning to improve LLM-based auxiliary support for interpreting user requirements and recommending capability configurations for user-requested functionality.

In summary, we developed ConfigWiz to automate privilege configuration for containerized applications. Our results show that for the 150 images we analyzed, ConfigWiz generates correct capabilities without overprivileging for more than 93% of the images based on static analysis. Furthermore, for 227 user posts (i.e., given a text prompt), ConfigWiz achieves over 86% accuracy.

In Section 2 we talk about the motivations of this study and give examples of attacks. Section 3 gives a brief overview of the entire system. In Section 4 we explain our approach to analyzing application images. Section 5 describes the design of a system that meets users' needs from a natural language perspective. Finally, Sections 6 and 7 cover the implementation details and present the evaluation results.

2 Measurement of Container Capability Misconfigurations

We begin by discussing the motivations behind this work in Section 2.1. In Section 2.2, we provide a concise background on the security mechanisms used to isolate containers. Following that, we illustrate several possible attacks and explain how systems can become compromised without proper configuration.

2.1 Motivation

We observed a growing tendency to overprivilege containers to ensure their successful execution. This trend was evident in numerous forum questions and responses across popular platforms such as `Stackoverflow.com`, which often reflect significant technical practices that are mistakenly adopted as best practices.

Our analysis began by searching top posts sorted by relevance using the queries mentioned in Table 1 on `Stackoverflow.com`. In our review of approximately 300 posts, we found that 55% included misconfigurations. Industry-wide surveys corroborate this trend: a 2024 benchmark found that only 21% of organizations consistently disable insecure default Linux capabilities, implying that the vast majority still run containers with unnecessary privileges [37]. Taken together, our forum-derived observations and independent industry telemetry indicate that ad hoc privilege tuning remains common in practice rather than an outlier of community Q&A. We therefore treat forum posts as a reasonable proxy for practitioner behavior when characterizing overprivilege and designing our measurements [37]. These misconfigurations were either introduced by the user in the questions or suggested as overprivileged configurations in the replies—whether or not these replies received upvotes. We also included single answers that presented multiple approaches, one of which was overprivileged. Our motivation is rooted in the observation that users may bypass reading the documentation on proper security mechanisms, instead opting for overprivileged commands like `"docker run --privileged <args>"` to achieve the desired functionality for their containers [48].

Table 1. Summary of Forum Posts on Stackoverflow

Mechanism	Capabilities	Seccomp	AppArmor
Query Used	Docker cap is:answer	Docker Seccomp is:answer	Docker AppArmor is:answer
Number of Posts Analyzed	100	100	70
Percentage of Misconfigurations	55	70	50

Table 1 summarizes our analysis based on the previously defined criteria for security misconfigurations, demonstrating that numerous posts on these forums can indeed be classified as misconfigurations, potentially posing risks to the system.

ConfigWiz utilizes the container image along with related user functionality provided as text. Our objective is to achieve a secure configuration that minimizes the attack surface while preserving normal runtime functionalities.

2.2 Mechanisms to Restrict Container Privileges

Previous studies [17] have compiled a list of mechanisms for restricting container privileges: Mandatory Access Control (MAC), Control Groups (Cgroups), Namespaces, Seccomp, Capabilities, and Container Linking.

To configure the above mechanisms for each application, we need to determine which ones can be mapped to understandable operations. Our analysis shows that only capabilities and seccomp configurations are directly related to specific application functionalities. In contrast, MAC systems like AppArmor function more like rule-based firewalls, focusing less on high-level operations and more on restricting access to paths on the host. These modules are typically configured based on access patterns observed at runtime, as demonstrated by existing work [31, 34].

Capabilities provide a user-friendly way to configure low-level Linux privileges, thereby confining runtime behaviors. While it is possible to limit the attack surface by disallowing certain syscalls [20, 25], we emphasize the importance of configuring capabilities to restrict operations of syscalls that might perform privileged actions. This dual approach is crucial to prevent the execution of privileged operations. Previous research has demonstrated how each capability can be misused, including those enabled by Docker by default [1, 2].

Given the extensive number of syscalls that a user must configure, many opt for capability configuration because it offers a higher-level representation of user requirements. Forum posts show that capability configuration is discussed about 2.5 times more frequently than syscalls. Therefore, it is essential to effectively configure capabilities for containers in a way that ensures full application functionality and meets user requirements while protecting the container from overprivileged capabilities. Additionally, we argue that it is possible to infer the required capabilities from available documentation and community resources.

2.3 Measurement Results and Risk Analysis

In this section, we study major security risks introduced by problematic configuration.

- **Overprivilege:** While reviewing Stackoverflow posts, we observed common patterns where users granted excessive privileges to containers. This typically stems from (i) default configurations that lack needed functionality, and (ii) limited understanding of container internals and Linux capabilities, making broad privileges easier than identifying precise requirements. We illustrate this with two examples:
  - `--privileged`**:** Running a privileged container is equivalent to having full root access on the host system [13]. An attacker with access to the container can potentially mount the host disk (e.g., `"mount /dev/sda1 / mnt/host_folder"`) and gain arbitrary file access [4]. The 2023 OverlayFS vulnerabilities ("GameOver(lay)", CVE-2023-2640 and CVE-2023-32629) demonstrate that permissive configurations can enable privilege escalation in practice [45, 21].
  - **Coverage gaps between isolation techniques:** There is a semantic gap between high-level functionalities and low-level operations, e.g., 41 capabilities versus hundreds of syscalls. For example, users are often advised to enable `CAP_SYS_ADMIN` to use `sethostname` [8], but this capability also permits unrelated powerful operations (e.g., `sys_mount`) that can be abused for privilege escalation [3].
- **Risk of Side Effects:** Docker features can amplify the risks of excessive capabilities. For instance, bind mounts allow containers to access host directories; when combined with capabilities such as `CAP_CHOWN` or `CAP_DAC_OVERRIDE`, a containerized process can reassign file ownership or bypass permission checks to modify protected host files. Similarly, `CAP_SETUID`, `CAP_SETGID`, and `CAP_SYS_ADMIN` can further expand what a container can do with mounted directories, turning ordinary features into serious security vulnerabilities.

