Cross Container Attacks: The Bewildered eBPF on Clouds

Yi He and Roland Guo, *Tsinghua University and BNRist;* Yunlong Xing, *George Mason University;* Xijia Che, *Tsinghua University and BNRist;* Kun Sun, *George Mason University;* Zhuotao Liu, Ke Xu, and Qi Li, *Tsinghua University*

https://www.usenix.org/conference/usenixsecurity23/presentation/he

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

Cross Container Attacks: The Bewildered eBPF on Clouds

Yi He*, Roland Guo*
Tsinghua University and BNRist

Yunlong Xing
George Mason University

Xijia Che
Tsinghua University and BNRist

Kun Sun
George Mason University

Zhuotao Liu, Ke Xu, Qi Li
Tsinghua University

Abstract

The extended Berkeley Packet Filter (eBPF) provides powerful and flexible kernel interfaces to extend the kernel functions for user space programs via running bytecode directly in the kernel space. It has been widely used by cloud services to enhance container security, network management, and system observability. However, we discover that the offensive eBPF that has been extensively discussed in Linux hosts can bring new attack surfaces to containers. With eBPF tracing features, attackers can break the container's isolation and attack the host, e.g., steal sensitive data, DoS, and even escape the container. In this paper, we study the eBPF-based cross container attacks and reveal their security impacts in real world services. With eBPF attacks, we successfully compromise five online Jupyter/Interactive Shell services and the Cloud Shell of Google Cloud Platform. Furthermore, we find that the Kubernetes services offered by three leading cloud vendors can be exploited to launch cross-node attacks after the attackers escape the container via eBPF. Specifically, in Alibaba's Kubernetes services, attackers can compromise the whole cluster by abusing their over-privileged cloud metrics or management Pods. Unfortunately, the eBPF attacks on containers are seldom known and can hardly be discovered by existing intrusion detection systems. Also, the existing eBPF permission model cannot confine the eBPF and ensure secure usage in shared-kernel container environments. To this end, we propose a new eBPF permission model to counter the eBPF attacks in containers.

1 Introduction

The extended Berkeley Packet Filter (eBPF) is a Linux kernel feature that aims at running userspace code (aka, eBPF programs) in the kernel safely and efficiently via the in-kernel eBPF virtual machine. The eBPF program can read kernel data and call kernel functions via eBPF helpers. By loading and running them directly in the kernel, users can obtain kernel state and manipulate the kernel interfaces without copying data to user space, resulting in better performance and observability. For kernel safety, eBPF programs are only allowed to execute at fixed locations and are validated by the eBPF verifier to prevent malicious behavior, e.g., unbounded loops and out-of-bounds read/write operations. However, some offensive eBPF features (e.g., the `bpf_probe_write_user` [23] helper function that permits writing to memory of other processes) have been added to the kernel since 2016 and they pose a potential threat to the userspace processes. These features are first revealed as dangerous by the community in 2019 [34], and several studies [2, 29, 36, 37] show the potential for developing eBPF based malware or rootkits.

Meanwhile, the security impact of those eBPF offensive features in container environments remains unexplored [8] and their threats are not well understood by the community [49, 56, 62, 64]. eBPF has gained broad usage in existing cloud native world to implement performance profiling, network management, and security monitoring tools for container, such as Cilium [4], Falco [12], and Calico [3]. In the past, the security risks of using eBPF in container environments were mainly considered as the potential to bring in new kernel vulnerabilities and thus lead to container escape [14, 17]. However, we find that even without exploiting eBPF kernel vulnerabilities, eBPF's normal features can bring new attack surfaces to containers. Particularly, the eBPF tracing programs are not restricted by the container's UID and PID namespaces and can monitor processes throughout the entire system, including those running on the host VM or other containers. As a result, those offensive eBPF features available on Linux hosts can also be exploited in containers to harm the container's external processes.

In this paper, we investigate the hazards of eBPF attack vectors in containers and measure their impacts on real-world services. Our study figures out that the known eBPF attacks can harm the processes outside the container, e.g., launching DoS attacks against the victim processes via killing other processes, stealing sensitive information by reading other pro-

*The first two authors contributed equally to this work.

cesses' memory/opened files, or hijacking these processes to execute malicious commands. Furthermore, by hijacking a privileged process (with root permission) outside the containers, attackers can escape the local containers. Specifically, since existing process hijacking solutions [29] cannot work in containers due to resource isolation (discussed in § 4.2), we propose new eBPF-based container escape approaches that have been assigned with one CVE (CVE-2022-42150). Also, we discover that eBPF can facilitate the exploitation of Kubernetes clusters by abusing the insecure Pods deployed on the same VM with the victim containers.

eBPF threats have not attracted enough attention to the container community since they require the *CAP_SYS_ADMIN* capability[1] which may not be granted to most containers. Nevertheless, we analyze the Docker Hub repositories and find that more than 2.5% of containers (see §5.3) have the permissions required for eBPF attacks. Once these containers are exposed, attackers can escape the containers and further compromise Kubernetes clusters via eBPF attacks.

To understand the actual impacts, we perform case studies and explore real-world exploits in online container environments. We investigate both container-based online services (that allow users to run their programs like online programming platforms) and container-based cloud products. Our experimental results show that 5 out of 11 online programming platforms, including four Jupyter Notebook services and one online interactive programming course service, can execute eBPF. We successfully escape the containers of all 5 platforms via eBPF by hijacking various processes outside the containers. Moreover, since 2 of these services are deployed in shared-kernel container environments, we can further exploit the containers of other users to steal data or plant Trojans. The other 3 platforms are deployed in containers that allocate a dedicated VM to a single user. They still rely on the container to confine user abilities (e.g., Colab disallows users to directly connect via ssh), which can be bypassed after container escape.

We also examine cross node attacks on top cloud vendor's Kubernetes cluster products. Three Kubernetes products contain insecure nodes with multiple over-privileged Operator Pods [18]. When any of these nodes are compromised by eBPF attackers, they can abuse the service accounts of all the Pods on this node to exploit the other nodes. Specifically, by exploiting some powerful Operator Pods deployed for data backup, network proxy, and performance metrics (e.g., the Prometheus Pods) on Alibaba ACK cluster, the attackers can compromise the entire cluster. On Azure and AWS's Kubernetes services, attackers can abuse the cluster management Pods to compromise several nodes, but they cannot control the entire cluster since these Pods can only affect limited nodes.

Moreover, eBPF attacks are stealthy and difficult to detect, since they may interfere with the system tracing results that are used by existing defense tools. Existing security tools [3–5, 12] have not included eBPF adversaries in their threat models, so they may also be compromised by malicious eBPF programs in another container. If a container with eBPF permissions is exposed to attackers, they can escape the container and further exploit the Kubernetes cluster without being detected by cloud security tools.

Linux's existing capabilities cannot prevent the abuse of eBPF since they can only disable or enable the eBPF features as a whole. It is infeasible to globally disable eBPF as some users or system services (e.g., `systemd`) still require eBPF, especially when eBPF has been applied to scenarios such as extending NVMe drivers [65], modifying kernel locks [55], and hotfixing the OS or programs' vulnerabilities [41]. One potential countermeasure to mitigate eBPF attacks is to deploy a fine-grained eBPF access control mechanism (e.g., LSM-based tools [5, 9, 20]) that only allows trusted programs to use the offensive eBPF functions. However, since some famous eBPF security tools (e.g., Datadog [5]) still need offensive eBPF features, users may eventually allow some programs to use these features, opening doors for supply chain attacks [40, 50]. To address this problem, we propose a new eBPF permission model that not only provides fine-grained control over a program's available eBPF functions, but also actively protects the victim processes from being violated by eBPF malware.

Contributions. In this paper, we investigate offensive eBPF features in containers and uncover new eBPF-based attack vectors for container escape attacks and cross-node attacks in Kubernetes clusters. To understand the real-world impact, we study various cloud services and find five vulnerable online services. We also find that all existing cloud security products can be exploited by eBPF, and three cloud vendors' default Kubernetes clusters are vulnerable to the proposed cross-node attacks. To tackle these concerns, we propose a new eBPF permission model that is promising to provide better performance and security than the existing LSM-based solutions.

Ethical Consideration. During this research, we examine all target platforms' bug bounty regulations and ensure that our studies comply with their rules. First, we check if a platform has eBPF permissions by executing normal shell commands. For the platforms that can run eBPF, we further identify if they use shared-kernel containers or VM-isolated containers. For the shared-kernel containers (i.e., LanQiao, EduCoder in § 5.1), we inform the service providers and conduct the attacks on the testing environments they provided. For other platforms, including Google Cloud Shell, Colab, Datalore, and Gradient, which have isolated a user's Jupyter container within a dedicated VM, our exploitation happens in our own VM and does not affect other users. For the cloud products discussed in § 5.2, we have informed the four vendors. Since their elastic serverless functions or containers do not support eBPF, we focus on exploiting their Kubernetes products in our private VMs and all tests do not affect others.

