CAPMAN: Detecting and Mitigating Linux Capability Abuses at Runtime to Secure Privileged Containers

Alireza Moghaddas Borhan[*1], Hugo Kermabon-Bobinnec[*1],
Lingyu Wang[3,1✉], Yosr Jarraya[2], and Suryadipta Majumdar[1]

[1] CIISE, Concordia University, Montreal, QC, Canada
[2] Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
[3] School of Engineering, University of British Columbia, Kelowna, BC, Canada
alireza.moghaddasborhan, hugo.kermabonbobinnec,
suryadipta.majumdar@concordia.ca
yosr.jarraya@ericsson.com
lingyu.wang@ubc.ca

Abstract. Linux capabilities represent an important security feature for enabling fine-grained management of privileges. However, limitations in selectively enabling capabilities for processes and lagging adoption from application developers often lead the operators to run containers with unnecessary privileges. Although this can potentially be addressed by modifying the application, minimizing the set of enabled capabilities, assigning capabilities to executable files, or using user space utilities like Ptrace, those solutions typically require manual efforts, only provide partial protection, or incur significant overhead. In this paper, we present CAPMAN, a solution that secures privileged containers by detecting and mitigating potential capability abuses at runtime. Our main idea is threefold. First, CAPMAN examines all capability requests made by system calls to ensure full protection. Second, CAPMAN performs the detection directly inside the Linux kernel to ensure its efficiency. Third, CAPMAN mitigates capability abuses in a transparent manner without requiring any change made to the application or container. Our evaluation of CAPMAN using real-world CVEs and capability abuses shows that it can mitigate all the tested capability abuses (most of which are missed by a state-of-the-art solution) with negligible performance overhead.

1 Introduction

As an important security feature of Linux, capabilities have been around for over two decades [35]. By assigning different privileged operations with different capabilities, this feature allows more fine-grained control of privileges for processes than with the traditional superuser approach. However, the adoption and development of capabilities have been lagging in practice, especially in container environments [25]. For instance, Docker by default enables 14 capabilities for all the processes, and during the whole

[*] These authors contributed equally to this work.

lifetime of a container [8]. By abusing capabilities enabled in such privileged containers, attackers can cause more serious damages such as escaping the containers to attack the underlying infrastructure [25,35,55,71], which is a common concern of Kubernetes operators [55]. Such security threats, combined with the increasing popularity of cloud-native applications hosted in container environments, make managing capabilities for privileged containers an important area to focus on [13,41,63].

Existing solutions have mostly focused on minimizing the capability sets of binary files, through static analysis (e.g., Decap [25] and LiCA [64]), analyzing system calls (e.g., ConfigWiz [32], TCLP [38] and SysCap [75]), and dynamic analysis (e.g., the RootAsRole framework [3, 4, 70]). The required capabilities can then be enforced at runtime using Docker's `--cap-add` and `--cap-drop` options, or Linux Security Modules (LSMs) such as SELinux or AppArmor. Such solutions can reduce the general attack surface of containers before their execution, but this only provides partial protection since attackers with access to the containers (legitimate or not) can still abuse the remaining capabilities at runtime (as shown in our experiments in Section 4). Tools such as SystemTap's *container_check* [46] and BCC's *Capable* [29] can trace capability checks but cannot block suspicious checks. Finally, existing works on limiting system calls for containers (e.g., [7, 17, 33, 39]) can indirectly influence capabilities but do not provide direct solutions for managing capabilities.

To address this research gap, we present CAPMAN, an in-kernel runtime solution for securing privileged containers. CAPMAN analyzes the capability usage of a container offline, and then detects and mitigates capability abuses at runtime. CAPMAN has several unique advantages as follows. First, compared to existing solutions for reducing the attack surface of containers before their execution, CAPMAN provides a complementary solution by extending the protection to runtime. Second, CAPMAN performs its detection entirely inside the kernel, which ensures its efficiency by eliminating the inherent delay for interacting with the user space. Third, CAPMAN mitigates capability abuses by dynamically dropping capabilities in a transparent manner to avoid the need for costly modifications to the container. In summary, our contributions are as follows.

- We propose CAPMAN as the first in-kernel runtime solution for detecting and mitigating capability abuses. Applying CAPMAN to a container environment can prevent attackers from exploiting privileged containers for more severe damages.
- We tackle several key challenges in realizing CAPMAN as follows: i) to ensure CAPMAN can cover every capability request, we develop a `kprobe`-based kernel module to intercept those requests via a kernel function; ii) to avoid the user space delay, we design CAPMAN to perform its detection completely inside the kernel using lightweight whitelisting and machine learning methods; iii) to safely override the rule that only the container itself can drop capabilities, we develop CAPMAN to perform its mitigation using standard kernel functions and procedures.
- Our evaluation of CAPMAN using real-world capability abuses and CVEs demonstrates its effectiveness and efficiency, e.g., it can mitigate all the 10 tested capability abuses (of which eight are missed by an existing solution) with negligible overhead ($< 0.73\%$) and resource consumption ($< 0.5\%$ CPU and < 40 KB memory).

2 Preliminaries

This section gives background on capabilities, the motivating example, and threat model.

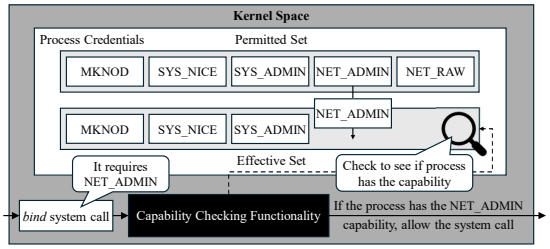

Fig. 1: Examples of Linux capabilities, capability sets, and capability checking functionality.

	Falco [11]	Sysdig [65]	Calico [67]	Docker (DinD) [9]	Netshoot [30]	Dillinger [45]	rclone-mount [57]	Redroid [58]	Wireguard [72]	Phoenix [33]
NET_ADMIN	✓	✓	✓	✓	✓		✓	✓	✓	✓
NET_RAW	✓	✓	✓	✓	✓		✓	✓	✓	✓
SYS_ADMIN	✓	✓	✓	✓		✓	✓	✓	✓	✓
Privileged	✓	✓	✓	✓			✓	✓	✓	✓

Table 1: Examples of popular (over-)privileged containers.

Capabilities and Capability Abuses. Linux capabilities were introduced to provide more fine-grained separation of privileges than the root vs. non-root design [35]. As illustrated in Fig. 1, the *permitted set* includes capabilities that the process can bring to its *effective set*, and the latter is checked by the kernel to ascertain privileges, e.g., the `bind` system call requires the `NET_ADMIN` capability. When a capability is removed from the effective set or the permitted set, it cannot be regained by that process. Although designed to improve security, capabilities are sometimes overloaded with privileges, e.g., `SYS_ADMIN` (nearly as powerful as the root), `NET_RAW`, and `SYS_MODULE` [35]. As such, attackers gaining control of a process with such capabilities can abuse them to lead to more severe security issues such as a container escape [25, 35, 55, 71].

In practice, many *privileged containers* may have such risky capabilities enabled for legitimate reasons, such as security analysis [11, 33], logging [65], networking [67], etc., as illustrated in Table 1. Moreover, regular containers may also become privileged due to i) the lack of temporal control, i.e., capabilities will be enabled for the lifetime of a container even if they are only needed at the beginning; ii) the lack of support from application developers (it is known that most applications are written without considering capabilities [10, 25, 47], and few publishers provide information regarding the required capabilities); iii) the fact that file capabilities are usually not set for common executable files in popular Linux distributions [25]. Running such (over-)privileged containers is a common practice that can lead to serious security concerns.

Motivating Example. Fig. 2 illustrates an example of capability abuses in default container environments (left), naive solutions (upper-right), and our ideas (lower-right).

Capability Abuse. A privileged Nginx container requires the `NET_RAW` capability, which is enabled (but not needed) for the entire lifetime of the container in a default containerized environment. Attackers accessing the container can abuse this capability to escape the container (e.g., via CVE-2020-14386 [48]) and attack the underlying infrastructure.

Existing Solutions. The upper-right corner of the figure shows three categories of potential solutions and their limitations. First, developers can design the application to add a capability from the permitted set to the effective set only when needed, and remove it afterwards. This can significantly reduce the chance of capability abuses. However, as shown in [25], very few applications are capability-aware (e.g., only seven out of 201 *setuid* programs in Ubuntu even use file capabilities), whereas operators typically lack the means due to the unavailability of source codes or their complexity.