3 System Overview

To automate Linux capability-based privilege configuration, our approach combines static and runtime analysis to thoroughly examine container images. Additionally, we leverage LLMs to identify the necessary capabilities specified by users when a functionality beyond the original design of the image is required. ConfigWiz is intended for use by developers and DevSecOps engineers in controlled environments such as CI/CD pipelines, where images are scanned prior to deployment. We focus on mitigating attacks originating from within the container by ensuring minimal privileges, and assume the analysis process itself is applied to images that are either vetted or sandboxed to prevent hostile behavior during extraction.

We divide the operation of a container into two modes:

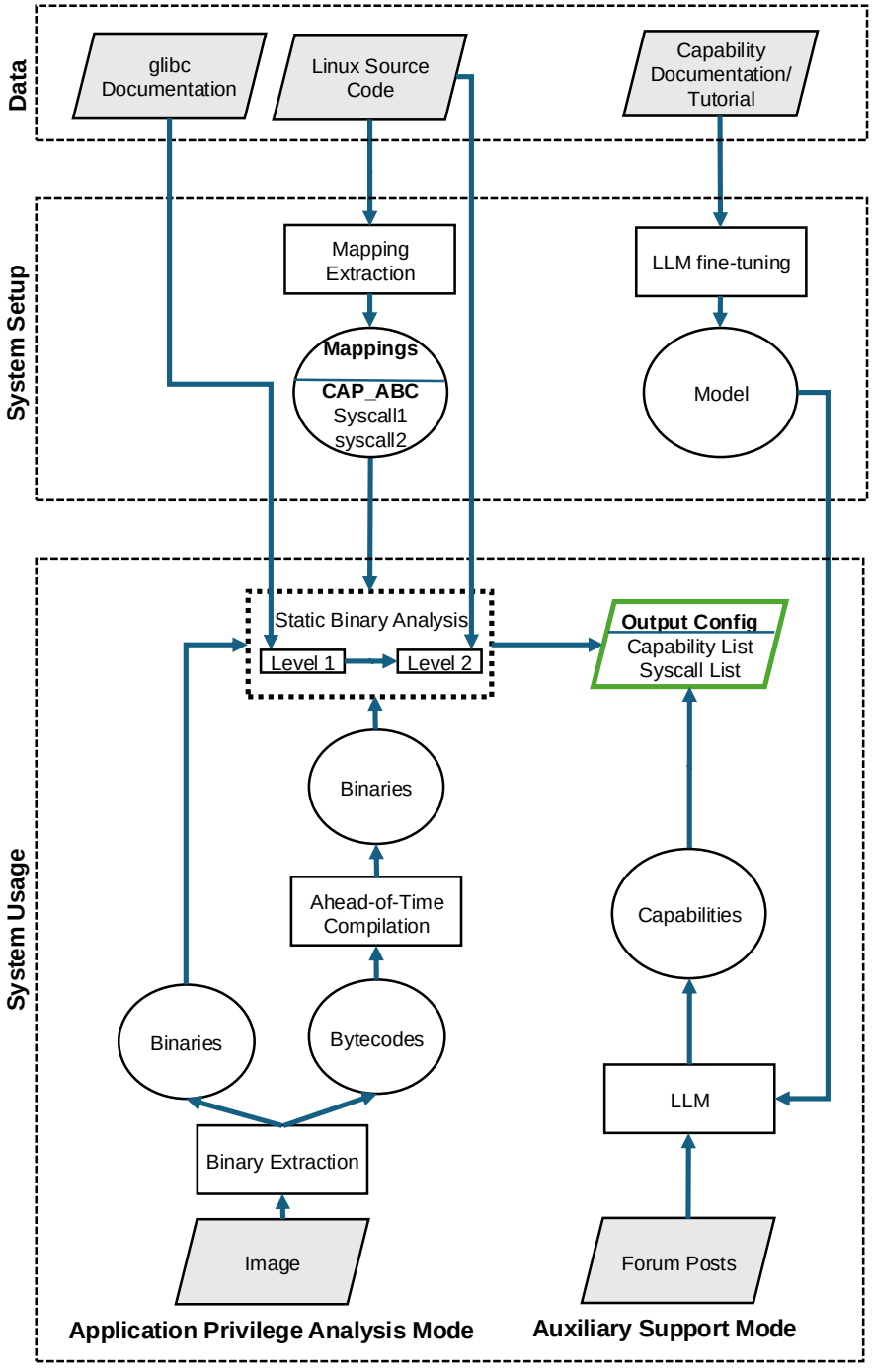


Fig. 1. Overall Architecture of ConfigWiz

1. **Application Image Privilege Analysis Mode:** This mode focuses on analyzing the container image to locate relevant binaries and derive the necessary capabilities for these binaries to function as intended.
2. **Auxiliary Support Mode:** The user may require additional privileges to perform a task outside of the main application functionality. For example, a user running an `httpd` web server might need to modify the host network configuration, a task not required by the web server by default. This mode addresses such needs by providing the additional required capabilities for the user to perform the desired functionality.

Figure 1 illustrates the architecture of ConfigWiz from a workflow perspective. We begin by incorporating the data required for both modes. In Section 4.1, we begin by using the Linux and glibc source code and converting them to their Intermediate Representation (IR) to identify relationships between capabilities, syscalls, and libraries. In the usage mode, we retrieve the Docker image and extract the binaries required for analysis (Section 4.2). We then refine the aforementioned relationships and their usages in Section 4.3 and generate the required configuration.

For functionalities that extend beyond the core application, we incorporate user specifications. We then leverage LLMs to configure the image according to the user's requirements (Section 5).

4 Application Privilege Analysis

In this section, we explain the static and runtime analysis techniques applied to a given container image and the analysis of Linux kernel source code to derive proper privilege configurations.

4.1 Linux Code Analysis

Since the enforcement of capabilities occurs within the Linux kernel, accurate configuration requires a precise analysis of where capabilities are implemented in the kernel.

Containers exhibit their behavior by making requests to the kernel through syscalls, which are typically invoked within standard library implementations, such as glibc.

Capabilities define the privileges necessary for performing specific actions, and they can be used as launch-time parameters to designate container privileges. Understanding which syscall requires which capability reduces the challenge of capability-based privilege configuration for a given Linux binary to identifying the syscalls that the binary actually uses.