[1] Since Linux 5.6, eBPF can use the alternative CAP_BPF capability.

2 Background and Motivation

2.1 eBPF and its Offensive Features

eBPF is introduced into Linux Kernel 3.18 in 2014 as a complement for the classic BPF (known as cBPF, e.g., Seccomp) by providing a universal in-kernel virtual machine with a general purpose RISC instruction set. The eBPF RISC instructions can be compiled from various front-end languages (e.g., C and Rust) using the LLVM toolchain. User space programs can install diverse eBPF program types to specific kernel subsystems, extending the kernel features such as XDP/TC for network flow control, Linux Secure Module (LSM) for security enforcement, and KProbe to hook kernel functions for performance analysis or debugging.

eBPF uses a static analysis based verifier to prevent eBPF code from damaging the kernel by tampering with kernel memory or exhausting resources via unbounded loops. However, some problematic eBPF features provided for the user space [22] may enable the malicious eBPF programs to manipulate network packets, terminate processes, access and modify the memory of other processes, and modify syscalls' arguments or return code [2, 29, 36, 37]. Since running eBPF in the Linux host requires root permission, it makes these powerful features most suitable for post-exploitation attacks. Existing works [29, 36, 37] mainly focus on exploiting these offensive features to enhance Linux rootkits by concealing malicious behaviors. Their hazards in container environments remain unexplored [8, 34, 43].

Inside a container, user's abilities are restricted by various security mechanisms, such as Cgroups for CPU and memory allocation, namespace for UID and PID process isolation, Seccomp for system call restriction, and chroot for filesystem isolation. However, we find that eBPF tracing features can break isolation to probe all processes in the kernel. This allows malicious eBPF programs to exploit the processes outside the containers using the known eBPF attack vectors. Unfortunately, this risk is still not fully understood by the container and kernel communities.

2.2 A Running Example of eBPF Attacks

Figure 1 shows a typical workflow of using eBPF to hijack the container's external processes. First, attackers utilize eBPF tracing features such as `RAW_Tracepoint` to install a snippet of eBPF tracing code (the `tp_exit` function in Figure 1) at the kernel's syscall dispatch function. This code executes whenever any process completes a syscall. Attackers can identify privileged `Bash` processes (step ❶) running on the host as potential hijack targets by recognizing the process name and `uid`. When the `Bash` process is found to be reading a shell script, attackers can utilize the `bpf_probe_write_user` helper to inject malicious shell commands (step ❷ ❸) into the script. Alternatively, attackers can use other offensive eBPF helpers

Figure 1: A running example for hijacking a privilege `Bash` process by altering its scripts via eBPF tracing programs.

described in Table 2 to perform DoS attacks or information theft attacks on the `Bash` processes. Finally, if the injected command is longer than the actual bytes length of the read, the attacker can further modify the return value (step ❹) of the read syscall[2], which indicates the length of bytes read. By hijacking the host's privileged `Bash` process, attackers can escape the container and execute arbitrary commands in the host with root privileges.

2.3 Limitations in eBPF Access Control

After the popular Linux distributions (e.g., Ubuntu, SUSE) disallow the unprivileged usages of eBPF Socket Filter and CGroup programs, current eBPF access control model supports only one permission level that requires *CAP_SYS_ADMIN* capability for all features [6]. Since the *CAP_SYS_ADMIN* is extremely dangerous to containers, Linux 5.6 provides alternative options by decoupling eBPF capabilities. Instead of exclusively using *CAP_SYS_ADMIN*, *CAP_BPF* is proposed for using the bpf syscall, and the installation of particular eBPF program types requires additional capabilities like *CAP_PERFMON* or *CAP_NET_ADMIN*. This approach can preclude some attacks (e.g., altering processes' memory or eBPF maps) using H1, H5 in Table 2 that still require the *CAP_SYS_ADMIN*.

However, these separated capabilities cannot fully prevent eBPF-based attacks such as DoS and information theft. Attackers may still create eBPF-based malware for containers. The proliferation of eBPF based cloud native applications has elevated this risk as users may deploy containers with

[2]The new size should not exceed the actual buffer size, otherwise it can corrupt the stack of `Bash` process and possibly crash it.

untrusted eBPF programs. Even worse, the eBPF-based risks in containers are not yet fully understood and some container services may inadvertently enable eBPF permissions due to various reasons, such as supporting the functionality of mount file systems (see § 5.3). The existing eBPF permission model fails to prevent the untrusted eBPF from misusing these offensive features or harming the containers. We address this problem by suppressing the dangerous features and proposing new permission models to confine a specific process or container's available eBPF features in § 6.

3 Threat Model

We focus on the container environments and assume the attackers can run eBPF tracing programs (e.g., KProbe) inside a container, which can make the bpf syscall and obtain the *CAP_SYS_ADMIN* capability. Though attackers with both the *CAP_BPF* and *CAP_PERFMON* capabilities can launch DoS and information theft attacks without requiring *CAP_SYS_ADMIN*, we assume that all attacks require *CAP_SYS_ADMIN* as *CAP_BPF* is rarely used now.

To understand the feasibility of fulfilling the attacking requirements, we investigate the proportion of real-world containers with eBPF permissions in § 5.3 and find that over 2.5% containers support eBPF tracing programs. Though not all these containers allow code execution (e.g., most Web Services), they may still become vulnerable since attackers can launch supply chain attacks to put eBPF malware in container repositories [50] or run eBPF malware via remote code execution attacks (e.g., Log4Java [21]).

The goals of eBPF attacks are to escape the container and even control the entire Kubernetes cluster without being detected by the security center. We assume all container security measures (e.g., Seccomp, AppArmor, SELinux), kernel hardening techniques (e.g., Non-Executable Memory, KASLR, SMEP/SMAP), and cloud provider's default protections are enabled.

4 eBPF-Based Cross Container Attacks

We first introduce exploitable offensive features in containers and provide an overview of attack vectors for various cross-container attacks in § 4.1. Then, we present several practical exploits to escape containers by hijacking host VM processes in § 4.2. After escaping the containers, we present the Kubernetes cross node attack in § 4.3. Finally, we discuss how to bypass the existing cloud security products in § 4.4.

4.1 Attack Overview

eBPF offensive features have been extensively discussed in Linux hosts for performing malicious activities such as using eBPF tracing features to kill a process [2, 34, 36, 37], obtain or alter the opened files of other processes [2, 37], hijack other processes' execve [2, 34], and using eBPF XDP/TC [29, 43] to build stealthy rootkits . We investigate these eBPF features within containers and find that some of them (e.g., tracing features) can inspect the entire kernel across containers, while others (e.g., networking features) cannot. The eBPF tracing features (e.g., KProbe and RAW_Tracepoint) can probe all processes of the kernel, including those of the host and other containers, and can further affect these processes via the offensive eBPF helpers (listed in Table 2). In contrast, eBPF network features such as XDP/TC can only attach to the network interfaces of their own container, without access to the network interfaces of the host or other containers. We summarize all the offensive eBPF features that can affect container environments in Appendix A.

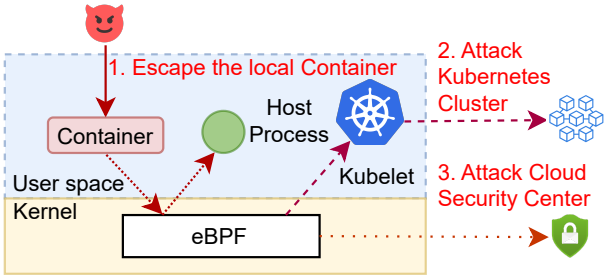

Figure 2: Attackers can escape the container and further exploit the cluster and the cloud security center.

By exploiting eBPF features that are not restricted by containers, attackers can target external processes and perform cross-container attacks. Table 1 summarizes the following attack vectors.

Process/System DoS Attack. The existing eBPF-based process DoS attacks [2, 36] in Linux hosts can also be applied in containers, since the eBPF tracing programs can intercept syscalls both inside and outside the container. When the target processes trigger the eBPF hooks, attackers can manipulate the syscall arguments (**H1**) or return code (**H3**) to crash the programs or send signals (**H4**) to kill the processes. They can continuously capture and kill all the processes of the host, including the processes belonging to other containers. If some crucial processes (e.g., `systemd`, `dockerd`) are killed, the host will be out of service. The attacks are described in detail in the Appendix B.1.

Information Theft Attack. It is known that eBPF tracing programs can access other programs' memory and syscall arguments via the `bpf_probe_read_user` helper (**H2**) [2,37]. Thus, it can be misused to steal other processes' opened files across containers and expose sensitive kernel addresses for further exploitation (discussed in Appendix B.2).