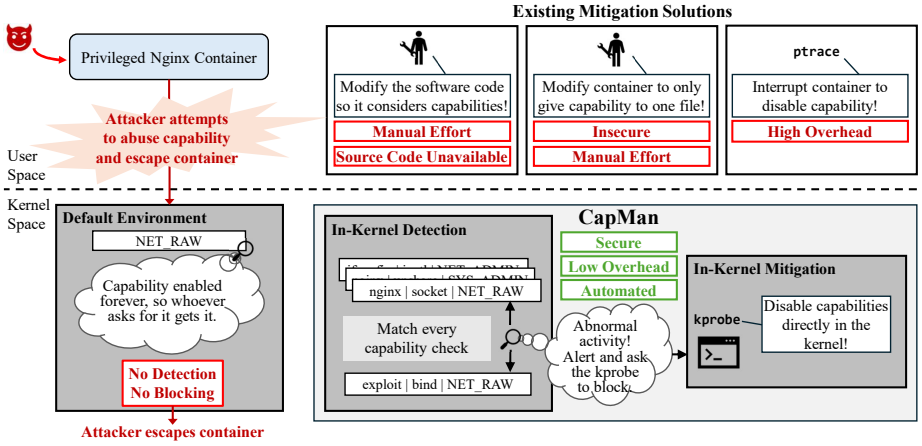

Fig. 2: Motivating example.

Second, the required capabilities can be assigned to executable files of the main application (e.g., using the `setcap` command [35]) such that the container can be run with fewer capabilities enabled. However, in reality attackers can still abuse capabilities by attacking the main application (which is usually the case in real-world attacks).

Third, a potential solution is to perform runtime detection by interrupting and matching each system call and its arguments against the existing mapping between system calls and capabilities [25] using a user space tool, e.g., Ptrace. Once an anomaly is detected, we can drop the capability before returning control back to the process, causing the system call to fail. However, such a user space solution can cause prohibitive delay, as shown in our experiments in Section 4 (as an extreme case, one of the containers we experimented on completely failed when attaching Ptrace to its processes).

Our Ideas. In contrast to those naive solutions, CapMan is designed to run entirely inside the kernel to avoid the delay caused by user space solutions, as illustrated in the lower-right corner of the figure. Specifically, we leverage `kprobes`, a common tool used to insert breakpoints into a running Linux kernel. Working at the kernel level, `kprobes` allows CapMan to access many kernel functions with very low overhead for efficient detection and mitigation. Moreover, CapMan matches every capability check invoked by system calls against the normal usage, which provides full protection without the need to modify the application or container.

Threat Model. Our in-scope threats include attacks involving capability abuses in (over-)privileged containers (e.g., CVE-2022-0492 [49] allows container escape using the `SYS_ADMIN` capability, and many other capabilities can be abused to elevate privileges [52]). These capabilities may be enabled either because they are legitimately required by the container (as shown in Table 1), or due to misconfigurations [41, 55]. We assume an attacker who has already gained access to a container with some privileged capabilities enabled. This could be a legitimate user who has access to the container for normal usage; or an external actor who has gained unauthorized access (e.g., through remote code execution, web-based vulnerabilities, weak credentials, etc.). Zero-days that involve abusing capabilities are also in the scope (as our detection does not rely on signatures of known attacks). Conversely, any attack that does not involve abusing

capabilities, or that tampers with the integrity of CapMan or the underlying infrastructure, the image and container, or the processed data, is out of the scope of this work. Similarly to most existing whitelisting and detection approaches, we assume the normal capability usage can be effectively captured during the offline phase. This is more feasible for containers since these are commonly used to host microservices and cloud-native applications, which usually have a well-defined and relatively simple functionality.

3 CapMan

This section details the methodology and implementation of CapMan.

3.1 Overview of CapMan

As shown in Fig. 3, CapMan works in two phases. First, during the offline phase, the target container is executed in a safe environment for CapMan to capture the normal capability check events as a whitelist or through machine learning (ML). Second, during the runtime phase, the container is run in the production environment for CapMan to detect potential capability abuses through either whitelist matching or ML classification, and CapMan either alerts the user about detected abuses, or drops the capability to mitigate such abuses[4]. To achieve these, CapMan consists of i) a kernel module based on a `kprobe` for data collection, detection, and mitigation, and ii) a user space module developed in Python for data processing (only used during the offline phase).

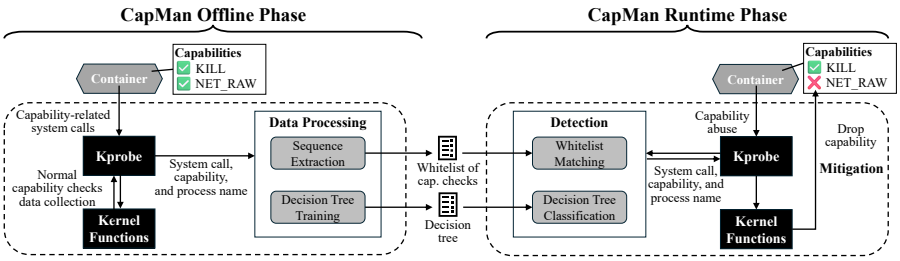

Fig. 3: Methodology Overview.

3.2 Offline Phase

During the offline phase (the left side of Fig. 3), CapMan captures and analyzes the normal capability usage of a target container to obtain a whitelist or ML model. The following i) details the data collection using the kernel module; ii) details the data processing using the user space module; iii) details the time-phase separation technique (inspired by SPEAKER [39]); iv) provides an example of how the offline phase works.

Data Collection Using the Kernel Module. To ensure CapMan can intercept and collect information about every capability check to provide full protection, we leverage the finding of existing works [21, 31, 59, 70] that the kernel relies on the `cap_capable` kernel function to determine whether a process requesting a capability has that capability inside its effective set. Specifically, CapMan installs a `kprobe` as its kernel module to interrupt the `cap_capable` function at the beginning of its execution in order to collect information about capability checks. To ensure `kprobe` only affects the target

[4] A future direction is to integrate CapMan with container orchestration tools like Kubernetes and Open Policy Agent (OPA) to enforce such policies without re-compiling a kernel module.

container, we configure it to only proceed if the namespace ID of the process is the same as the namespace ID of the container (namespaces are used to isolate a container from the rest of the host). We also set a flag in the `kprobe` so it does not block any capability checks and only passively collects information about the passing capability checks.

Various information is collected about each capability check from different sources inside the kernel. First, the capability being checked is obtained from the `dx` register. Second, the system call is obtained by first identifying the reserved process registers using the `task_pt_regs` function, and then obtaining the `orig_ax` register. The namespace ID is deep inside several layers of C structs in the process object in the kernel, namely, `current->nsproxy->pid_ns_for_children->ns.inum`. The PID and process name are obtained from `current->pid` and `current->comm`, respectively. To transmit that information to the user space for processing, we leverage `dmesg` logging [36], a Linux utility for retrieving and displaying information from the kernel ring buffer (we do not consider a `sysfs` file due to its limited size of 4 KB [44]).

Data Processing Using the User-Space Module. The collected data is then processed in the user space to build a whitelist and train an ML model for performing detection in the next stage. Specifically, for the whitelisting method, the goal is to build a whitelist in the form of a set of unique sequences of capability checks, using a sliding window over all the capability check events. For the ML method, since most ML models are known to be too complex and computationally expensive for the kernel space [1, 14], we follow the state-of-the-art work [74] to train a decision tree model that can fit into the kernel space, using the *scikit-learn* Python library. For both methods, each capability check is identified by three attributes, i.e., the capability being requested, the system call, and the name of the process. Note that both methods need to be repeated once container behaviors change (e.g., due to software update or new environments), while a future direction is to apply incremental learning techniques to simplify the model update [54].