To achieve this, we analyze the Linux kernel source code to extract the mapping between syscalls and capabilities. For glibc, we analyze the function calls that lead to syscall invocation. We compile the Linux and glibc source codes, extract their intermediate representation (IR), and trace the calling relationships between glibc functions to syscalls, and from syscalls to capabilities.

This methodology has been adopted by several previous works [25, 20, 26] to extract the glibc to syscall mapping. We also reuse this method to obtain the syscall to capability mapping. These mappings are kernel-version dependent, and ConfigWiz regenerates them when upstream kernels evolve. Automating this refresh ensures that capability assignments remain aligned with changes in kernel interfaces and checks.

Our main observation is that the syscall-to-capability mapping alone is insufficient, since many mappings apply only in rare cases and require deeper analysis to determine when a capability is actually enforced. We refer to some mappings as "rare" when, although a syscall is associated with a capability check, the capability is only required for non-common functionality rather than the syscall's typical use. For example, `CAP_IPC_LOCK` is mapped to a small set of syscalls, but in many cases this mapping is related to exceeding the `mlock` limit, which is uncommon in typical application functionality. We identify such cases through a systematic review of kernel documentation, source-code comments, and observed usage frequency, rather than ad hoc judgment, and treat them conservatively during configuration.

Figure 2 shows the process of using the source codes of the Linux kernel and glibc. As mentioned above, the process of extracting syscall to capability mappings will go through an additional step of code analysis to extract and omit the rare mappings. The mapping extraction will result in two mappings: from glibc to syscalls, and from syscalls to capabilities. These mappings will be further used in our analysis.

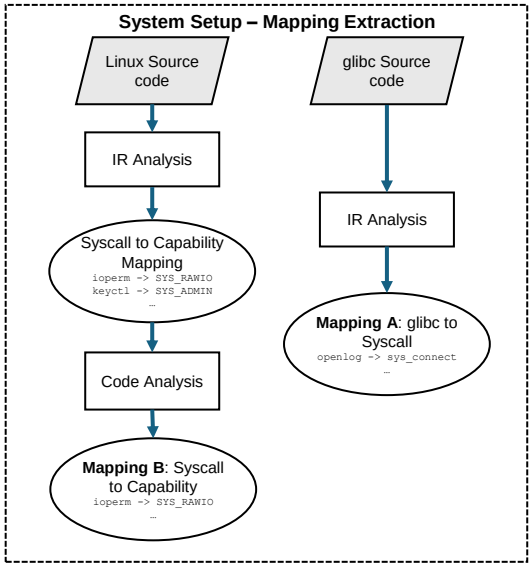


Fig. 2. Mapping Extraction and Refinement

4.2 Runtime Analysis for Binary Extraction

Runtime analysis is crucial for pinpointing the specific binaries involved when executing a container image. This step is necessary because many of the binaries packaged within the container image, such as standard operating system tools, do not play a role in the container's primary functionality. While these binaries are present as part of the base image, they are not utilized in executing the container's intended operations and, therefore, do not require analysis for capability extraction.

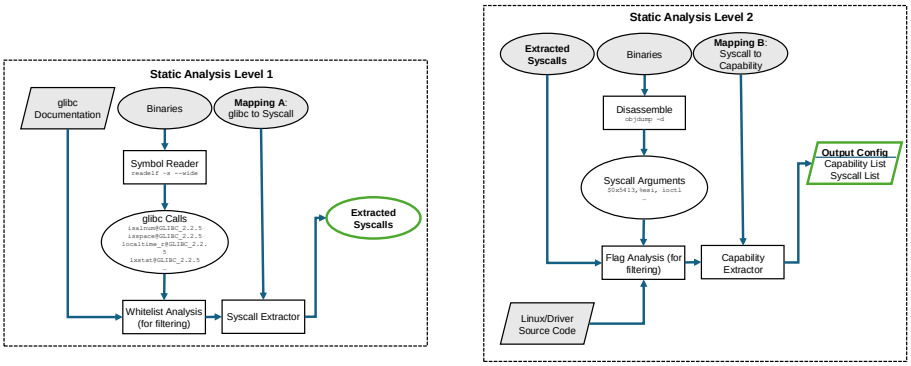
Our focus is on identifying and analyzing only the binaries necessary for running the container's applications, such as the "entrypoint" binaries and their dependencies. By narrowing our scope in this manner, we avoid assigning unnecessary privileges to binaries that remain unused during the application's runtime. To achieve this, we initiate the container image, verify that it runs without errors, and extract only those binaries that are actively engaged in fulfilling the application's functionality. This approach ensures a streamlined and secure privilege assignment process, excluding any additional or irrelevant executables. We note that uncommon runtime paths or optional features not exercised during startup may evade detection; extending ConfigWiz with fuzzing or AI-generated workload simulation is a natural direction to improve coverage.

Language Compatibility for Binary Extraction: ELF (Executable and Linkable Format) is the standard binary format executed by the Linux kernel. While most compiled languages (e.g., C/C++) naturally produce ELF binaries, container images may include components written in languages such as Java. To enable uniform analysis, ConfigWiz employs Ahead-Of-Time (AoT) compilation with GraalVM [12], which translates Java bytecode and dependencies into a single ELF executable. This allows Java workloads to be analyzed alongside native binaries while also improving startup time and memory use. Although GraalVM supports additional languages (e.g., Python, JavaScript, Ruby, R), some features such as reflection or dynamic class loading require special handling, and other environments (e.g., Go, Rust, JIT workloads) remain outside our current scope but can be addressed in future extensions.

4.3 Static Binary Analysis

After extracting the mappings we perform two passes to perform static analysis:

– **Level 1:** The first pass focuses on extracting syscalls from the extracted binaries. Binaries require syscalls to interact with the operating system and perform actions like I/O, memory management, and process control, which are often subject to access limitations for security and stability. There are several ways for binaries to make syscalls. In this work, we focus on the implementation provided by glibc, the standard C library commonly used in ELF binaries. Other methods for invoking syscalls such as using alternative libraries, Go's runtime, Just-In-Time (JIT) compiled code, Rust's standard library, or direct syscall invocations, are deferred for future exploration. These



(a) Syscall Extraction of Binaries (b) Final Capability/Syscall Extraction

Fig. 3. Static analysis pipeline. (a) Level 1 extracts syscalls from binaries. (b) Level 2 derives the final capability and syscall configuration.

alternatives can also be evaluated using the methodology established in this work.