Container Escape Attack. The basic idea of eBPF based container escape attacks is to hijack the privilege processes of the host VM. Tampering with the execve syscall of other processes is a straightforward approach, but compilers generally place the command string (first argument) of execve in read-only memory, making most programs' execve argu-

Table 1: Different attack vectors of eBPF cross container attacks. (The rows with gray background are our new attacks.)

Attack Vector	ID	Description and Impact	Required eBPF Feature	Offensive Helpers					Victim Process
				H1	H2	H3	H4	H5	
Process/System DoS	D1	Killing processes by sending signal	eBPF Trace				✓		Any Process
	D2	Abusing LSM rules to crash processes	eBPF LSM						Any Process
	D3	Altering processes' syscall arguments or return code	eBPF Trace	✓		✓			Any Process
Information Theft	T1	Stealing processes' opened files	eBPF Trace		✓				Any Process
	T2	Stealing kernel data addresses to bypass KASLR	eBPF Trace		✓				-
Container Escape by Hijacking Processes	E1	Code reuse attacks (ROP) to hijack processes	eBPF Trace	✓	✓	✓			Any Process
	E2	Manipulating container's routine tasks	eBPF Trace	✓	✓	✓			Cron, Kubelet
	E3	Shellcode injection during mprotect syscall	eBPF Trace	✓	✓	✓			UPX/JIT
	E4	Forging credentials to login as root via SSH	eBPF Trace	✓	✓	✓			SSH
eBPF Map Tamper	M1	Altering other eBPF programs' maps to manipulate them	Any					✓	eBPF Program

Table 2: The offensive eBPF helpers.

ID	Helper Name	Functionality
H1	bpf_probe_write_user	Write any process's user space memory
H2	bpf_probe_read_user	Read any process's user space memory
H3	bpf_override_return	Alter return code of a kernel function
H4	bpf_send_signal	Send signal to kill any process
H5	bpf_map_get_fd_by_id	Obtain eBPF programs' eBPF maps fd

ments not modifiable. Existing eBPF control flow hijacking approaches [29,34] are inadequate for manipulating processes outside the container since they all require extra information of the host process (e.g., binary code or memory layouts from `/proc/[pid]/*`) that is inaccessible from the container. In § 4.2, we propose practicable exploits for real world container environments.

eBPF Map Tamper Attack. The eBPF maps created by one program can be globally accessed by other programs via the `bpf_map_get_fd_by_id` helper (**H5**). Attackers can manipulate eBPF programs by altering their maps since eBPF programs depend on maps for receiving control configurations and exchanging data with user-space programs. Large eBPF programs like Tetragon [27] and Datadog [5], which rely heavily on eBPF tail call maps to dispatch jumps to other eBPF functions, can be fully paralyzed by deleting the map items.

Among these attacks, we focus on container escape attacks as they can further lead to DoS and information theft attacks. As shown in Figure 2, after escaping the containers, attackers may exploit over-privileged Kubernetes plugins (in § 4.3) to compromise other nodes in the same cluster. Additionally, attackers can evade detection (in § 4.4) by exploiting existing cloud security tools (i.e., cloud security centers) to conceal their malicious activities.

4.2 Container Escape with eBPF

It is infeasible to directly apply the existing eBPF control flow hijacking attacks [29, 34] to the container's host processes as their binary files and memory layout are isolated by the container. Dumping process memory via the `bpf_probe_read_user` (**H2**) is also impossible, since it may fail due to a page fault. Moreover, containers' host VM environments (e.g., Linux version) can be diverse and unknown to attackers. The vulnerable processes (e.g., `Bash`, `Python`, and `Node.js`) that can be exploited to inject malicious scripts may be absent in the target container's host VM.

We address these challenges by attacking the common programs or features used by containers and developing an eBPF snoop program to identify the appropriate attack targets. For instance, since most programs depend on `libc`, we propose a `libc`-based ROP attack (**E1**) that does not require knowledge of the victim processes' memory or implementation. Since the `libc` is mapped into a process's memory space by the mmap syscall, attackers can use eBPF tracing programs to get the `libc`'s base address from the mmap return value. Once the base address is identified, attackers can obtain useful ROP gadget addresses based on the `libc` version. With these gadgets, attackers can use the `bpf_probe_write_user` helper (**H1**) to place the ROP chain on the return address of frequently used syscalls, such as read, to manipulate control flow and perform process hijacking. Our attack can hijack any process by reusing the ROP gadgets in `libc` (see Figure 10 in Appendix B.3).

The containers' common daemon task execution processes, such as `Cron` and `Kubelet`, are ideal hijacking targets as they usually run as root and are included in many containers. However, these programs may stay inactive if there are no new tasks, and it requires some efforts to trigger their task execution. Figure 3 shows that we can trigger and inject malicious `Cron` tasks by manipulating multiple syscall results (**E2**). `Cron` periodically scans its configuration directory (e.g., the `/var/spool/cron/crontabs` directory) to find and run new tasks. We can modify the stat result to make `Cron` immediately scan its configurations (step ❶). To make `Cron` read and execute new tasks, we further modify the fstat result to trick it into loading the global `Cron` task config file, i.e., the `/etc/crontab` file (step ❷). Finally, we modify the arguments of the read syscall (step ❸) to change the task configurations read by `Cron`. In this way, attackers can manipulate `Cron` to execute malicious shell commands in the host VM as root permission. Similar approaches can be used to manipulate `Kubelet`'s static Pods, which are periodic tasks.

To facilitate exploitation, we use the eBPF snoop program to capture all vulnerable processes in a container's host VM,

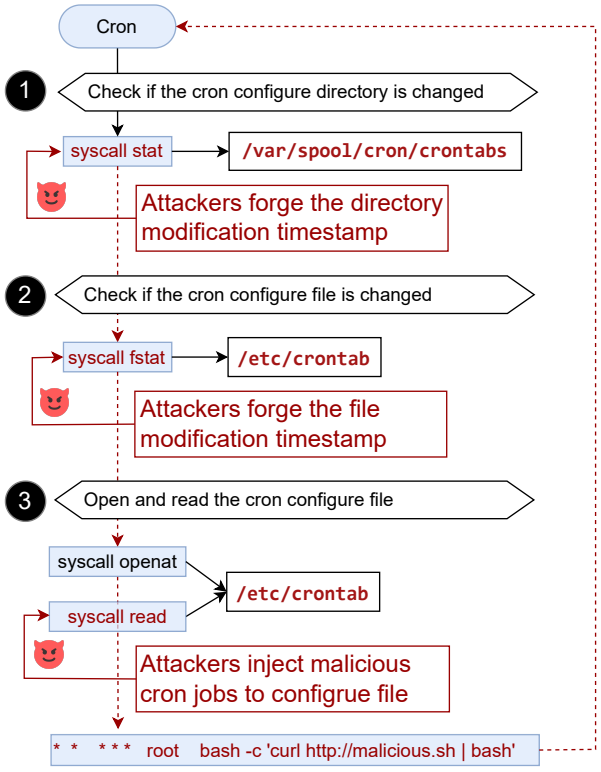

Figure 3: The workflow of forging multiple syscall results to inject malicious jobs and manipulate `Cron`.

such as root processes that execute scripts or have modifiable execve arguments. Additionally, the snoop program can identify programs that change memory attributes to executable, which can be exploited through shellcode injection (see **E3** in Appendix B.3).

4.3 Kubernetes Cross Node Attack

As the most popular cluster management tool, Kubernetes [15] has been widely used by cloud providers to automate the orchestration and scaling of containers. In Kubernetes' terminologies, Pod is the minimal manage and schedule unit that contains one or more containers; Node refers to a physical machine or VM that can host multiple Pods. Some applications may need to access or alter other Pods for better cluster metrics or maintenance. Kubernetes deploys these applications as Pods and assigns credentials with powerful permissions to these Pods. Service Accounts (SA) are used to authenticate these Pods' access to cluster resources. Kubernetes adopts a role-based access control (RBAC) model in which various permissions, such as listing pods, updating pods, or even executing commands in other pods, are granted to roles, and SAs can bind these roles to share their permissions to operate multiple pods.

eBPF attacks pose a significant threat to Kubernetes clusters, as attackers can steal and abuse the SAs of Pods [16] on the same node. A Pod's SA is placed at the path `/var/run/secrets/kubernetes.io/serviceaccount/` and it can be easily obtained by eBPF information theft attacks when the SA files are read by the Kubelet. Also, attackers can misuse all SAs in this node by escaping the container with eBPF. After obtaining the SAs with Pod management permission, attackers can send HTTP requests [16] to the Kubernetes API server to create or update Pods across different nodes. As shown in Figure 4, the number of affected nodes is determined by the permissions of these SAs' binding roles. We find that many Operator [18] pods have powerful permissions and are usually widely distributed as daemon pods (i.e., DaemonSet pod) in every node of the cluster. These Pods greatly aid eBPF attackers in harming more nodes.