Time-Phase Separation. Inspired by SPEAKER [39], we study the capability usage of containers, and our results indicate that many capabilities are only used at the startup of a container and then never needed again. Therefore, CAPMAN also employs a time-phase separation technique during its offline phase. Specifically, similarly as in [39], we first find a threshold after which all containers' capability usage stabilizes, and our results show 20 seconds is an acceptable threshold (see Section 4). We then collect capability checks before/after this threshold for the booting/running phases, respectively.[5]

Example 1. Fig. 4 shows an example of what happens during the offline phase. First, CAPMAN installs its kernel module (`kprobe`). Second, a container is run in a safe environment, where the `ifconfig` process from this container invokes the `socket` system call, which requires the `CAP_NET_RAW` capability. Therefore, the kernel calls its `cap_capable` function, which is intercepted by our `kprobe`. Third, the `kprobe` collects various information as shown in the table (bottom of the figure). The text boxes below the table show how those attributes are retrieved from the relevant registers, kernel variables, and kernel functions. Fourth, the collected information is sent to the user space for processing. Fifth, considering the whitelisting method, the data is processed to identify a unique list of sequences each of which includes three capability checks.

[5] More than two fine-grained phases can also be considered [66, 76].

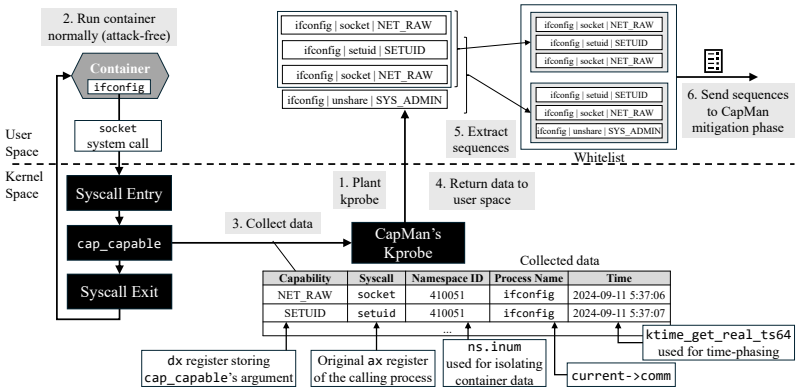

Fig. 4: Example of the offline phase of CAPMAN.

3.3 Runtime Phase

During the runtime phase (the right side of Fig. 3), CAPMAN detects and mitigates capability abuses as they happen in the target container.[6] The detection is performed on the observed capability checks based on either the whitelist or the ML model obtained during the offline phase. As soon as a potential capability abuse is detected, CAPMAN can either alert the user or mitigate it by removing the requested capability from the effective set of the process. The following discusses the challenges, details the detection and mitigation solutions, and gives an example of how the runtime phase works.

Challenges to Detection. A key challenge when detecting capability abuses at runtime is to keep the delay at an acceptable level. Specifically, CAPMAN can no longer mitigate a capability abuse once the corresponding system call has already taken effect. Therefore, CAPMAN needs to put system calls on hold while it determines whether they are making anomalous capabilities requests. As a result, any time spent on the detection becomes additional delay experienced by the system calls (and the user process). This explains why a naive solution of performing detection in user space would cause unacceptable delay to the system call (user process) or even disrupt the normal operation of the kernel itself. First, as `kprobes` are simply code pieces inserted in the middle of the kernel code [42], making the kernel module of CAPMAN (i.e., `kprobe`) wait for the user space detection results may cause the entire kernel (and hence the host system) to be blocked from its normal function (while some parts of the kernel can run system calls in parallel, many other parts do not [2]). Second, the delay introduced by this design consists of the detection time and the time required for exchanging messages between the kernel and user spaces. Such a delay is typically much higher than the average delay between two consecutive system calls (e.g., around 0.003 ms [33]).

Detection. To address the aforementioned challenges, we need to keep the detection lightweight enough such that it may be executed entirely inside the kernel space. For this purpose, we have adopted two lightweight methods, i.e., whitelist matching and decision tree classification (as mentioned earlier, more complex ML models do not fit into the kernel space [1, 14]). These two approaches are complementary and aim at different

[6] CAPMAN can be selectively disabled/enabled with different policies for individual containers.

use cases, i.e., the whitelist matching can achieve higher accuracy if the training data is complete, whereas the decision tree is able to generalize better for previously unseen data. Despite their relative simplicity, those methods can provide reasonable detection accuracy, as demonstrated in our experiments in Section 4.

Specifically, to detect capability checks deviating from the normal behavior, the whitelist of unique sequences (of fixed length N) constructed during the offline phase for each process is fed into the kernel module. At runtime, the kernel module of CAPMAN separately keeps track of the N-most recent capability checks requested by each process, and compares these N capability checks (as a sequence) to the whitelist to determine whether there is an anomaly. Here, N is a parameter reflecting the inherent tradeoff between detection accuracy and time (i.e., larger N may give high accuracy but require more time), which will be evaluated through experiments in Section 4.

For classification-based detection, the decision tree trained during the offline phase is deployed inside the kernel module. This is achieved by translating the corresponding model conditions (i.e., branches) into C code [74]. To overcome the floating-point limitation of the kernel [43], we only employ integer values for branch conditions. At runtime, the kernel module of CAPMAN saves each capability check in a ring buffer, and assembles a sample data point in the form (*process_name*, *syscall_1*, *capability_1*, ..., *syscall_N*, *capability_N*), with N being the length of the sequence. Finally, the data point is passed through the decision tree for classification.

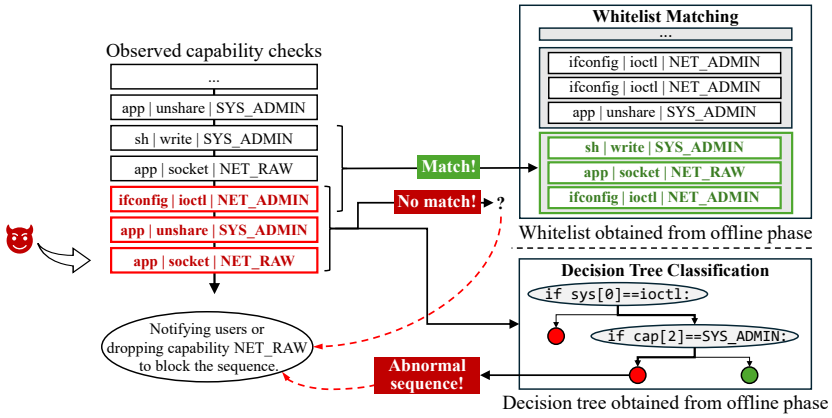

Fig. 5: Example of CAPMAN detection.

Example 2. In Fig. 5, suppose an attacker with access to an Nginx container attempts to escape it by abusing available capabilities. The left side of the figure shows the observed capability checks including those triggered by the attacker (in red color). The upper-right shows how the kernel module of CAPMAN matches each observed sequence of three capability checks against the whitelist. The lower-right shows an excerpt of the in-kernel decision tree (where the nodes show branches written as C code inside CAPMAN's kernel module). For whitelist matching, CAPMAN finds a match in the whitelist for the three latest capability checks when ⟨`ifconfig|ioctl|NET_ADMIN`⟩ is observed. This is expected since there is nothing abnormal about this capability check by itself, although it is indeed part of the attack, since `ifconfig` generally requires `NET_ADMIN` (e.g., as seen in the top two rows). Nonetheless, as the attack progresses,

CapMan successfully detects the sequence shown completely in red color to be an anomaly that should be mitigated.

Challenges to Mitigation. Once a potential capability abuse is detected, CapMan can mitigate it by dropping the requested capability for the process. However, a key challenge here is that the default Linux rules governing the capability sets indicate that a Linux process is not allowed to alter the capabilities of other processes [35], and hence CapMan cannot directly drop the capability for a process. Moreover, although overriding such default rules governing the capability sets is possible, this must be done with extra caution in order not to disrupt the normal kernel functionality, and, similarly to the case of detection, this must also be done in an efficient manner to avoid blocking the normal activities of the container and the wider system.