Figure 3(a) shows the process of extracting syscalls from binaries. We extract an allow-list from glibc functions used by the binaries. The allow-list enables us to check whether the function is using a privileged functionality of the syscall. For example, the `getrlimit` library call maps to `prlimit64` syscall. According to the documentation, `getrlimit` is used solely to retrieve the current limits for various resources [7]. Although it is associated with a seemingly privileged syscall, which maps to the `CAP_SYS_RESOURCE` capability, it does not typically involve privileged functionality that would affect the system in standard scenarios.

We create the allow-list by analyzing the documentation of glibc functions. Given that there are 13,634 mapping entries between glibc functions and syscalls, it is impractical to perform code analysis for each entry as done in the previous step. Therefore, we focus on constructing an allow-list directly at the glibc-function level. Out of the 1,702 functions that map to a syscall, we review their documentation and add the functions whose typical behavior does not require privileged functionality to the allow-list.

We then consult the glibc to syscall mapping extracted from Section 4.1 to infer the final set of syscalls used in the set of image binaries.

– **Level 2:** This pass will now extract the required capabilities for the image. The observation is that in some cases, we must decide whether to include certain capabilities based on the flags passed to a syscall. For example, in the Linux source code, the `CAP_NET_ADMIN` capability is required if the `SO_DEBUG` flag is passed to the `setsockopt` syscall. Extracting the constant value of this flag is straightforward since it is defined in the Linux kernel. We extract the hex value of the flag, identify which syscall it corresponds to, and determine whether a capability is required when using this flag. We then disassemble the

binary to check which parameter is being passed to the syscall, enabling us to decide whether to provide the necessary capability.
Flag analysis is also useful for syscalls that handle functionality differently based on the flags passed to them. A typical example is the `ioctl` syscall, commonly used by device drivers to expose services to user-space applications. This syscall can take function pointers as parameters, making it difficult to extract all required capabilities from the IR code alone. For instance, the `tun` driver provides the `TUNSETIFF` operation, which configures the interface name and other parameters, and this operation goes through the `ioctl` syscall. Network configuration operations like this can affect the entire system's behavior, necessitating the `CAP_NET_ADMIN` capability. We identify the constant value of `TUNSETIFF` and determine whether it is passed to `ioctl` via the `$esi` register, which is used to pass the second parameter to a function in x86_64 architecture. Additionally, we can extract information from forum posts where users have encountered issues running their images without the required capability. We incorporate the results of this flag analysis into our static analysis to improve the accuracy of capability assignment.
Figure 3(b) illustrates the final process for extracting the required capability set using the syscall to capability mapping extracted in Section 4.1. Starting with the syscalls identified at the previous level, we disassemble each binary to analyze the arguments passed to the syscalls. By consulting the Linux kernel or driver source code, we determine whether a specific capability is required for each syscall identified from the binaries.

5 Large Language Models for Auxiliary Support

Not all privilege requirements can be derived directly from the container image. Users may need functionalities beyond what the image is intended to support. For instance, a user might require a `MySQL` image to be able to modify firewall rules, a functionality typically not required by the database server itself. To ensure a fully functional system, we rely on user specifications to address these uncovered cases. We turn to forum posts where users request specific functionalities to run their containers.

5.1 Design Overview

Given the distinct vocabulary associated with Linux capabilities, we apply LLMs to suggest privileges based on user requirements. However, directly asking questions without context often produces verbose or imprecise answers that fail to solve the user's immediate problem. Users expect concise and actionable results, while ConfigWiz requires a concrete capability output for automation.

To address this, we employ two complementary methods:

– **Method I: Prompt Engineering:** We craft structured prompts that frame a query as a Docker/Linux capability issue. The model's response is then parsed to extract the predicted capability, yielding a direct answer.

– **Method II: Prompt Engineering + Fine-tuning:** Building on Method I, we fine-tune an LLM on a curated dataset of capability-related documentation and filtered forum posts, and evaluate it on a held-out test set.

We use prompt engineering and supervised fine-tuning because our task, predicting Linux capabilities from textual problem descriptions, is a semantic classification problem where these techniques are effective and far more practical than training a model from scratch.

5.2 Extraction and Filtering of Forum Posts

In addition to trusted sources from documentations and tutorials, our dataset comprises posts from community sources. Specifically, we use posts from Stack Overflow, Stack Exchange, and ServerFault, where the answers mention a specific Linux capability. We retrieve these posts using query sets such as `"SYS_XXX is:answer"`, `"CAP_XXX_XXX is:answer"`, or `<capability> is:answer`, specifically targeting posts related to Docker issues. The capability mentioned in the answer of each forum post is used as the label for that sample.

Each post collected is evaluated against *exclusion criteria* to ensure that only trusted and accurate community answers are included. We exclude posts: (i) when no decisive answer is available (e.g., the relevant answer is ranked in the bottom half and not accepted, or the question has only one answer with a score below 1); (ii) when answers mention multiple capabilities instead of the capability of interest, which prevents isolating keywords unique to a single capability; and (iii) when the question itself already specifies the capability, because in such cases the issue is not missing privileges but something unrelated.

Our framework can be extended to incorporate additional sources (e.g., CIS guidance or Docker security references) as needed.

5.3 Prompt Engineering

To steer the LLM toward precise predictions, we craft a structured prompt that explicitly states the question concerns Linux capability configuration in a container. The system message specifies that the task is capability prediction, and the user message provides the post title and body along with a short instruction to avoid enabling all capabilities by selecting only what is required. We then parse the response to extract a capability label.

Prompting alone is not sufficient: simply mentioning "Linux capability" does not reliably anchor the model to the fixed set of capabilities, and the model may output long troubleshooting advice rather than a single capability name. Consequently, we treat prompt engineering as a strong baseline and build on it with supervised fine-tuning.

5.4 Creating a Fine-Tuned Model

To create the fine-tuned model, we divide our dataset into a fine-tuning set and an evaluation set, with most samples reserved for testing to support a compre-

hensive evaluation. We include capability documentation in the fine-tuning set and randomly select filtered forum samples to cover realistic, noisy descriptions.