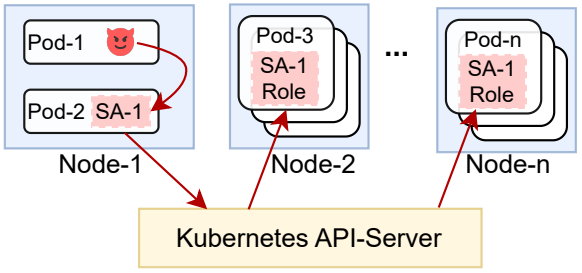

Figure 4: Attackers can control one node and exploit its Service Account (SA-1) to further exploit the Pods that are managed by the service account's binding role (SA-1 Role).

Abusing Operators' Service Accounts. Kubernetes supports Operator [18] extensions, which can automatically manage application deployment, scaling, backup, and updates in a customized way. Operators are deployed as Pods and may have the permission (e.g., Postgres Operator [24] for autoscaling) to create or update Pods on other nodes. Ideally, these Operators should be placed on independent nodes away from the attackers. However, some Operators need to be deployed in multiple nodes together with the Pods that host public services. For instance, Cilium [4] before version 1.11.5 has pod update permissions and is deployed as DaemonSet, which creates a daemon Pod on all nodes for better network management. Once these dangerous Operators are deployed on the same node with the victim containers, attackers can use the eBPF based information theft attacks or container escape attacks to steal their SAs and further deploy malicious Pods to take control of other nodes.

4.4 Bypassing Cloud Security Products

Two factors make it difficult for existing security tools (e.g., Cloud Security Center and container security tools) to detect the eBPF-based cross-container attacks. First, the eBPF attacks happen in kernel space, while cloud security tools [5,12] mainly analyze the processes' system calls and userspace behavior. Second, eBPF malware can preemptively compromise these security tools to prevent them from collecting logs.

The cloud security center can detect malicious behavior in

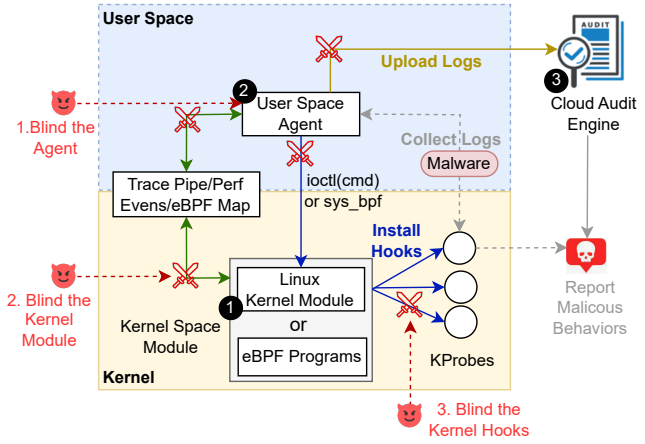

Figure 5: Attacking the cloud security center.

containers and set security rules (e.g., firewall policies) to safeguard the system. Figure 5 depicts a typical architecture of one cloud security center and several cloud security tools (e.g., Falco [12], Datadog [5]). It contains three parts, namely, the kernel space module (implemented by Linux Kernel Module or eBPF) that collects processes and system information via KProbes, the userspace agent that processes and conveys the kernel logs to the outside analysis engine, and the cloud audit engine that detects the anomalous behavior by analyzing data from the userspace agent. Attacker can bypass these security tools by blinding its userspace agent or kernel probes via eBPF and setup an eBPF-based covert command and control (C&C) channel to receive remote commands.

Blinding the Log Collection of the Defenders. All the steps of audit log collection and transport (marked by sword symbols in Figure 5) can be disrupted by malicious eBPF. Attackers can obstruct the userspace agent's system calls and intercept its socket channel to prevent it from receiving and uploading logs. Furthermore, attackers can disable the logging pipe, perf event, and eBPF maps of the kernel module or even bypass the kernel KProbe hooks (see Appendix C). When using the `bpf_probe_write_user` helper, attackers can block security tools from getting syslog warnings and make container escape attacks more stealthy.

Establishing a Covert C&C Channel. Since the container services are usually deployed in cloud hosts with Virtual Private Cloud (VPC) networks, the open ports and the ip allow list are managed by the VPC firewall at the hypervisor level. Typically, the servers with public IPs open only the ports for ssh login (i.e., prot 22) or web service (i.e., port 80). Attackers can reuse these ports to create a covert communication channel [29] by injecting and picking out their own packet based on IP via eBPF XDP and TC programs. By disguising their packets in regular requests, attackers can covertly connect to the C&C server. Then, using eBPF-based process hijacking attacks, they can distribute the exploitation steps of the commands across multiple processes or containers and evade triggering logs with malicious patterns.

Table 3: A statistics of the permissions (i.e., root user, the *CAP_SYS_ADMIN* capability, bpf system call) and vulnerable platforms in various types of container based online services.

Service Type	#Platform	#Root	#CAP	#bpf	#Vul
Jupyter	9	7	4	4	4
Online Labs	2	2	1	1	1
CI/CD Platform	8	4	1	0	0
Online Compiler	5	0	1	0	0

5 Real World Study

We conduct an empirical study to assess the real world impacts of our attacks on online services and cloud products by answering the following research questions:
RQ1: What online services are vulnerable to these attacks?
RQ2: What cloud products are affected by these attacks?
RQ3: How widely is eBPF enabled in containers?
RQ4: Do eBPF cloud tools require eBPF offensive features?

5.1 RQ1: Online Container Services Security

To determine if an online service is vulnerable, we need to run test scripts and eBPF programs locally inside the container of that service. However, not all online services (e.g., most websites) allow users to run customized code. The pre-exploitation phase (e.g., RCE attacks) for remotely running eBPF code is not our research target. Instead, our focus is on online development platforms that allow for custom code execution, enabling our PoCs to run directly on them. As shown in Table 3, we analyze four types of container-based online development platforms, including 9 Jupyter platforms, 2 online programming labs, 8 CI/CD platforms [46], and 5 online compilers. The 8 CI/CD platforms [46] and 5 online compilers have both disabled the bpf system call, preventing the use of eBPF. Online coding platforms, including Jupyter and programming labs, offer more capabilities, with 5 out of 11 platforms allowing the bpf system call and having eBPF permissions.

Table 4 shows the detailed permissions of the 11 online coding platforms and we manage to escape all 5 platforms with eBPF permissions by hijacking the host's various privilege processes. Out of the 11 platforms, only 4 platforms' containers share the kernel. Among these, LanQiao cloud course and EduCoder online coding labs support eBPF and can be exploited by our attacks. CoCalc disallows Root users, thereby restricting users from installing packages with `apt`. As a result, the platform provides thousands of pre-installed programs to meet users' programming needs, and for any new software, users must request the platform administrator to manually install it. Kaggle does not provide network access and prevents the bpf system call. The other 7 platforms all deploy containers in separate VMs. We have escaped the 3

Table 4: The detailed environments, permissions, and exploitation results of online programming services (◐: can escape the container but restricted by the VM, ●: can escape the container and harm other containers.).

Id	Platform	Service Type	Kernel Version	Cloud Vendor	Shared Kernel	Has Root	CAP_SYS_ ADMIN	bpf syscall	Escape	Victim Process
1	Deepnote	Juptyer	5.4.190	AWS	✗	✓	✗	✗		
2	Colab	Juptyer	5.4.188	Google Cloud	✗	✓	✓	✓	◐	sshd, bash
3	CoCalc	Juptyer, Desktop	5.13.0		✓	✗	✗	✗		
4	Datalore	Juptyer	5.11.0	AWS	✗	✓	✓	✓	◐	cron
5	Gradient	Juptyer	5.4.0	Paperspace	✗	✓	✓	✓	◐	bash, kubelet
6	LanQiao	Juptyer, Shell	4.18.0	Alibba Cloud	✓	✓	✓	✓	●	bash, cron
7	EduCoder	Shell	5.4.0	Alibba Cloud	✓	✓	✓	✓	●	cron, kubelet
8	Kaggle	Juptyer	5.10	Google Cloud	✓	✓	✗	✗		
9	Saturn	Juptyer	5.4.181	AWS	✗	✓	✗	✗		
10	mybinder	Juptyer	5.4.0	Google Cloud	✗	✗	✗	✗		
11	O'reilly	Shell	5.4.0		✗	✓	✗	✗		

platforms with eBPF permissions, including Colab, Datalore, and Gradient.