Mitigation. To address those challenges, we study the Linux kernel source code to understand how to change the capabilities of another process (i.e., the target container's) in a safe and efficient manner. Specifically, our study shows that every process is represented as a C struct in the Linux kernel called `task_struct`, which contains the PID, process name, namespace ID, and the security context of the process stored in another C struct called `cred`. This `cred` object contains the `cap_permitted` and `cap_effective` data objects, which correspond to the permitted and effective sets of capabilities, respectively. Therefore, CapMan can mitigate detected capability abuses by altering those kernel data objects that store the capability sets of each container process. Specifically, since the effective set is what the kernel will check to determine whether the process should be granted access, the kernel module of CapMan removes the capability from the effective set of the process, by modifying `cap_effective`. It also drops the capability from the permitted set by modifying `cap_permitted` to prevent the process (attacker) from adding the capability back. To ensure those modifications are performed in a safe manner, we choose not to directly modify those data objects in an arbitrary way, but instead follow the existing procedure for updating the `cred` struct used in the kernel code, and leverage standard kernel functions.

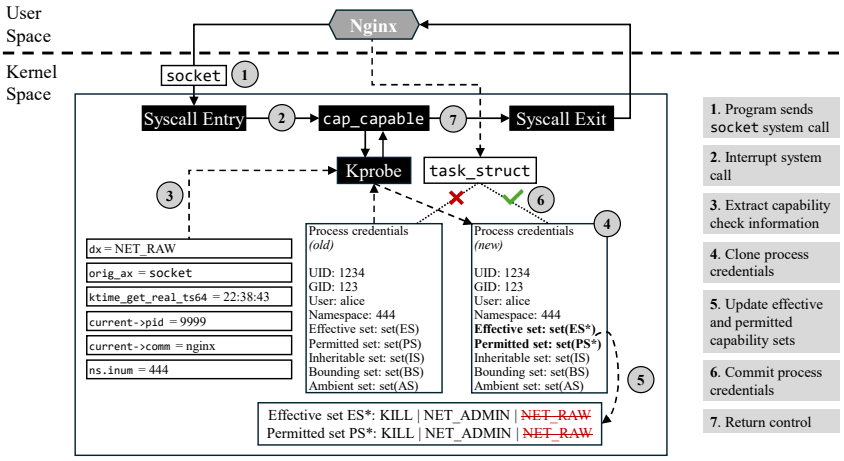

Fig. 6: Example of CapMan mitigation.

Example 3. Following Example 2, Fig. 6 shows how CAPMAN mitigates the detected capability abuse. Specifically, the attacker's process makes a `socket` system call that requests for the `NET_RAW` capability. This is detected as an anomaly by the kernel module of CAPMAN, which runs at the beginning of the `cap_capable` function's runtime. The kernel module thus i) clones the `cred` object of this process using `prepare_creds`, ii) modifies the cloned `cred` struct using `cap_drop` to drop the capabilities from the effective set (and permitted set), iii) commits the new `creds` struct using `commit_creds`, and iv) returns the control back to the kernel to continue with the `cap_capable` function. Consequently, `cap_capable` finds that the process does not have the required capability, and hence the capability abuse is prevented.

4 Evaluation

This section evaluates CAPMAN by answering the following research questions:
RQ1: How effective is it in mitigating capability abuses compared to existing works?
RQ2: How accurate is its detection compared to existing works using only system calls?
RQ3: How much time/CPU/memory overhead does it incur compared to current works?
RQ4: How much does time-phase separation help it reduce the attack surface?

Evaluation Environment. For our experiments, we use a VirtualBox virtual machine (10 vCPUs and 16 GB RAM) running Ubuntu 18.04 on Linux Kernel v5.4. We create Docker containers with 10 of the most popular images on Docker Hub (totaling over 55 million downloads in one week) to evaluate CAPMAN's effectiveness[7]. We also create images with 10 other popular applications that involve intensive capability checks to evaluate CAPMAN's overhead. To evaluate CAPMAN against real-world capability abuses, we implement two exploits of vulnerabilities (CVE-2020-14386 [48] and CVE-2022-0492 [49]), and eight other capability abuses (Table 3a) from the Linux manual and other sources [35, 52]. Each experiment is repeated 100 times.

Effectiveness. To answer RQ1, we compare the effectiveness of CAPMAN to two state-of-the-art solutions, Decap [25] and Confine [17].
CAPMAN *vs. Decap [25].* To show the benefits of CAPMAN's runtime detection and mitigation approach over the static analysis approach of Decap [25], we run 10 of the most popular container images from Docker Hub (totaling over 55 million downloads in a week) with default configurations during CAPMAN's offline phase. We run Decap on the main binary file extracted from each image to identify the required capabilities.

As shown in Table 2, CAPMAN is significantly more effective than Decap in reducing the attack surface. Specifically, Decap allows 18.5 more capabilities than CAPMAN on average per application (with no impact on the applications). For instance, for the RabbitMQ image, Decap allows 24 capabilities, whereas CAPMAN allows only one capability. This shows the inherent difficulty for a static approach like Decap to identify the capabilities actually used at runtime (which is a trivial task for a runtime approach like CAPMAN). A few capabilities (e.g., `SYS_ADMIN`) are allowed by CAPMAN (also by Decap) based on observed capability checks, even though these are not needed (e.g., `SYS_ADMIN` can be checked when a *fork* or *execve* system call is made by the process [3, 70]).

[7] The signficant amount of manual efforts required to invoke each application's normal behaviors to ensure coverage explains why we would not be able to perform a larger-scale study.

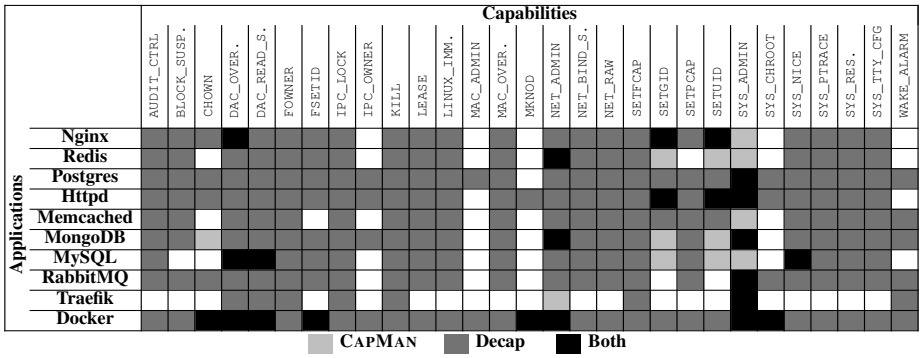

Table 2: Comparing the capabilities allowed by CapMan with the capabilities allowed by Decap [25] for 10 of the most popular container images on Docker Hub.

CapMan *vs. Confine [17].* To show the benefit of considering capabilities (in addition to system calls), we compare CapMan with Confine [17], a widely used system call-based runtime solution. We first apply Confine to a custom Ubuntu container requiring capabilities as mentioned in Table 3a. As Confine turns out to be over-restrictive, disrupting the container's normal operations, we manually unblock the necessary system calls from the generated Seccomp filter to make the containers functional. We then evaluate Confine and CapMan (sequence length of three) against capability abuses.

Table 3a presents the results of CapMan, Confine, and Docker's default capability check [8]. CapMan can detect and mitigate all the capability abuses, whereas Confine only mitigates two out of the 10 abuses (by blocking the system calls shown under the table), and Docker by default cannot mitigate any abuse. One of the capability abuses (installing a kernel module) poses a unique challenge to the mitigation by CapMan, i.e., the `cap_capable` function appears too late in the kernel's source code to mitigate this particular abuse, and consequently is replaced with the `capable` function.

Description	Required Cap.	CapMan	Confine [17]	Docker [8]
Change proc. priorities	SYS_NICE	✓	✓	✗
Clear system logs	SYS_ADMIN	✓	✓	✗
Memory corruption (CVE-2020-14386 [48])	NET_RAW	✓	✗	✗
Bypass ns isolation (CVE-2022-0492 [49])	SYS_ADMIN	✓	✗	✗
Escape cont. via cgroup release agent [52]	SYS_ADMIN	✓	✗	✗
Change iptables rules	NET_ADMIN	✓	✗	✗
Remount read-only filesystem as writable	SYS_ADMIN	✓	✗	✗
Bypass net. restrictions through raw sockets	NET_RAW	✓	✗	✗
Sniff packets and monitor network traffic	NET_RAW	✓	✗	✗
Install kernel modules	SYS_MODULE	✓	✗	✗

(a) Effectiveness of CapMan, Confine [17], and Docker [8].