We fine-tune the model using the LLM provider's API with instruction-style training examples. We keep the system instruction constant (e.g., *"You are an assistant that predicts Linux capabilities."*), use the post title+body as the user input, and use the capability label derived from the selected forum answer (e.g., `CAP_NET_ADMIN`) as the target output. During evaluation, we provide only the system+user messages and measure whether the model predicts the correct capability on the held-out test set.

6 Implementation

6.1 IR Analysis and Mapping Extraction

The glibc to syscall mapping has been extensively studied in previous works such as Confine [25], Chestnut [20], and Temporal [26]. We reuse the general IR-based call-tracing methodology from this line of work for glibc-to-syscall extraction, and extend it to construct a syscall-to-capability mapping. In particular, our analysis identifies capability-checking functions, extracts their capability parameters, and traces the relevant call paths that connect these checks to syscall handlers. The construction of this mapping involves the following steps:

To establish the mapping between syscalls and capabilities, we first compile the Linux kernel into bytecode, which is subsequently disassembled into LLVM Intermediate Representation (IR) code. This process generates approximately 4 million lines of IR code. We analyze the IR code to identify capability-checking functions and their associated parameters. Specifically, we focus on functions that accept `%cap` as a parameter. We recognize 34 capability-checking functions, such as `capable`, `ns_capable`, and `avc_has_perm_noaudit`. Each of these functions is examined to identify the specific capability parameters, which are then marked for further analysis. Next, we format the IR code by tracing the function calls, retaining only the relevant function names and their associated parameters. This step simplifies the analysis and allows us to automatically construct the syscall to capability mapping from the formatted IR.

Following this, we conduct a rareness analysis as outlined in Section 4.1, refining the mapping to account for uncommon cases. This refined mapping is applied when a non-whitelisted glibc function is encountered during binary analysis. The corresponding glibc function is then analyzed using the refined mapping, and the required capability is extracted based on the associated syscall.

6.2 Large Language Models for Extracting Capabilities

To build the language model, we gather and filter the data according to the queries and criteria outlined in Section 5.2. Using Python's BeautifulSoup [11], we extract the posts' titles and bodies and apply the exclusion criteria to ensure relevance and quality. From these filtered posts, we implement prompt engineering by passing the format `"<title>, <question body>, <prompt>"` to guide

the model toward identifying the capability configuration issue and producing a capability label. For a baseline, we also query the model using only `"<title>, <question body>"`.

For fine-tuning, we curate the extracted content into a `.jsonl` file in the format expected by the LLM's API. Each training instance includes a constant instruction specifying the task (capability prediction), the post title and body as input, and the expected capability label (e.g., `CAP_NET_ADMIN`) as the target output. During testing, we provide only the instruction and input, allowing the model to predict the capability without being explicitly shown the answer.

We fine-tune a GPT-4 family model (GPT-4o) for our experiments. For evaluating the final fine-tuned and prompt-engineered design, we also utilize another fine-tunable LLM, Cohere's Command-R, with our baseline model being `command-r-20.1.0`. The goal of this work is not to directly compare different LLMs, but to evaluate the effectiveness of fine-tuned LLMs for capability prediction.

Our implementation introduces modest overhead: static analysis runs once per image and typically completes within minutes, and all steps can be parallelized across images. This makes ConfigWiz suitable for integration in CI/CD pipelines.

7 Evaluation

In this section, we present our evaluation methodology and results. First, we assess the accuracy of the extracted mappings. Next, we measure the accuracy of the image analysis mode. Finally, we evaluate the accuracy of deriving capabilities from user descriptions.

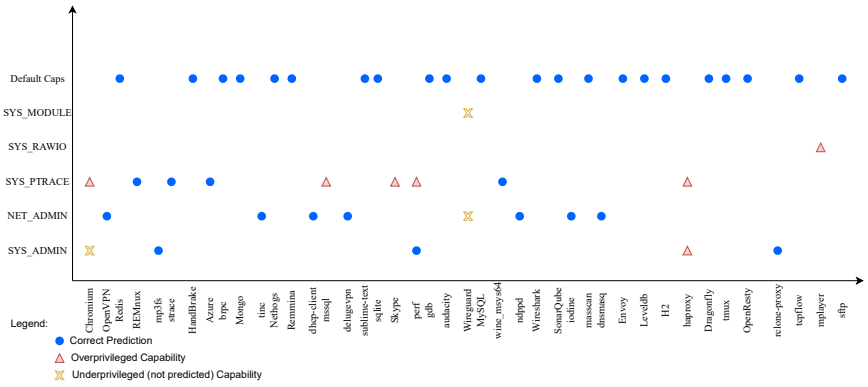


Fig. 4. Capability Configuration Results

7.1 Evaluation Methodology

After setting up the system, we proceed to demonstrate the process of extracting and evaluating results. We initially assess the system's performance using a specified image and further evaluate the natural language model based on user input.

The evaluation metrics encompass accuracy, ensuring that the system provides only the necessary capabilities required for operation, and the overhead associated with performing these tasks. We note that our dataset reflects publicly available container images and forum discussions, which capture common troubleshooting and deployment patterns. Enterprise-specific or proprietary deployments may be underrepresented, so our evaluation should be viewed as characterizing prevalent practices rather than the entire deployment landscape.

Image Privilege Analysis We evaluate ConfigWiz on two classes of container images. First, we search Docker Hub using the query `site:hub.docker.com "--cap-add"` to identify images that explicitly request additional capabilities (approximately 1800 results). Second, we select widely used images that do not advertise extra capabilities, sourced from Docker Hub and popular GitHub projects that include Docker build files (e.g., GitHub Ranking [14] and GitHub Trending [15]).

For each image, we focus on the containerized application rather than all binaries in the base operating system. We apply the runtime binary extraction pipeline of Section 4.2 to run a simple workload under Linux Audit and identify application-specific binaries; when the application is packaged as a `.jar`, we use GraalVM to obtain an ELF binary. We then use the static analysis in Section 4.3 to derive the required capabilities by mapping glibc calls to syscalls and capabilities, including flag analysis for privileged operations. The resulting capability set is finally compared against the configuration specified in the image documentation or `--cap-add` flags to assess whether the documented configuration is correct, overprivileged, or underprivileged.