The Detail Approaches of Escaping Containers. Our container escape approaches need to hijack a privileged process of the host. Due to the differences in host environments, we try various exploit methods to hijack a suitable process and spawn a reverse shell for further investigation.

(1) *Hijacking the Container's routine tasks.* EduCoder uses Kubernetes to deploy containers to host online coding courses. To support the courses (e.g. database tutorials and eBPF tutorials) that require additional system calls and capabilities, they enable *CAP_SYS_ADMIN* for all containers. To exploit their containers, we can run our eBPF PoC program in the online shell of any course. We identify that they use Ubuntu as the host OS and contain the routine task scheduler processes, such as `cron` and `kubelet`. We are able to spawn a reverse shell by injecting malicious commands to `cron` tasks.

(2) *Identifying more victim processes via snoop.* Unlike the Educator, Google Colab's Jupyter container is isolated by a separate VM that belongs to the current user. There are far fewer processes outside the container. To identify a target process for hijacking, we run our eBPF snoop program for hours to enumerate all the processes. Finally, we find that Colab's automation tool, which seems to be the Python Fabric, uses `ssh` to connect to the VM as the root user and start some routine maintenance tasks periodically. Thus, we can hijack the `Bash` process and inject the malicious commands of creating a reverse shell when a new ssh client login.

Hazards of Exploiting VM-Based Containers. After escaping the container, we cannot affect other users on platforms like Colab, Datalore, and Gradient as they isolate each user's container in a dedicated VM. However, we can bypass some regulations on these platforms, such as exceeding the limit of free users for Jupyter notebooks or accessing the server via SSH, which is restricted to prime users only. Even protecting the containers with dedicated VMs, containers escape may still be dangerous because Jupyter platforms usually allow users to share their notebooks. The malicious Jupyter Notebooks can be shared to (or via Phishing attack) other users and the eBPF malware can escape to their host VM and plant Trojans to occupy their computing resources or steal their private data (e.g., Colab can access user's Google Driver). Furthermore, the VMs may contain exploitable access control vulnerabilities, e.g., Colab has once shared the same ssh private key [11] for multiple users' VMs. If configured incorrectly, the container management API [10] may be exploited to compromise the pods in other VMs.

5.2 RQ2: Cloud Container Products Security

We confirm eBPF attacks on four leading cloud vendors' all container based cloud products, including cloud shells, serverless functions (e.g., AWS Lambda), serverless containers (e.g., AWS Fargate), and the Kubernetes services.

Exploitable Cloud Container Products. Table 5 shows their default permissions and exploitable products. Among these services, only the Kubernetes services allow to deploy containers with eBPF permissions which can be escaped via eBPF attacks and the risks depend on whether these containers will be deployed online and exposed to the attackers. Specifically, we also find that the Google Cloud Shell can be exploited as it exposes the `docker.sock` to containers, allowing attackers to create a privileged container to run eBPF. The serverless products are now all using virtualized containers which can isolate different users' containers with separate VMs. Only AWS Lambda and Google Cloud Run have the *CAP_SYS_ADMIN* but Google Cloud Run does not compile the eBPF subsystem within the kernel and AWS has disabled the `bpf` system call. Alibaba's serverless products can run unprivileged eBPF such as Socket filter programs but lack the capability to run eBPF tracing programs. Although these products have exposed some dangerous features, such as allowing the unshare syscall, it is still difficult for attackers to exploit them via eBPF [17] or other container escape exploits [49] as they enable Seccomp and AppArmor to prevent many dangerous syscalls (e.g., mount).

Table 5: The eBPF permission of container based services on various platforms. R: has root permission, B: enable the bpf system call, C: has *CAP_SYS_ADMIN* capability, E: container escape. ◐: can escape the container but restricted by the VM, ●: can escape the container and harm other containers.

Service Name	R	B	C	E
Cloud Shell				
AWS Cloud Shell	✓	✗	✗	
Alibaba Cloud Shell	✗	✗	✗	
Azure Cloud Shell	✗	✗	✗	
Google Cloud Shell	✓	✓	✓*	◐
Serverless Function				
AWS Lambda	✗	✗	✓	
Alibaba Function Compute	✓	✓	✗	
Azure Functions	✗	-	✗	
Google Cloud Functions	✗	-	✗	
Serverless Container				
Aws Fargate	✓	✗	✗	
Alibaba Elastic Container Instance	✓	✓	✗	
Azure Container Instances	✓	-	✗	
Google Cloud Run Service	✓	-	✓	
Customized Kubernetes Cluster				
Amazon Elastic Kubernetes Service (EKS)	✓	✓	✓	●
Alibaba Service for Kubernetes (ACK)	✓	✓	✓	●
Azure Kubernetes Service (AKS)	✓	✓	✓	●
Google Kubernetes Engine (GKE)	✓	✓	✓	●

* The capability is gained by exploiting the `docker.sock` (see § 5.3).

Table 6: The number and percentage of nodes that can be affected (C: Create Pod, U: Update Pod, D: Delete Pod) by insecure Pods.

Service	#Pods	#Vul Pods	#DaemonSet Pods	#Affected Node			
				C	U	D	(%)
AWS EKS	12	5	0	0	5	0	100%
Alibaba ACK	58	30	4	5	5	5	100%
Azure AKS	31	3	0	0	3	0	60%
Google GKE	28	0	0	0	0	0	0

Cross Nodes Attacks in Kubernetes Clusters. To investigate if these Kubernetes products are vulnerable to eBPF based cross node attacks, we create a cluster of five worker nodes on each of the four leading cloud providers and measure the number of insecure Pods that can be exploited to perform cross node attacks. Note that we do not install any third-party Kubernetes plugins but use their default settings. Table 6 shows the detailed insecure Pods in each platform. Three platforms contain insecure Pods that can be exploited to launch cross-node attacks.

(1) *Exploitable Clusters.* Three platforms' default Kubernetes clusters (i.e., Alibaba ACK, Azure AKS, and AWS EKS) containers over-privileged Pods. If these Pods are exploited by eBPF attacks, attackers can further exploit other nodes (belonging to the same tenant) by abusing these Pods' service accounts. The Alibaba ACK has four types of DaemonSet Pods to host Prometheus metric services, storage backup services, snapshot management services, and problem detecting services, respectively. These Pods can all create, modify, and delete Pods on every node even though some Pods do not actually requires these permissions, which indicates that attackers can abuse the privileges of DaemonSet Pods to take control of the entire clusters by compromising any node on the cluster. In particular, all these Pods use the `kube-system` namespace which is extremely dangerous because they may be abused to attack the control panel. The Azure AKS has three types of Pods with Pod updating permissions for managing the cloud, each of which can affect three nodes. If any of these three Pods are compromised by attackers via the eBPF attack, the attackers can abuse the Pod updating permission to further compromise three extra nodes by deploying malicious Pods on these nodes. AWS EKS contains cluster-init Operator Pods that can update all the five nodes of the cluster.

(2) *Unexploitable Cases.* All the Google GKE Pods do not have cross-node operation permissions. Only one Pod can modify the CofingMap but are not exploitable. Different to the Alibaba ACK, GKE does not provide any default Operators for metric or cluster management. This means users need to install these tools by themselves which may bring new threats.

Table 7: The vulnerable process numbers in the Linux host VM for different virtualized-container runtime.

VM Env	#Proc	#Root Proc	#Target Proc
AWS FireCracker	12	12	sshd
Alibaba Kata-Containers	5	4	kata-agent
Google Colab	24	15	dockerd, bash
Alpine Linux	15	11	cron, ash

Exploring Virtualized Containers. Cloud vendors like AWS and Alibaba offer high-performance, lightweight VM-based container runtimes for their serverless products, such as AWS FireCracker [28] and Alibaba Kata-Containers [47]. We investigate eBPF based container escape attacks and cross node attacks on these products by locally deploying FireCracker and Kata-Containers. If attackers can access a virtualized container with eBPF permissions, they can still escape the container by hijacking the host VM's privileged processes using eBPF. As shown in Table 7, we measure the resident processes of the host VMs for different container runtimes. The Alpine Linux is adopted by both Firecracker and Kata-Containers as the host VM to isolate the containers. Even though Alpine Linux only contains `musl-libc` and `busybox`, we still find some target processes can be hijacked. gVisor is only a user-space isolation mechanism and cannot prevent eBPF from hijacking other processes. For products using virtualized containers, the container no longer acts as a security boundary, and escaping the container is not considered a threat. However, we find that the Kata-Containers use Kubernetes in the same ways with normal containers and some Operator [18] Pods (e.g., Prometheus) need to be deployed in the same VMs with other Pods. Attackers can still launch Kubernetes cross-node attacks on Kata containers by abusing the Operator pods residing in the victim VM.

Bypassing Cloud Security Products. All the four leading

Table 8: The percentage of insecure Docker Hub repositories.