			Sequence Length (N)			
			1	2	3	4
Syscalls (e.g., [5, 15, 27, 33])		*TPR (%)*	55.14	70.94	79.14	83.41
		FPR (%) / FP	0 / 0	0.007 / 810k	0.017 / 1,968k	0.06 / 6,946k
CapMan Whitelist Matching		*TPR (%)*	72.07	71.99	71.42	70.91
		FPR (%) / FP	0 / 0	0 / 0	0 / 0	1.59 / 3,372k
CapMan Decision Tree	Depth 1	*TPR (%)*	69.94	70.75	70.22	69.96
		FPR (%) / FP	0 / 0	0 / 0	0 / 0	0 / 0
	Depth 3	*TPR (%)*	87.88	87.85	88.51	87.47
		FPR (%) / FP	0 / 0	0 / 0	0 / 0	0 / 0
	Depth 5	*TPR (%)*	88.04	99.89	99.90	99.85
		FPR (%) / FP	0 / 0	0 / 0	0 / 0	0 / 0

(b) Detection accuracy of CapMan and approaches using only system calls.

Table 3: Comparing the effectiveness and accuracy of CapMan

Accuracy. To answer RQ2, we evaluate CapMan 's detection accuracy in comparison to existing approaches using only system calls [5, 15, 27, 33]. We first build a tool to col-

lect sequences of system calls in a per-process manner from a Ubuntu container, while running over 50 common Linux commands using a script, as well as capability abuses. We then re-implement the system call matching approach following existing works, and measure the true positive rate (TPR), the false positive rate (FPR), and the absolute number of false positives (FP). For CAPMAN, we repeat those steps using the capability check data and CAPMAN's whitelist matching and decision tree classification methods (with sequence lengths up to four, since we observe that about 90% of the processes have four or less capability checks). Overall, we collect a dataset of $\approx 39,000$ labeled samples of normal and abuse system calls as well as corresponding capability checks.

As Table 3b shows, the system call-based method (first row) has an increasing true positive rate in the sequence length, ranging from about 55% ($N = 1$) to 83% ($N = 4$). In contrast, CAPMAN with whitelist matching (second row) generally has a more stable true positive rate (between 70% and 72%) across various sequence lengths. The false positive rate of CAPMAN is slightly lower than the system call-based method for $N \leq 3$, and is larger for $N = 4$. However, the absolute number of false positives is more revealing, since for the system call-based method it increases from zero for $N = 1$ to more than 800k for $N = 2$, and almost two and seven million for $N = 3$ and $N = 4$, respectively. In contrast, the absolute number of false positives stays at zero for CAPMAN for $N \leq 3$, and only becomes prohibitive for $N = 4$. This significant difference shows that, although both methods perform similar sequence matching, examining capability checks in addition to system calls allows CAPMAN to dramatically reduce the false positives (from millions to zero). The worse result of CAPMAN under $N = 4$ is mainly due to the fact that most (around 70%) of the processes are observed to have only three capability checks, and hence in practice N should be limited to three for CAPMAN.

The last three rows of Table 3b report the results for CAPMAN with decision trees of different depths (one, three, and five). We can see that a decision tree of depth three or more can yield higher true positive rates than both the system call-based method and CAPMAN's whitelist matching, and a tree of depth five paired with a sequence length of two or more can achieve almost perfect results ranging between 99.85% and 99.90%. In all cases, the false positive rate (and absolute number) remains zero. Clearly, despite its simplicity, the decision tree model can help CAPMAN achieve accurate detection. Moreover, more complex ML models (which do not fit in kernel [1, 14]) can transfer their knowledge to tree-based models [74], paving the way to further improve accuracy.

Overhead. To answer RQ3, we evaluate CAPMAN's overhead compared to that of a user space solution based on Ptrace [34] and a kernel space solution for observability based on the eBPF toolkit BCC [29]. The experiment is based on 10 short-lived containerized applications under test usage that involve intensive system call and capability check activities. Those applications are run under seven different settings: i) the *baseline* setting without CAPMAN, ii) a *BCC capable* setting where we collect capability checks using an eBPF-based tool [29], iii) an *offline* setting where CAPMAN only records the capability checks, iv) a *whitelist* setting with $N = 1$, where CAPMAN performs whitelist matching with a sequence length of one, v) a *whitelist* setting with $N = 3$ (based on our detection accuracy results), vi) a *decision tree* setting, with a tree depth of five and $N = 3$, and vii) a *Ptrace* setting where Ptrace is used to trace the activity of every spawning process within the container.

Table 4a compares the overhead in terms of application performance under those different settings. The overall results show that CAPMAN has a negligible level of overhead on the performance of applications, with an average overhead increase of 0.40%, 1.33%, and 2.51% among these containers, for the *offline* setting, and the *whitelist* settings with $N = 1$ and $N = 3$, respectively. The *decision tree* setting incurs 0.73% overhead on average, comparatively less than the *whitelist* setting, since the former does not require searching in a whitelist, and thus is more efficient. In contrast, Ptrace introduces up to 786.14% overhead (156.59% on average) and even causes the *OpenJDK* container to fail (marked as *F*), which confirms the benefit of CAPMAN's in-kernel detection. The overhead of BCC Capable and CAPMAN's *offline* setting is similar, which is expected since both do not involve detection or mitigation.

	BCC Capable	**CAPMAN**				**Ptrace**
		Offline	Whitelist $N = 1$	Whitelist $N = 3$	Decision Tree	
Alpine	1.41%	1.28%	1.98%	2.48%	1.44%	67.26%
Golang	0.02%	0.05%	1.18%	2.02%	0.23%	62.21%
Node	0.02%	0.08%	0.97%	2.21%	0.65%	102.15%
OpenJDK	1.07%	1.47%	2.08%	2.40%	1.77%	*F*
Python	0.23%	0.11%	0.26%	0.45%	0.21%	49.58%
Stress-ng	0.11%	0.11%	3.13%	7.15%	0.49%	786.14%
MySQL	0.23%	0.25%	1.45%	1.68%	1.24%	85.04%
GCC	0.14%	0.15%	0.85%	1.94%	0.51%	42.68%
Nginx	0.24%	0.15%	0.71%	3.91%	0.38%	167.41%
Redis	0.27%	0.35%	0.65 %	0.83%	0.37%	46.89%
Average	**0.37%**	**0.40%**	**1.33%**	**2.51%**	**0.73%**	**156.59%**

F: The container failed when attaching Ptrace to its processes.

(a) Overhead in terms of application response time.

		Average Response Time per System Call
Baseline		5,447 ns
BCC Capable		5,499 ns
CAPMAN	Offline	5,593 ns
	Whitelist, $N = 1$	5,669 ns
	Whitelist, $N = 3$	5,605 ns
	Decision Tree	6,121 ns
Ptrace		207,237 ns

(b) Overhead in terms of system call response time.

Table 4: Comparing the response time of CAPMAN with BCC Capable and Ptrace.

Table 4b shows the overhead in term of the average response time for handling each system call under CAPMAN, in comparison to Ptrace, BCC, and the *baseline* setting. The results translate to around 0.95% and 2.68% overhead for BCC Capable and *offline* settings, 4.08% ($N = 1$) and 2.90% ($N = 3$) for the *whitelist* setting, and 12.37% for *decision tree* (note such overhead only applies to individual system calls, whereas the aggregated impact on the application and users remains negligible, as shown in Table 4a). In contrast, using Ptrace introduces about $3,705\%$ overhead for each system call. This further confirms the benefits of CAPMAN's in-kernel detection approach.

We also compare CAPMAN's CPU and memory consumption with those of Ptrace [34], BCC Capable [29], and the *baseline* setting. Table 5a shows the results, where the *Container* column is for the container in the presence of the corresponding solution, and the *Solution* column is for the solution itself. Specifically, among all the solutions, Ptrace incurs the most overhead both for the container (1.09% CPU) and for itself (0.98% CPU and 9.89 MB of memory). CAPMAN introduces negligible CPU overhead to both the container and itself, even under the most complex methods (maximum 0.56% and 0.44% for decision tree, and 0.55% and 0.41% for whitelist matching, respectively). The memory consumption in all cases remains constant at around 1.53 MB, while the memory consumption of CAPMAN is only that of the kernel module; which represents at most 36 KB when loaded with the whitelist matching

engine. BCC Capable includes negligible overhead in terms of both CPU and memory consumption, since it only collects capability checks (no detection/mitigation).