Auxiliary Support To evaluate the LLM system, we first apply the exclusion criteria described in Section 5.2 to construct a filtered evaluation set with clearer and more targeted labels. We then compare the model predictions against the capability labels derived from the selected forum answers to assess agreement on these filtered samples. We conduct three types of analysis for comparison:

- **GPT:** In this method, we use GPT to provide an answer based solely on the question title and body. This analysis demonstrates the baseline performance and highlights how prompt engineering and fine-tuning can enhance the system's effectiveness compared to conventional interactions with GPT.
- **Prompt Engineering + GPT:** In this approach, we append a crafted prompt that includes a capability requirement at the end of the question title and body. This method tests the impact of structured prompts on improving response accuracy.

– **Prompt Engineering + Fine-tuning + GPT/Command-R:** This method involves dividing the dataset into two parts: one for training the model and the other for testing. After building the fine-tuned model using the training set via the LLM's API, we apply the test set to compare the results with those found in forum posts, assessing the improvements achieved through fine-tuning.

7.2 Results

Application Privilege Analysis Our analysis of the container images, some of which require additional privileges while others do not, shows an accuracy rate of 93.3% in assigning the correct capabilities without overprivileging. Among the 150 tested images, only two were underprivileged, and nine were overprivileged. In one case, both underprivileging and overprivileging were observed simultaneously. Figure 4 presents the outcomes of our analysis, highlighting the instances where accurate configurations were achieved, as well as the cases that require further improvement. On average, the capability configuration process takes 51.2 milliseconds per application on a typical desktop machine.

For some capabilities, we argue that they might not be necessary in any system. The `CAP_SYS_NICE` and `CAP_SYS_RESOURCE` capabilities were the most frequently occurring in our output sets. These capabilities adjust process priority and scheduling policies or increase system resource limits for a process. While these capabilities appear as requirements in some images based on static analysis, they are not actual requirements from the application itself, as the image can run effectively without them.

Error Analysis: The reason for underprivilege is typically due to either not identifying the correct binaries or the image relying on existing Linux tools to perform the operation, where the binary may have been classified as a "non-application-specific" binary. Overprivileging occurs when there is code within the program that suggests the need for additional capabilities, but this requirement is not explicitly documented by the program developer in the image documentation. To further improve accuracy, we should actively refine our heuristic collection methods and conduct deeper flag analysis, which will enhance the system's ability to precisely determine the necessary capabilities. Despite these challenges, our methodology remains robust, and the system consistently demonstrates high accuracy in capability assignment.

Large Language Model We evaluated 481 posts from real-world forum discussions. Initially, we applied the exclusion criteria mentioned in Section 5.2, which resulted in a reduction to 227 forum posts. We then conducted three types of analysis, as described in Section 7.1.

– **No-Knowledge GPT:** We evaluated the standard GPT model without any optimization to determine its ability to respond accurately. The model achieved an accuracy of only 16.9%, indicating significant limitations in answering the evaluated questions. This suggests that, despite being trained on large-scale

internet data, GPT struggles to effectively respond to specific queries, even when they might pertain to information that has been widely available online.

- **Prompt Engineering:** By crafting the prompt and emphasizing a capability issue, the results improved. Without fine-tuning and using only prompt engineering, we achieved an average accuracy of 19.2% with a generic prompt such as *"I have a Docker problem: <title, question body>"*. This accuracy increased to 60.65% when the query explicitly highlighted a capability issue, for example, *"<title, question body>. What Linux capability should I use?"*.
- **Fine-Tuning:** For fine-tuning, in addition to the documentation and tutorial data (66 entries), we trained on 20% of the forum data (46 posts). The rationale behind using a 20/80 split for training and testing, respectively, is that our use of LLM differs from conventional machine learning models, which typically rely on learning from historical data to make future predictions. In our case, the questions are considered to be distinct, since end-users are unlikely to repeat questions that have already been answered. Therefore, predicting a capability based on one question does not directly inform the prediction for another. Instead, we focused on structuring the data and guiding the model to leverage documentation effectively for predicting capabilities in new queries. On the 181 filtered test posts, our fine-tuning approach achieved 86.7% matching accuracy for GPT (trained over 15 epochs) and 85.6% matching accuracy for Command-R (trained over 10 epochs) with the capability labels derived from the selected forum answers.

  Figure 5 reflects the accuracy results for each method.

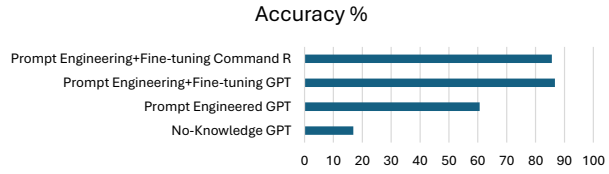


Fig. 5. Accuracy of Different Mechanisms

Error Analysis: An analysis of the incorrect cases reveals that some questions may lack clarity or be potentially misleading. In several instances, the model's answers are not directly related to the questions but instead provide general guidance or alternative suggestions, or users might only include code snippets without explicitly describing the functionality they are asking about. While the quality of forum posts varies, the LLM model's training process does not account for community validation metrics, such as upvotes. Instead, it treats highly-voted and low-voted answers equally, connecting them indiscriminately during training. Our approach involves filtering out low-quality data and fine-tuning on selected high-quality data. However, improving the model's ability

to prioritize and contextualize validated information may require architectural changes beyond standard text-based training.

8 Limitations

Some images fall outside the current target range of ConfigWiz:

- **Non-glibc stacks:** Some images use libraries other than glibc to interact with the OS (e.g., alternative C libraries or specialized libraries), which require different analysis.
- **Non-ELF / dynamic languages:** ConfigWiz focuses on ELF binaries. Interpreted languages (e.g., Python) and dynamic code generation (e.g., JIT/self-modifying code) are out of scope. Even with Ahead-of-Time (AoT) compilation, dynamic features such as reflection or framework-driven behavior can leave functionality outside our current scope (see Section 4.2).
- **Hard-to-exercise workloads:** Images that require specific hardware or environment dependencies may be skipped when runtime execution is infeasible; expanding support for these cases is part of our roadmap.