Dataset	#Repos	#C1	#C2	#C3	All
Top-300	300	2	1	6	9 (3%)
Newest	10000	187	3	179	369 (3.7%)
All [51]	343068	4353	431	3982	8766 (2.56%)

cloud providers have their own security centers, i.e., the Amazon Security Hub for AWS, Microsoft Defender for Cloud, Alibaba Security Center, and the Security Command Center for Google Cloud. For the eBPF malware, only the Alibaba Security Center can produce warning messages to inform that a suspicious eBPF process is launched which may be malicious and the other 3 platforms do not warn the launch of eBPF malware. All the platforms include Alibaba failed to detect and warn the cross container affecting (e.g., hijack, DoS, or read memory) other processes. But some post-exploitations can be alerted by several platforms, e.g., Alibaba can notice reverse shells and copy or move a file, and Azure can notice the download and execution of a remote file via `curl`.

Currently, all the cloud security centers do not support setting enforcement policies to prevent eBPF attacks. Attackers can easily use eBPF to suppress or blind the security centers' user space agents. For instance, the Alibaba Security Center runs a userspace agent (i.e., `AliyunDun`) which can be easily terminated or disrupted by eBPF in ways of intercepting the message channels (e.g., KProbes and sockets) or performing TOUTOC attacks to escape detection.

5.3 RQ3: The Prevalence of Enabling eBPF

We find three kinds of Docker configs that can lead to eBPF permission leakage: (1) running Docker with the `-privileged` flag (**C1**), which grants all capabilities to the container, (2) running Docker with the `-cap-add SYS_ADMIN` flag (**C2**), which grants *CAP_SYS_ADMIN* to the container, and (3) exposing the `docker.sock` to the container, which allows attackers to create a privileged container with the eBPF permission using the Docker API (**C3**). We measure container repositories on Docker Hub to understand how many containers in the wild use these insecure configs.

If a container is run with any of the three configs, it can be misused to perform eBPF attacks. We analyze the documents of the Docker Hub repositories and check their `docker run command` to discern if they support eBPF. As shown in Table 8, we consider three datasets, the *Top-300* dataset contains 300 most downloaded repositories collected in March 2023, the *Newest* dataset contains 10000 newly updated repositories collected in March 2023, and the *All* [51] dataset collected all Docker Hub projects in 2020 and contains 975859 repositories. However, 63% repositories in the *All* dataset are inactive and have no documentation, making it difficult to distinguish their Docker commands. Consequently, we focus on the left 343068 active projects (37%). About 2.5% of all active Docker Hub repositories have eBPF permission, while this number is over 3% for repositories in the Top 300 Highest Download and 10000 Latest Update datasets. This number is also significant in Kubernetes containers. Blaise et al. [31] measure the Kubernetes Helm repositories and find about 4.7% projects need to be run by privileged containers (**C1**). We further analyze the reasons why containers use these configs in appendix D.

Table 9: The offensive helpers used by popular eBPF tools.

Helpers	Tools
bpf_probe_write_user	Datadog
bpf_probe_read	Falco, Datadog, Tetragon, Inspektor, Pixie
bpf_override_return	Tetragon
bpf_send_signal	Tetragon

5.4 RQ4: eBPF Features Used by Cloud

eBPF cross-container attacks stem from the eBPF features that are unrestricted by containers and allow offensive eBPF helpers to tamper with processes outside the container. To understand whether these features are actually required by the existing cloud, we analyze 8 popular eBPF cloud network and security products, including Cilium [4], Calcio [3], Falco [12], Hubble, Datadog, Tetragon, Inspektor-Gadget, and Pixie.

Among the 8 products, Cilium, Calico, and Hubble focus on managing the network connections of containers and only employ eBPF XDP/TC programs without using the eBPF tracing programs. As shown in Table 9, the other 5 products all rely on eBPF tracing and one or more offensive helpers for cloud security and monitoring. Datadog is a container security monitor tool and uses the `bpf_probe_write_user` helper to exchange file system entry states with userspace. Tetragon implements access control enforcement by modifying other processes' system call return value or sending signals to kill processes. Specifically, the `bpf_probe_read` helper is required by all 5 tools for accessing other processes' userspace data such as Socket data, syscall arguments, and processes' exit code. The eBPF security tools are all deployed inside the container and benefit from eBPF's cross-container tracing capability to monitor processes outside the container. Disabling these features can make Kubernetes debugging tools such as Pixie and Inspektor-Gadget unusable.

5.5 Responsible Disclosure

We have disclosed all the vulnerabilities to the related vendors. For the container escape attacks discussed in § 5.1, all 5 vendors have confirmed the problem. Google Cloud Shell, LanQiao, Gradient, and EduCoder have temporarily addressed this issue by restricting eBPF permissions. Google Colab and JetBrains Datalore claim that their threat model takes into account the ability of attackers to take full control of the VM

and not need to take any action. For the Kubernetes cross node attacks discussed in § 5.2, Azure, AWS, and Alibaba have confirmed the issues and plan to remove these overprivileged Pods. We communicate with the Docker and Kubernetes communities and help warn their users to avoid deploying containers with insecure configs (see § 5.3) and eBPF features enabled.

6 Mitigation

eBPF shares the `CAP_SYS_ADMIN` capability with other features (discussed in Appendix D) and may be inadvertently enabled by containers. Currently, the containers with eBPF permissions can utilize all the offensive eBPF features and may be abused by the attackers to perform various cross container attacks. With the increasing popularity of eBPF-based container tools, globally disabling all the eBPF offensive features in the kernel becomes impractical, as some features like `bpf_probe_read` are also required by existing eBPF tools. Therefore, it is essential to implement a fine-grained access control mechanism for eBPF to selectively minimize the eBPF features for the trusted processes. However, users may still need to use eBPF programs with offensive features (e.g., Datadog). To thoroughly avoid sensitive processes being harmed by any trusted/untrusted eBPF programs, the eBPF permission model should make the offensive eBPF features cannot violate certain processes.

6.1 Process Attributes Based Access Control

To mitigate eBPF malware on Linux hosts, LSM-bpf based tools [9, 20] are proposed to implement allow-list policies for confining the eBPF features, eBPF helpers, and eBPF maps of specific processes. The main limitation of LSM-based solutions is that they can only disable or enable eBPF features, but cannot protect the victim processes, while powerful untrusted eBPF programs can still harm other processes. To address this issue, we propose a lightweight and efficient new eBPF permission model called CapBits, which can not only control the eBPF properties of a process but also protects the victim processes from being violated by other process's eBPF offensive features. For example, a program can set the attribute bits to reject being traced by other programs, which can help sensitive processes such as `ssh` to prevent other programs from stealing its private key via eBPF tracing features.

LSM Based eBPF Access Control. The Linux Security Module (LSM) is a framework for enhancing kernel security by adding security hooks to predefined kernel functions. The eBPF LSM subsystem (LSM-bpf) allows users to attach eBPF code to kernel LSM hooks. Existing tools such as eBPF-Monitor [9] use LSM-bpf to implement fine-grained access control for eBPF features, eBPF helpers, and eBPF maps. These approaches add custom eBPF permission checks to the `security_bpf_enter` LSM hook that is invoked each time the bpf syscall is used. We reimplement them in LSM C code as a kernel module for performance evaluation.

Table 10: The latency of different features in different defense solutions (- refers to not supporting such function).

	Default	CapBit	LSM	LSM-bpf
Program-Load	98 ns	110 ns	479 ns	471 ns
Code/Map `fd`	110 ns	103 ns	533 ns	891 ns
Helper	-	100 ns	524 ns	300 ns
Namespace	-	113 ns	-	-

CapBits Design and Implementation. CapBits assigns two bitmap fields to the `task_struct` of a process. The first field, `cap_bits`, manages the available eBPF program types, helper functions, and tracing scopes (e.g., namespace) of a process. The second field, `allow_bits`, specifies which eBPF features (e.g., tracing) from other eBPF programs can work on this process. Each bit of these fields denotes the enable (1) or disable (0) of a feature.

Besides fine-grained control of a process's eBPF offensive features, CapBits can also use these fields to restrict the influence scope (e.g., namespaces, processes) of malicious eBPF programs. Containers can set the CapBits fields for each process during creation. The kernel checks specific bits in the relevant eBPF VM implementation to regulate a process's access to various eBPF features. When using eBPF trace features to probe a target process, the kernel first checks the `cap_bits` of the eBPF program's installation process to verify its capability. Next, kernel checks the target process's `allow_bits` to verify if it permits to be traced by eBPF. For instance, the attribute bits of both the eBPF program's installer and the probed process are checked in the `trace_call_bpf` function to determine if a KProbe should be triggered. In addition, processes can set `allow_bits` to make themselves untraceable by certain eBPF features, such as denying the `bpf_probe_write_user` to modify their memory.