Finally, we also measure the impact of the depth of decision trees on the overhead and CPU consumption of CAPMAN. Although a depth greater than 5 does not significantly improve the accuracy in our experiences, we still report results for such cases. Specifically, Table 5b reports the overhead on the application response time, as well as the overall CPU consumption for the `stress-ng` container (our most intensive capability-checking application) while varying the tree depth between 1 and 100. Even with an extreme tree depth of 100, CAPMAN only adds 0.38% of CPU consumption, and incurs 4.44% overhead on the response time (which is almost two times less than with whitelist matching, as shown in Table 4a).

		Container		Solution	
		CPU	Memory	CPU	Memory
Baseline		0.46%	1.53 MB	N/A	N/A
BCC Capable		0.46%	1.52 MB	0.61%	4.64 KB
CAPMAN	Offline	0.52%	1.52 MB	0.40%	24 KB
	Whitelist, $N = 1$	0.55%	1.53 MB	0.42%	24 KB
	Whitelist, $N = 3$	0.55%	1.54 MB	0.41%	36 KB
	Decision Tree	0.56%	1.57 MB	0.44%	24 KB
Ptrace		1.09%	1.53 MB	0.98%	9.89 MB

(a) CPU/memory consumption of the container and CAPMAN, Ptrace [34], and *BCC Capable* [29].

		Overhead on CPU Usage	Overhead on Response Time
Tree Depth	1	+0.08%	+0.17%
	3	+0.09%	+0.21%
	5	+0.08%	+0.65%
	10	+0.08%	+0.84%
	30	+0.12%	+2.52%
	50	+0.22%	+3.33%
	100	+0.38%	+4.44%

(b) CPU/Application response time overhead under different tree depths.

Table 5: The performance overhead of CAPMAN.

The Effect of Time-Phase Separation. To answer RQ4, Fig. 7 shows that 48 out of the top 50 downloaded container images on Docker Hub perform their capability checks in the first 20 seconds (our threshold). Table 6 shows many more capabilities are needed during booting than running, and many high-risk capabilities (e.g., `NET_ADMIN`) can be disabled after booting. Appendix provides more detailed discussions of those results.

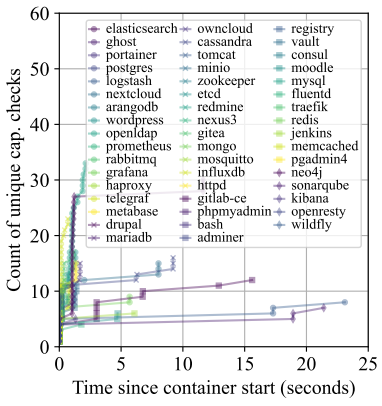

Fig. 7: # of unique capabilities checks over the execution time of containers

	Boot.	Run.	Cap. Disabled by CAPMAN in Running Phase	Running Phase Cap.
Nginx	47	1	`DAC_OVER.`, `SETGID`, `SETUID`	`SYS_ADM.`
Redis	21	0	`SETUID`,`NET_ADM.`, `SYS_ADM.`, `SETGID`	-
Postgres	25	3	-	`SYS_ADM.`
Httpd	14	1	`SETGID`, `SETUID`	`SYS_ADM.`
Memcached	9	0	`SYS_ADM.`	-
MongoDB	39	1	`CHOWN`, `SETGID`, `SETUID`, `NET_ADM.`	`SYS_ADM.`
MySQL	49	0	`SYS_ADM.`, `SYS_NICE`	-
RabbitMQ	35	0	`SYS_ADM.`	-
Traefik	8	0	`NET_ADM.`, `SYS_ADM.`	-
Docker	64	4	`SYS_CHROOT`, `CHOWN`, `MKNOD`, `[...]`, `NET_ADM.`, `SYS_MOD.`	`SYS_ADM.`, `DAC_OVER.`, `DAC_READ_S.`

Table 6: Capability checks during booting/running phases of 10 applications.

5 Discussions and Limitations

Coverage of Normal Behavior. Like most existing solutions involving a learning step [39, 69], CAPMAN shares the challenge of ensuring sufficient coverage of normal behavior during its offline phase. Although not a unique challenge to CAPMAN, we make the following best efforts to address it: i) we build our evaluation containers according to best practices of microservices [62], i.e., with a single task in mind, and minimum dependencies (base image and external packages); ii) we apply existing solutions to generate realistic workloads, e.g., Siege [16] and DirBuster [53] for webservers, HammerDB for databases, and compilation benchmarks [20,26] for GCC/Golang. In real-world scenarios, CAPMAN could be applied alongside the program's integration and unit testing framework [37]. It can also be combined with either static analysis tools that take into account the program's configuration and parameters [19], or program fuzzers [40, 51] to maximize the coverage. Finally, profiles of capability checks may be constructed for popular applications and shared with the community via crowdsourcing.

Zero-day Capability Abuses. Since the whitelist and decision tree of CAPMAN are both learned from the normal behavior (capability checks) of a container, CAPMAN does not require prior knowledge about a vulnerability or exploit. Therefore, similar to other anomaly detection approaches [23], CAPMAN can potentially detect (and mitigate) zero-day attacks involving capability abuses.

Advanced Detection Models. Although its in-kernel design currently limits CAPMAN to simple models such as whitelist and decision tree, knowledge distillation [74] can help transfer knowledge from more advanced ML models (which do not fit in the kernel space) to such tree-based models to further improve the detection accuracy.

Security Trade-offs. The in-kernel design of CAPMAN ensures its efficiency but also renders it a critical system component. To minimize the risk, we i) implement it without any external dependencies; ii) develop its kernel module with only $\approx$ 800 LoC to reduce potential bug sites; iii) use only existing kernel functions to update the capability sets of a process. A future direction is to integrate CAPMAN as a Linux Security Module (LSM) hook [60, 73], or leverage eBPF to make it more secure and usable.

Removing Capabilities vs. Denying the Checks. Instead of removing capabilities from a process's effective set for mitigation, an alternative is to simply deny the capability check by returning `-EPERM`. We choose the former approach to ensure consistency/transparency for user space programs, since some programs may behave differently depending on whether a capability is in the effective sets, and denying capability checks at the kernel level could confuse such programs and cause unexpected behaviors.

Evasive Attacks and False Negatives. The stateful models of CAPMAN make it less sensitive to evasive attacks, such as mimicry [68], since it is harder for attackers to mimic *sequences* of capability checks. To reinforce against such attacks, system call arguments [50] and function call stacks [12] can be added to CAPMAN's detection features. Another evasive technique is to attack during the container's *booting* phase (more enabled capabilities), which can be prevented by booting containers offline. Finally, the immutable nature of containers means that CAPMAN 's offline phase can occur in a controlled, attack-free environment to prevent adversarial ML attacks. These can all help to keep the false negatives of CAPMAN under control. Allocating more resources

to CAPMAN can also reduce false negatives through more capable detection models (e.g., deeper decision trees and longer sequences).

Dynamic Workload. CAPMAN currently does not support dynamic changes in the workload, and its models require re-training for changed behaviors of the container (e.g., software update or new environment). Using incremental learning techniques to dynamically update the models over time can address this [54].

Integration with Orchestration Platforms. CAPMAN is not currently integrated with cloud and container orchestration platforms. While different policies can be selectively enforced for different containers, updating them requires re-compiling the kernel module and keeping track of those policies manually. CAPMAN could benefit from a user space interface to keep track and add/remove policies from the kernel module (e.g., using `ioctl`), and integration with tools like Kubernetes and Open Policy Agent (OPA) will allow central and easier management of CAPMAN's policies.

6 Related Work

Closest to our work, Decap [25] and LiCA [64] employ static analysis to find the necessary and sufficient capabilities set of Linux binaries. Those can be applied to containers, but need manual efforts to identify all binaries in the container, and attackers can still abuse the legitimate capabilities. Similarly, ConfigWiz [32], TCLP [38] and SysCap [75] identify a minimal capabilities set by analyzing the system calls and learning a mapping to capabilities. The RootAsRole framework [3, 4, 70] implements the *capable* tool to identify a binary's "least privilege" capabilities set, and the *sr* tool to enforce them as RBAC policies in Linux. Those tools can effectively reduce the attack surface of applications, although they do not consider sequence of capabilities nor their runtime and time-phase separation aspects as addressed by CAPMAN. Minicon [31], SystemTap's *container_check* tool [46], and BCC's *Capable* tool [29] all employ a similar in-kernel capability hooking technique as our solution but do not consider sequences of capabilities. SELinux [61] and AppArmor [22] can be used to enforce capability restrictions for each process at runtime, although they do not directly provide any analysis and detection features or consider sequences of capability checks like CAPMAN does.