Rare runtime paths triggered only under specific workloads may remain unobserved, limiting completeness. Additionally, our syscall-based mapping can miss cases where privileges are mediated via `bpf` rather than syscalls, and some mappings depend on kernel build options; using additional compilation configurations and intercepting `bpf` calls can address these gaps.

9 Related Work

Isolation mechanisms in containerized environments have been extensively analyzed in prior studies [19]. For deployment-focused work, Brady et al. [18] leveraged a CI/CD pipeline to monitor container behavior, while Wong et al. [46] applied the STRIDE threat model to identify vulnerabilities in container ecosystems. Martin et al. [33] examined core Docker components, showcasing real-world scenarios where vulnerabilities could be exploited. Wan et al. [41] dynamically profiled system calls invoked in containers under various workloads, and Zeng et al. [49] proposed a stateless model to generate syscall whitelists. Ghavamnia et al. [25] combined dynamic and static analysis to identify required syscalls for binaries extracted from container images. PeX [51] established relationships between Linux permission-check mechanisms and privileged operations, which can be leveraged to gate container actions on the host.

Syscap [47] maps binaries to their required capabilities for cross-checking privilege requirements, with a primary focus on preserving application functionality. However, it does not fully address the overprivilege problem in practical scenarios. LiCA [39] employs LLVM-based code-flow analysis to map system calls to capabilities. While effective for individual programs in static contexts, it is less suited to the dynamic complexity of modern cloud-native systems.

Similarly, Decap [27] is a binary analysis tool that automatically deprivileges single executables by reducing their capabilities. In contrast, ConfigWiz takes a semantic-driven approach to configuration, extending beyond traditional static methods.

The risk of capability misuse extends to container orchestration frameworks like Kubernetes. Rahman et al. [36] conducted a large-scale analysis of Kubernetes manifest misconfigurations, including Linux capability misuse and insecure Docker socket mounts. Provelengios et al. [35] demonstrated syscall-related vulnerabilities in containerized systems, underscoring the importance of mitigating privilege escalation attacks. PrivAnalyzer [22] evaluates programs' privilege requirements to detect and remove unused permissions, while Wist et al. [44] analyzed vulnerabilities in 2,500 Docker images, identifying vulnerability trends to predict future risks. Zeng et al. [50] performed a full-stack vulnerability analysis of cloud-native platforms, including issues stemming from internal (insider) modules.

Prompt-based learning approaches [30], and specifically "chain-of-thought" prompting [43] enable step-by-step reasoning in large language models, opening new possibilities for analyzing complex security configurations. In addition, large language models have been applied to automate configuration tasks: Wang et al. [42] explore translating high-level network policies into low-level device configurations, and Lian et al. [28] investigate using LLMs to validate cloud system configurations. Insights from these studies are valuable as ConfigWiz expands the range of configuration parameters it supports.

10 Conclusion

We presented ConfigWiz, a system that automates privilege configuration for containerized applications by jointly considering both application-derived requirements and user-specified auxiliary requirements. Our evaluation shows that this combination can substantially reduce overprivilege while preserving application functionality across a broad set of container images, and can provide useful auxiliary support on filtered forum-derived user requests.

There are several directions for future work. First, we plan to extend the binary analysis pipeline beyond glibc and ELF-centered workflows to cover a broader range of language runtimes and containerized software stacks. Second, we plan to improve coverage of uncommon execution paths through richer workload generation and deeper flag and driver-level analysis. Third, we plan to broaden auxiliary support by incorporating additional high-quality sources and improving how community-derived evidence is filtered and prioritized.

More broadly, this work highlights the value of connecting low-level privilege mechanisms with higher-level application behavior and user intent. We believe this direction can help make least-privilege configuration more practical for real-world containerized systems.

References

1. False Boundaries and Arbitrary Code Execution. https://forums.grsecurity.net/viewtopic.php?f=7\&t=2522 (2011), accessed on 2022-11
2. Secure Your Containers with this One Weird Trick. https://www.redhat.com/en/blog/secure-your-containers-one-weird-trick (2016), accessed on 2022-10
3. Understanding Docker container escapes. https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/ (2019), accessed on 2022-02
4. Why A Privileged Container in Docker Is a Bad Idea. https://www.trendmicro.com/en_us/research/19/l/why-running-a-privileged-container-in-docker-is-a-bad-idea.html (2019)
5. AquaSecurity Report on Cloud Native Threats. https://info.aquasec.com/cloud-native-threats (2021), accessed on 2021-11
6. docker-bench. https://github.com/docker/docker-bench-security (2021), accessed on 2022-10
7. getrlimit()–Get resource limit. https://www.ibm.com/docs/en/i/7.1?topic=ssw_ibm_i_71/apis/getrlim.html (2021), accessed on 2023-04
8. Is there any way to use hostname command in the container? https://github.com/moby/moby/issues/8902 (2021), accessed on 2022-02
9. PRNewswire Report on Container Market. https://prn.to/4eSR4pX (2021), accessed on 2021-11
10. Global Cloud Native Platforms Market Size (2023 - 2030). https://virtuemarketresearch.com/report/cloud-native-platforms-market (2022), accessed on 2023-04
11. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/ (2023), accessed on 2023-04
12. build-a-native-executable (Mar 2023), https://www.graalvm.org/latest/reference-manual/native-image/, uRL-9 related
13. Container running as privileged. https://securitylabs.datadoghq.com/cloud-security-atlas/vulnerabilities/use-of-privileged-containers/ (2023), accessed on 2023-08
14. Github Ranking. https://github.com/EvanLi/Github-Ranking/ (2023), accessed on 2023-04
15. Github Trending. https://github.com/trending/ (2023), accessed on 2023-04
16. Anonymous: Playing in the sandbox: A study on the usability of seccomp. In: Proceedings of the Symposium on Usable Privacy and Security (SOUPS) (2025), accessed: 2025-08-15
17. Babar, M.A., Ramsey, B.: Understanding container isolation mechanisms for building security-sensitive private cloud. The University of Adelaide, Australia (2017)
18. Brady, K., Moon, S., Nguyen, T., Coffman, J.: Docker container security in cloud computing. In: 2020 10th Annual Computing and Communication Workshop and Conference (CCWC). pp. 0975–0980. IEEE (2020)
19. Bui, T.: Analysis of docker security. arXiv preprint arXiv:1501.02967 (2015)
20. Canella, C., Werner, M., Gruss, D., Schwarz, M.: Automating seccomp filter generation for linux applications. In: Proceedings of the 2021 on Cloud Computing Security Workshop. pp. 139–151 (2021)
21. Cloud Security Alliance: Gameover(lay) vulnerabilities: Cve-2023-2640 and cve-2023-32629. https://cloudsecurityalliance.org/blog/2023/08/overlayfs-vulnerabilities/ (2023), accessed: 2025-08-06