6.2 Evaluation

We compare CapBits with the LSM/LSM-bpf eBPF access control mechanisms in terms of performance, security, and usability. All experiments are run in Linux VMs with an Intel i7-1195G7 CPU (2.90 GHz) and 16GB RAM. The baseline is kernel's capabilities mechanism (default).

Micro-Performance Evaluation. We evaluate the micro performance impact of verifying an allow-list process' access to various eBPF features in different defense mechanisms by analyzing the incurred delays of access control functions for eBPF program types, eBPF code/map `fd` access, eBPF helpers, and namespaces in eBPF tracing programs. To precisely measure the CPU cycles of these functions, we add `rdtsc` time count instructions at the start and end of each function in the kernel and calculate an average time by running each function 100 times. Table 10 shows the micro

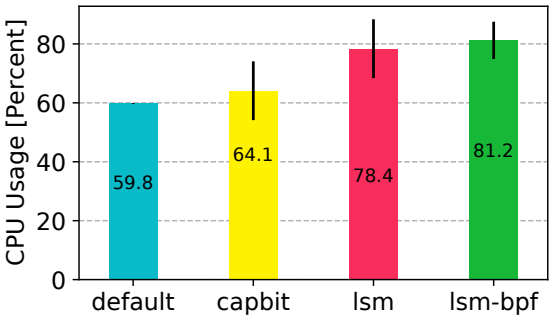

Figure 6: CPU utilization of Cilium.

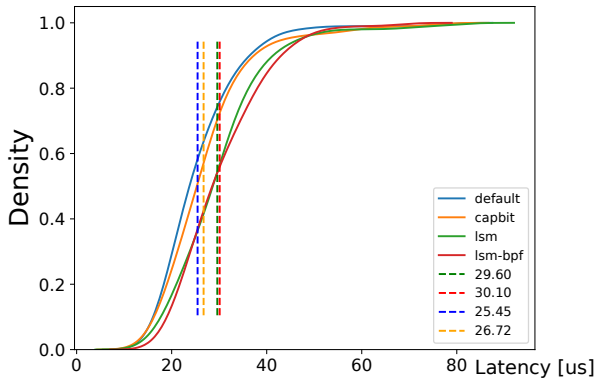

Figure 7: The CDF of Cilium packet latency.

performance of each feature in different defenses. CapBits's permission check delay is similar to the kernel's default capability check, e.g., `ns_capable()`. LSM/LSM-bpf results in a delay 4-8 times larger than CapBits.

Macro-Performance Evaluation. We evaluate the macro-level performance impact of the defenses by measuring Cilium's network packet processing overhead. The experiment is conducted on a Kubernetes cluster with one control plane node and two worker nodes on VMs running our custom kernel. We deploy Cilium at the control plane node (server) and set up 100 Cilium egress packet filtering rules and send requests from the worker nodes (clients). On the server, we start the `iperf3` in server mode and measure the node's CPU utilization (shown in Figure 6) and packet latency (shown in Figure 7). CapBits incurs less than 5% overhead in both CPU usage and packet latency, while LSM/LSM-bpf incur over 31% CPU overhead and 16% latency overhead. This is because the current LSM eBPF permission check can only be added at the beginning of the bpf syscall and is invoked by userspace programs' every eBPF operation including access maps, while CapBits's such checks are only invoked during the eBPF programs' installation.

Security Evaluation. Both LSM-based approaches and CapBits can prevent untrusted eBPF programs from using offensive eBPF features via allow-list policies. However, LSM-based approaches need to receive the policies from userspace and an allow-list process is identified by file path, `PID`, and namespace, which can be easily faked in a container. Attackers can masquerade as an allow list process running in the same container by creating a process with the same identifiers (e.g., path, `PID`) to confuse LSM policies. We find that all existing eBPF access control tools [5, 9, 20, 27] are vulnerable to this kind of process identity confusion attack. A potential mitigation is modifying the container runtime to assign allow-list processes with a unique identifier (UUID) and checking this UUID in kernel's LSM hooks. CapBits can resist this attack as a process's permission attribute fields are stored in the kernel and can only be modified by host's privileged processes (e.g., `setcap`). Processes created by attackers cannot modify these fields to elevate their eBPF permissions.

Usability. As discussed in § 5.4, some famous eBPF programs also use some offensive features. The allow-list processes may have the ability to launch eBPF attacks and this may pose a threat from supply chain attacks [40, 50]. CapBits suppress this risk by restricting an eBPF program's working namespace and enabling sensitive processes to block eBPF probing. Compared to existing LSM tools, CapBits can support all existing eBPF tools to use offensive eBPF features without degrading the security of other processes.

7 Related Work

eBPF Attacks. The eBPF offensive features and eBPF malware are first discussed in DEFCON 27 [34] and have been extensively discussed [2, 29, 36, 37] in the Linux host to build rootkits. We discover that these features can also be used to exploit the container's external processes, and further assess their hazards to real-world cloud environments. Moreover, their control flow hijacking approaches [29, 34] cannot work in the containers (discussed in §4.1) and we propose new approaches of container escape attacks. Besides abusing the offensive features, Kirzner et. al. [44] propose an eBPF based side channel attack to leak user data by triggering speculative execution via eBPF code. Our attack not only provides a more efficient way to read both userspace and kernel-space sensitive data, but also allows arbitrary modification of userspace memory while requiring the same privileges as them.

eBPF Security. Several works [38, 45, 54] have been proposed to improve the security and performance of eBPF. Jitterbug [54] leverages formal verification to check the correctness of different architectures' eBPF JIT compilers. Their approaches can significantly reduce the eBPF's vulnerabilities [14] which may be used to exploit the kernel. But their method needs to decouple the JIT compiler's code from the kernel and verify it as an independent module, which cannot work for the modules which are tightly entangled with the kernel, e.g., the eBPF verifier. Harishankar et al. [60] add a new eBPF verifier abstract domain for the Linux `tnum` data structure. Elazar et al. [38] implement a new abstract interpreter based eBPF verifier to more efficiently vet the insecure

eBPF programs. Our attacks only use valid eBPF programs and do not exploit any eBPF JIT or verifier's vulnerabilities.

Container Security. The existing research on container security falls into three areas, which are attacks on containers [57, 61], analysis and measurement of the insecure containers [49, 50, 62], and container security enforcement [35, 42, 48, 53]. Yang et al. [61] perform DoS attack on the shared-kernel containers by exhausting the global kernel variables and resources in one container. Gringotts [57] introduces a novel Denial-of-Wallet (i.e., degrade other users' computing resource) attack to serverless computing services by slowing down the memory reading performance of all the VM and containers on the same node which share the same CPUs. Lin et al. [49] measure the container exploitation and summary the general container escape attack steps. Unlike these attacks, our eBPF attacks do not exploit the kernel. Instead, we attack the processes outside the container. Liu et al. [50] measure the typosquatting docker repositories and find that the malicious images may be downloaded mistakenly by the users. Minna et al. [52] conclude the security implications in Kubernetes' network layer, which can also be exploited via eBPF.

Syscall Security. Since our eBPF attacks can violate the syscall arguments or results of other processes, e.g., override their file reading contents, or manipulate the heap pointers to conduct Iago [32] attacks, we discuss if the existing syscall security works can prevent these attacks. Ghavamnia et al. [39] implement multiple Seccomp profiles for the program and change the Seccomp configuration to minimize the syscall requirements at different execution phases. Draco [58] uses a new hardware architecture to accelerate Seccomp's syscall vetting. Blackbox [42] can prevent eBPF to violate the syscalls arguments by encrypting the data. Emilia [33] uses fuzzing test to catch the Iago vulnerabilities in TEE's legacy code (i.e., code from normal applications). The approach can reduce the attack surface (i.e., vulnerable processes) of eBPF-based Iago attacks. Proxos [59] allows defining rules to configure the untrusted syscall interfaces and use a private VM to handle them. With their method, the eBPF attacks can no longer damage other processes' syscalls if these syscalls are set as untrusted ones and run in a private VM.

System Extensions via eBPF. eBPF is now widely used to extend various aspects of the system, increasing the need to enable eBPF in containers. For instance, new eBPF program types have been added to extend the system in various areas [30, 41, 55, 65]. ExtFuse [30] implements an efficient and flexible kernel interface for the FUSE file systems via eBPF. XRP [65] uses eBPF to provide a new datapath that can offload the storage operations to the kernel and bypass the traditional kernel storage stacks. SYNCORD [55] allows the applications using the eBPF extensions to customize the kernel locks from user spaces for different hardware and scenarios. RapidPatch [41] uses eBPF to hotpatch the embedded devices and can use one eBPF bytecode patch to fix the same vulnerability across heterogeneous devices.