AutoPriv [28] and CapWeave [24] modify a program at compilation time to add directives for allowing capabilities only when needed. Unlike CAPMAN, these require access to the application's source code. PrivAnalyzer [6] measures the efficacy of capabilities and PeX [77] assesses the soundness of permission checks (including capabilities) in the Linux kernel, while neither provides a solution for mitigating capability abuses. As capabilities are closely related to system calls, existing works on system call filtering for containers are also related to our work. SysFilter [7] uses static analysis to identify the set of system calls required by a program, while Confine [17] extends the work to containers. SPEAKER [39], Ghavamnia et al. [18], SysPart [56], and DynBox [76] restrict unnecessary system calls for different phases of a program's life cycle. Phoenix [33] and SFIP [5] go even further by monitoring sequences of system calls and their arguments. All those works cannot prevent capability abuses employing the same system calls as the application, resulting in more false positives, as shown in Section 4.

7 Conclusion

We proposed CapMan, an in-kernel runtime solution for protecting privileged containers against capability abuses. Specifically, we developed a `kprobe`-based kernel module for CapMan to intercept and collect information about capability checks. We also developed lightweight detection methods that could be deployed inside the kernel to ensure efficiency, and safe mitigation methods leveraging standard kernel functions and procedures for altering capabilities. Our implementation and evaluation showed CapMan effectiveness in mitigating capability abuses, with negligible delay and overhead.

Acknowledgements. We thank the anonymous reviewers for their valuable comments. This work was partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Ericsson Canada, Prompt Quebec under Alliance Project 598498, and the Canada Foundation for Innovation (CFI) under JELF Project 38599.

References

1. Bachl, M., Fabini, J., Zseby, T.: A Flow-based IDS using Machine Learning in eBPF. arXiv preprint arXiv:2102.09980 (2022)
2. Billimoria, K.N.: Linux Kernel Programming: A Comprehensive Guide to Kernel Internals, Writing Kernel Modules, and Kernel Synchronization. Packt Publishing Ltd. (2021)
3. Billoir, E., Laborde, R., Wazan, A.S., Rütschlé, Y., Benzekri, A.: Implementing the Principle of Least Privilege Using Linux Capabilities: Challenges and Perspectives. In: Cyber Security in Networking Conference (CSNet). pp. 130–136 (2023)
4. Billoir, E., Laborde, R., Wazan, A.S., Rütschlé, Y., Benzekri, A.: Enhancing Secure Deployment with Ansible: A Focus on Least Privilege and Automation for Linux. In: International Conference on Availability, Reliability and Security. pp. 1–7 (2024)
5. Canella, C., Dorn, S., Gruss, D., Schwarz, M.: SFIP: Coarse-Grained Syscall-Flow-Integrity Protection in Modern Systems. arXiv preprint arXiv:2202.13716 (2022)
6. Criswell, J., Zhou, J., Gravani, S., Hu, X.: PrivAnalyzer: Measuring the Efficacy of Linux Privilege Use. In: Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 593–604 (2019)
7. DeMarinis, N., Williams-King, K., Jin, D., Fonseca, R., Kemerlis, V.P.: sysfilter: Automated System Call Filtering for Commodity Software. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID) (2020)
8. Docker: Docker Engine Security. https://docs.docker.com/engine/security/#linux-kernel-capabilities (2024), accessed: 2024-11-15
9. Docker Hub: Docker in Docker (DinD). https://hub.docker.com/_/docker (2024), accessed: 2024-12-25
10. Edge, J.: Inheriting capabilities. https://lwn.net/Articles/632520 (2015), accessed: 2024-12-25
11. Falco. https://falco.org (2018), accessed: 2024-11-15
12. Feng, H.H., Kolesnikov, O.M., Fogla, P., Lee, W., Gong, W.: Anomaly Detection using Call Stack Information. In: IEEE Symposium on Security and Privacy (S&P) (2003)
13. Findlay, W., Barrera, D., Somayaji, A.: BPFContain: Fixing the Soft Underbelly of Container Security. arXiv preprint arXiv:2102.06972 (2021)
14. Fingler, H., Tarte, I., Yu, H., Szekely, A., Hu, B., Akella, A., Rossbach, C.J.: Towards a Machine Learning-Assisted Kernel with LAKE. In: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 846–861 (2023)

15. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A.: A Sense of Self for UNIX Processes. In: IEEE Symposium on Security and Privacy (S&P). pp. 120–128. IEEE (1996)
16. Fulmer, J.: siege - An HTTP/HTTPS Stress Tester. https://linux.die.net/man/1/siege (2004), accessed: 2025-04-05
17. 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)
18. Ghavamnia, S., Palit, T., Mishra, S., Polychronakis, M.: Temporal System Call Specialization for Attack Surface Reduction. In: USENIX Security Symposium (2020)
19. Ghavamnia, S., Palit, T., Polychronakis, M.: C2C: Fine-grained Configuration-driven System Call Filtering. In: ACM SIGSAC Conference on Computer and Communications Security (CCS) (2022)
20. Golang: Golang Standard Library. https://pkg.go.dev/std (2025), accessed: 2025-04-05
21. Gregg, Brendan: Linux BCC Tracing Security Capabilities). https://www.brendangregg.com/blog/2016-10-01/linux-bcc-security-capabilities.html (2016), accessed: 2025-04-05
22. Gruenbacher, A., Arnold, S.: Apparmor technical documentation. SUSE Labs/Novell (2007)
23. Guo, S., Sivanthi, T., Sommer, P., Kabir-Querrec, M., Coppik, N., Mudgal, E., Rossotti, A.: A Zero-day Container Attack Detection based on Ensemble Machine Learning. In: IEEE International Conference on Emerging Technologies and Factory Automation (ETFA). pp. 1–8. IEEE (2023)
24. Harris, W.R., Jha, S., Reps, T., Anderson, J., Watson, R.N.: Declarative, Temporal, and Practical Programming with Capabilities. In: IEEE Symposium on Security and Privacy (S&P). pp. 18–32 (2013)
25. Hasan, M.M., Ghavamnia, S., Polychronakis, M.: Decap: Deprivileging Programs by Reducing Their Capabilities. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID). pp. 395–408 (2022)
26. Henning, J.: SPEC CPU 2017 Documentation. Standard Performance Evaluation Corporation (SPEC) (2024), https://www.spec.org/cpu2017/
27. Hofmeyr, S.A., Forrest, S., Somayaji, A.: Intrusion Detection using Sequences of System Calls. Journal of Computer Security **6**(3), 151–180 (1998)
28. Hu, X., Zhou, J., Gravani, S., Criswell, J.: Transforming Code to Drop Dead Privileges. In: IEEE Cybersecurity Development (SecDev). pp. 45–52 (2018)
29. IOVisor: *capable.py* - BCC tools. https://github.com/iovisor/bcc/blob/master/tools/capable.py (2024), accessed: 2024-11-15
30. Kabar, N.: Netshoot. https://github.com/nicolaka/netshoot (2024), accessed: 2024-12-25
31. Kang, H., Kim, J., Shin, S.: MiniCon: Automatic Enforcement of a Minimal Capability Set for Security-Enhanced Containers. In: IEEE International IOT, Electronics and Mechatronics Conference (IEMTRONICS). pp. 1–5 (2021)
32. Kavousi, M., Xue, H., Chen, Y., Chen, X., Xing, Y., Sun, K., Schachner, T., Tao, Z.: ConfigWiz: Automating Privilege Configuration for Containerized Applications. Pre-print; SSRN 4995115 (2024)
33. Kermabon-Bobinnec, H., Jarraya, Y., Wang, L., Majumdar, S., Pourzandi, M.: Phoenix: Surviving Unpatched Vulnerabilities via Accurate and Efficient Filtering of Syscall Sequences. In: Network and Distributed Systems Security Symposium (NDSS) (2024)
34. Kerrisk, M.: Ptrace Man Page. https://man7.org/linux/man-pages/man2/ptrace.2.html (2023), accessed: 2024-11-15
35. Kerrisk, M.: Capabilities Man Page. https://man7.org/linux/man-pages/man7/capabilities.7.html (2024), accessed: 2024-11-15
36. Kerrisk, M.: dmesg: Diagnostic Message Log Man Page. https://man7.org/linux/man-pages/man1/dmesg.1.html (2024), accessed: 2024-12-23