22. Criswell, J., Zhou, J., Gravani, S., Hu, X.: Privanalyzer: Measuring the efficacy of linux privilege use. In: 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 593–604. IEEE (2019)
23. CVE Details: Top 50 products by total number of vulnerabilities in 2024. https://www.cvedetails.com/top-50-products.php?year=2024 (2024), accessed: 2025-08-10
24. Docker Documentation: Runtime privilege and linux capabilities. https://docs.docker.com/engine/security/capabilities/ (2023), accessed: 2025-07-30
25. Ghavamnia, S., Palit, T., Benameur, A., Polychronakis, M.: Confine: Automated system call policy generation for container attack surface reduction. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID) (2020)
26. Ghavamnia, S., Palit, T., Mishra, S., Polychronakis, M.: Temporal system call specialization for attack surface reduction. In: 29th USENIX Security Symposium (USENIX Security) (2020)
27. Hasan, M.M., Ghavamnia, S., Polychronakis, M.: Decap: Deprivileging programs by reducing their capabilities. In: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 395–408 (2022)
28. Lian, Z., Chen, X., Luo, W., Zhang, J.: Configuration validation in cloud systems via llm-assisted analysis. In: Proceedings of the 32nd USENIX Security Symposium. pp. 4127–4144. USENIX Association (2023), https://www.usenix.org/conference/usenixsecurity23/presentation/lian
29. Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., Zhou, Q.: A measurement study on linux container security: Attacks and countermeasures. In: Proceedings of the 34th Annual Computer Security Applications Conference. pp. 418–429 (2018)
30. Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., Neubig, G.: Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys **55**(9), 1–35 (2023)
31. Loukidis-Andreou, F., Giannakopoulos, I., Doka, K., Koziris, N.: Docker-sec: A fully automated container security enhancement mechanism. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS). pp. 1561–1564. IEEE (2018)
32. LWN.net: The kernel is now a cve numbering authority. https://lwn.net/Articles/978711/ (2024), accessed: 2025-07-31
33. Martin, A., Raponi, S., Combe, T., Di Pietro, R.: Docker ecosystem–vulnerability analysis. Computer Communications **122**, 30–43 (2018)
34. Mattetti, M., Shulman-Peleg, A., Allouche, Y., Corradi, A., Dolev, S., Foschini, L.: Securing the infrastructure and the workloads of linux containers. In: 2015 IEEE Conference on Communications and Network Security (CNS). pp. 559–567. IEEE (2015)
35. Provelengios, G., Pouraghily, A., Tessier, R., Wolf, T.: A hardware monitor to protect linux system calls. In: 2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI). pp. 551–556. IEEE (2018)
36. Rahman, A., Shamim, S.I., Bose, D.B., Pandita, R.: Security misconfigurations in open source kubernetes manifests: An empirical study. ACM Transactions on Software Engineering and Methodology (2023)
37. Red Hat: State of kubernetes security report 2024. https://www.redhat.com/en/resources/state-kubernetes-security-report-2024 (2024), accessed: 2025-08-04
38. Sultan, S., Ahmad, I., Dimitriou, T.: Container security: Issues, challenges, and the road ahead. IEEE Access **7**, 52976–52996 (2019)

39. Sun, M., Song, Z., Ren, X., Wu, D., Zhang, K.: Lica: A fine-grained and path-sensitive linux capability analysis framework. In: Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses. pp. 364–379 (2022)
40. TuxCare: The great kernel cve flood of 2024. `https://tuxcare.com/blog/the-great-kernel-cve-flood-of-2024/` (2024), accessed: 2025-07-28
41. Wan, Z., Lo, D., Xia, X., Cai, L., Li, S.: Mining sandboxes for linux containers. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST). pp. 92–102 (2017). https://doi.org/10.1109/ICST.2017.16
42. Wang, C., Scazzariello, M., Farshin, A., Ferlin, S., Kostić, D., Chiesa, M.: Netconfeval: Can llms facilitate network configuration? Proceedings of the ACM on Networking **2**(CoNEXT2), 1–25 (2024)
43. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., et al.: Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems **35**, 24824–24837 (2022)
44. Wist, K., Helsem, M., Gligoroski, D.: Vulnerability analysis of 2500 docker hub images. In: Advances in Security, Networks, and Internet of Things: Proceedings from SAM'20, ICWN'20, ICOMP'20, and ESCS'20. pp. 307–327. Springer (2021)
45. Wiz Research: Gameover(lay): Container escape vulnerabilities in overlayfs. `https://www.wiz.io/blog/gameoverlay-container-escape` (2023), accessed: 2025-07-25
46. Wong, A.Y., Chekole, E.G., Ochoa, M., Zhou, J.: Threat modeling and security analysis of containers: A survey. arXiv preprint arXiv:2111.11475 (2021)
47. Xing, Y., Cao, J., Wang, X., Torabi, S., Sun, K., Yan, F., Li, Q.: Syscap: Profiling and crosschecking syscall and capability configurations for docker images. In: 2022 IEEE Conference on Communications and Network Security (CNS). pp. 236–244. IEEE (2022)
48. Yasrab, R.: Mitigating docker security issues. arXiv preprint arXiv:1804.05039 (2018)
49. Zeng, Q., Xin, Z., Wu, D., Liu, P., Mao, B.: Tailored application-specific system call tables. The Pennsylvania State University, Tech. Rep (2014)
50. Zeng, Q., Kavousi, M., Luo, Y., Jin, L., Chen, Y.: Full-stack vulnerability analysis of the cloud-native platform. Computers & Security **129**, 103173 (2023)
51. Zhang, T., Shen, W., Lee, D., Jung, C., Azab, A.M., Wang, R.: Pex: A permission check analysis framework for linux kernel. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 1205–1220 (2019)