37. Khorikov, V.: Unit Testing Principles, Practices, and Patterns. Simon and Schuster (2020)
38. Lee, S., Seo, J., Nam, J., Shin, S.: TCLP: Enforcing least privileges to prevent containers from kernel vulnerabilities. In: ACM SIGSAC Conference on Computer and Communications Security (CCS). pp. 2665–2667 (2019)
39. Lei, L., Sun, J., Sun, K., Shenefiel, C., Ma, R., Wang, Y., Li, Q.: SPEAKER: Split-phase Execution of Application Containers. In: Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) (2017)
40. Li, Y., Chen, B., Chandramohan, M., Lin, S.W., Liu, Y., Tiu, A.: Steelix: program-state Based Binary Fuzzing. In: FSE Joint Meeting on Foundations of Software Engineering (2017)
41. Lin, X., Lei, L., Wang, Y., Jing, J., Sun, K., Zhou, Q.: A Measurement Study on Linux Container Security: Attacks and Countermeasures. In: Annual Computer Security Applications Conference (ACSAC). pp. 418–429 (2018)
42. Linux Kernel Docs: Linux Kernel Probes (Kprobes). https://docs.kernel.org/trace/kprobes.html (2023), accessed: 2024-11-15
43. Linux Kernel Docs: Linux Kernel Floating-point API. https://docs.kernel.org/next/core-api/floating-point.html (2024), accessed: 2024-12-25
44. Linux Kernel Docs: sysfs: Virtual File System for Kernel Objects. https://www.kernel.org/doc/html/latest/filesystems/sysfs.html (2024), accessed: 2024-12-23
45. LinuxServer.io: Dilinger. https://docs.linuxserver.io/deprecated_images/docker-dillinger (2024), accessed: 2024-12-25
46. Liu, X.: SystemTap Examples - *container_check.stp*. https://github.com/boat0/SystemTap-examples/blob/master/container_check.stp (2024), accessed: 2024-11-15
47. McCune, R.: Linux Capabilities and When to Drop All. https://raesene.github.io/blog/2017/08/27/Linux-capabilities-and-when-to-drop-all (2017), accessed: 2024-12-25
48. MITRE: CVE-2020-14386: Linux Kernel Memory Corruption Vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-14386 (2020), accessed: 2024-12-25
49. MITRE: CVE-2022-0492: Linux Kernel Cgroups Vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0492 (2022), accessed: 2024-12-25
50. Mutz, D., Robertson, W., Vigna, G., Kemmerer, R.: Exploiting Execution Context for the Detection of Anomalous System Calls. In: International Symposium on Recent Advances in Intrusion Detection (RAID) (2007)
51. Nagy, S., Nguyen-Tuong, A., Hiser, J.D., Davidson, J.W., Hicks, M.: Breaking through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing. In: USENIX Security Symposium (2021)
52. Nechakhin, V.: Cheat Sheets: Container Escaping - Excessive Capabilities. https://github.com/0xn3va/cheat-sheets/blob/main/Container/Escaping/excessive-capabilities.md (2024), accessed: 2024-12-25
53. OWASP: dirbuster. https://www.kali.org/tools/dirbuster (2009), accessed: 2025-04-05
54. Pancholi, M., Kellas, A.D., Kemerlis, V.P., Sethumadhavan, S.: Timeloops: Automatic System Call Policy Learning for Containerized Microservices. arXiv preprint arXiv:2204.06131 (2022)
55. 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 **32**(4) (2023)
56. Rajagopalan, V.L., Kleftogiorgos, K., Göktas, E., Xu, J., Portokalidis, G.: Syspart: Automated Temporal System Call Filtering for Binaries. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 1979–1993 (2023)
57. Rclone: rclone-mount. https://hub.docker.com/r/mumiehub/rclone-mount (2024), accessed: 2024-12-25
58. Redroid: Redroid. https://hub.docker.com/r/redroid/redroid (2024), accessed: 2024-12-25

59. Robertson, A.: *capable.bt* - BPFTrace. https://github.com/bpftrace/bpftrace (2025), accessed: 2024-04-05
60. senyuuri: Linux Capability - A Kernel Walkthrough. https://blog.senyuuri.info/posts/2021-02-06-linux-capability-a-kernel-workthrough (2021), accessed: 2025-04-05
61. Smalley, S., Vance, C., Salamon, W.: Implementing selinux as a linux security module. NAI Labs Report **1**(43), 139 (2001)
62. Souppaya, M., Morello, J., Scarfone, K.: Application Container Security Guide. Tech. rep., National Institute of Standards and Technology (2017)
63. Sultan, S., Ahmad, I., Dimitriou, T.: Container Security: Issues, Challenges, and the Road Ahead. IEEE Access **7**, 52976–52996 (2019)
64. Sun, M., Song, Z., Ren, X., Wu, D., Zhang, K.: LiCA: A Fine-grained and Path-sensitive Linux Capability Analysis Framework. In: International Symposium on Research in Attacks, Intrusions and Defenses (RAID). pp. 364–379 (2022)
65. Sysdig. https://sysdig.com/ (2023), accessed: 2024-11-15
66. Thévenon, G., Nguetchouang, K., Lazri, K., Tchana, A., Olivier, P.: B-Side: Binary-Level Static System Call Identification. In: Proceedings of the 25th International Middleware Conference. pp. 225–237 (2024)
67. Tigera: Project Calico - Open Source Networking and Security. https://www.tigera.io/project-calico (2024), accessed: 2024-11-15
68. Wagner, D., Soto, P.: Mimicry Attacks on Host-Based Intrusion Detection Systems. In: ACM SIGSAC Conference on Computer and Communications Security (CCS) (2002)
69. Wang, X., Shen, Q., Luo, W., Wu, P.: RSDS: Getting System Call Whitelist for Container Through Dynamic and Static Analysis. In: IEEE International Conference on Cloud Computing (CLOUD). pp. 600–608 (2020)
70. Wazan, A.S., Chadwick, D.W., Venant, R., Billoir, E., Laborde, R., Ahmad, L., Kaiiali, M.: RootAsRole: A Security Module to Manage the Administrative Privileges for Linux. Computers & Security p. 102983 (2022)
71. Wenhao, J., Zheng, L.: Vulnerability Analysis and Security Research of Docker Container. In: IEEE International Conference on Information Systems and Computer Aided Education. pp. 354–357 (2020)
72. WireGuard: WireGuard VPN. https://www.wireguard.com/ (2024), accessed: 2024-12-25
73. Wright, C., Cowan, C., Smalley, S., Morris, J., Kroah-Hartman, G.: Linux Security Modules: General Security Support for the Linux Kernel. In: USENIX Security Symposium (2002)
74. Xie, G., Li, Q., Duan, G., Lin, J., Dong, Y., Jiang, Y., Zhao, D., Yang, Y.: Empowering In-Network Classification in Programmable Switches by Binary Decision Tree and Knowledge Distillation. IEEE/ACM Transactions on Networking **32**(1), 382–395 (2024)
75. Xing, Y., Cao, J., Wang, X., Torabi, S., Sun, K., Yan, F., Li, Q.: SysCap: Profiling and Cross-checking Syscall and Capability Configurations for Docker Images. In: IEEE Conference on Communications and Network Security (CNS). pp. 236–244 (2022)
76. Zhang, Q., Zhou, C., Xu, Y., Yin, Z., Wang, M., Su, Z., Sun, C., Jiang, Y., Sun, J.: Building Dynamic System Call Sandbox with Partial Order Analysis. Proceedings of the ACM on Programming Languages **7**, 1253–1280 (2023)
77. Zhang, T., Shen, W., Lee, D., Jung, C., Azab, A.M., Wang, R.: PeX: A Permission Check Analysis Framework for Linux Kernel. In: USENIX Security Symposium. pp. 1205–1220 (2019)