8 Discussion

Most eBPF features are designed to run on kernel's shared execution paths and have no awareness of the container's isolation. These features are safe to the kernel as they can not alter any kernel status. When the offensive features are added to modify the user space processes and not restricted by the containers, the eBPF cross container attacks emerge. People know eBPF is dangerous [36, 37] and requires capabilities for using it. But they do not know eBPF can be abused to break the container. Due to the resource limitation, not all the multiple-user container services use a separate VM to further isolate the container and many services are still hosted in shared-kernel containers. The widely usage of eBPF in containers (e.g., Cilium) enlarges this threat. Actually, most eBPF tools only use benign eBPF features. eBPF needs a better permission model to selectively enable some useful features to facilitate system observability.

Lessons Learned. eBPF only has one permission level and is not designed to be utilized [26] by unprivileged users (i.e., without the necessary capabilities). This is reasonable for Linux host as only root users have the capabilities to run eBPF. However, to host various types of applications, some containers have to grant these capabilities and the eBPF permissions may be exposed to attackers. The offensive eBPF helpers (documents [23] say that they are only for experiments) make eBPF dangerous to be used in container environments, especially when lots of security tools also rely on eBPF. These tools cannot defend against eBPF attacks from the same security level (e.g., use eBPF to prevent eBPF-based post-exploitations), and difficult to distinguish benign eBPF and malicious eBPF via vetting the eBPF code. Currently, most containers have to disable the bpf syscall to prevent eBPF abuse, which severely limits the development of eBPF. Thus, a better eBPF permission model is required to ensure eBPF can only be used by trusted programs for security or debugging purposes.

9 Conclusion

In this paper, we uncover the eBPF cross-container attacks, which use eBPF tracing features to exploit the processes outside the container to escape the container or even compromise the cluster. Using separate VMs to isolate different tenants' containers or using virtualized containers can make it harmless to escape the container. But a few multiple-user real-world online services are still deployed in shared-kernel environments and are exploitable. Moreover, some of the cloud vendors' container services are risky under the default configuration. To make matters worse, the malicious eBPF programs can easily break the existing security tools' kernel trace points and user space message channels. We systematically study the attacks and reveal the threats. Finally, we propose fine-grained eBPF access control to mitigate this problem.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their insightful feedback. This work was in part supported by NSFC under Grant 62132011 and 61825204, Beijing Outstanding Young Scientist Program under grant BJJWZYJH012019100003011. Kun Sun's work was in part supported by U.S. ONR grant N00014-23-1-2122. Qi Li is the corresponding author of the paper.

References

[1] alluxio. https://hub.docker.com/r/alluxio/alluxio/.

[2] Bad eBPF. https://blog.tofile.dev/2021/08/01/bad-bpf.html.

[3] Calico: networking and network security solution for containers. https://www.tigera.io/project-calico/.

[4] Cilium: eBPF-based Networking, Observability, Security. https://cilium.io/.

[5] Datadog: Cloud Monitoring as a Service. https://www.datadoghq.com/.

[6] Disabled unprivileged eBPF. https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm.

[7] docker in docker. https://hub.docker.com/_/docker.

[8] eBPF Malware. https://redcanary.com/blog/ebpf-malware/.

[9] eBPF-Monitor: detects and protects against eBPF powered rootkits. https://github.com/Gui774ume/ebpfkit-monitor.

[10] Escaped Docker in Azure Functions. https://www.intezer.com/blog/research/how-we-escaped-docker-in-azure-functions/.

[11] Exploit Colab's unchanged host private keys. https://internet-of-tomohiro.netlify.app/google_colab/security_bug.en.html.

[12] Falco: Cloud-Native Runtime Security. https://falco.org/.

[13] IR decoding with BPF. https://lwn.net/Articles/759188/.

[14] Kernel Compromise via eBPF Vulnerabilities. https://pentera.io/blog/the-good-bad-and-compromisable-aspects-of-linux-ebpf/.

[15] Kubernetes. https://kubernetes.io/.

[16] Kubernetes Attack Surface. https://www.optiv.com/insights/source-zero/blog/kubernetes-attack-surface.

[17] Kubernetes Container Escape Using eBPF. https://www.tigera.io/blog/cve-2021-31440-kubernetes-container-escape-using-ebpf/.

[18] Kubernetes Operators. https://kubernetes.io/docs/concepts/extend-kubernetes/operator/.

[19] Linux Kernel Exploitation. https://lkmidas.github.io/posts/20210205-linux-kernel-pwn-part-3/.

[20] Linux Kernel Runtime Integrity with eBPF. https://github.com/Gui774ume/krie.

[21] Log4Java RCE. https://www.lunasec.io/docs/blog/log4j-zero-day/.

[22] Offensive BPF. https://embracethered.com/blog/posts/2021/offensive-bpf/.

[23] Offensive eBPF helpers added to kernel for experimentation in 2016. https://github.com/torvalds/linux/commit/96ae52279594706622ff0585621a13e96b700600.

[24] Postgres operator. https://github.com/zalando/postgres-operator.

[25] rclone. https://github.com/rclone/rclone.

[26] Reconsidering unprivileged BPF. https://lwn.net/Articles/796328/.

[27] Tetragon – eBPF-based Security Observability Runtime Enforcement. https://github.com/cilium/tetragon.

[28] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *NSDI*, 2020.

[29] Marcos Sánchez Bajo. An analysis of offensive capabilities of ebpf and implementation of a rootkit. *Bachelor Thesis of Charles III University of Madrid*, 2022.

[30] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *USENIX ATC*, 2019.

[31] Agathe Blaise and Filippo Rebecchi. Stay at the helm: secure kubernetes deployments via graph generation and attack reconstruction. In *IEEE CLOUD*, 2022.

[32] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *ASPLOS*, 2013.

[33] Rongzhen Cui, Lianying Zhao, and David Lie. Emilia: Catching Iago in legacy code. In *NDSS*, 2021.

[34] Jeff Dileo. Evil eBPF: Practical Abuses of an In-Kernel Bytecode Runtime. In *DEFCON 27*, 2019.

[35] William Findlay, David Barrera, and Anil Somayaji. Bpfcontain: Fixing the soft underbelly of container security, 2021.

[36] Guillaume Fournier and Sylvain Baubeau. With Friends like eBPF, who needs enemies ? In *Blackhat USA*, 2021.

[37] Guillaume Fournier, Sylvain Baubeau, and Sylvain Baubeau. eBPF, I thought we were friends! In *DEFCON 29*, 2021.

[38] Elazar Gershuni, Nadav Amit, A. Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Shmuel Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *PLDI*, 2019.

[39] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *USENIX Security*, 2020.

[40] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan. Investigating package related security threats in software registries. In *IEEE S&P*, may 2023.

[41] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. RapidPatch: Firmware hotpatching for Real-Time embedded devices. In *USENIX Security*, 2022.

[42] Alexander Van't Hof and Jason Nieh. BlackBox: A container security monitor for protecting containers on untrusted operating systems. In *OSDI*, 2022.

[43] Pat Hogan. Warping Reality - creating and countering the next generation of Linux rootkits using eBPF. In *DEFCON 29*, 2021.

[44] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security*, 2021.

[45] Hsuan-Chi Kuo, Kai-Hsun Chen, Yicheng Lu, Dan Williams, Sibin Mohan, and Tianyin Xu. Verified Programs Can Party: Optimizing Kernel Extensions via Post-Verification Merging. In *EuroSys*, 2022.

[46] Zhi Li, Weijie Liu, Hongbo Chen, XiaoFeng Wang, Xiaojing Liao, Luyi Xing, Mingming Zha, Hai Jin, and Deqing Zou. Robbery on devops: Understanding and mitigating illicit cryptomining on continuous integration service platforms. In *IEEE S&P*, 2022.

[47] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. RunD: A lightweight secure container runtime for high-density deployment and high-concurrency startup in serverless computing. In *USENIX ATC*, 2022.

[48] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. Secure namespaced kernel audit for containers. *Proceedings of the ACM Symposium on Cloud Computing*, 2021.

[49] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on linux container security: Attacks and countermeasures. In *ACSAC*, 2018.

[50] Guannan Liu, Xing Gao, Haining Wang, and Kun Sun. Exploring the unchartered space of container registry typosquatting. In *USENIX Security*, 2022.

[51] Peiyu Liu, Shouling Ji, Lirong Fu, Kangjie Lu, Xuhong Zhang, Wei-Han Lee, Tao Lu, Wenzhi Chen, and Raheem Beyah. Understanding the security risks of docker hub. In *ESORICS*, 2020.

[52] Francesco Minna, Agathe Blaise, Filippo Rebecchi, Balakrishnan Chandrasekaran, and Fabio Massacci. Understanding the security implications of kubernetes networking. *IEEE S&P*, 2021.

[53] Jaehyun Nam, Seungsoo Lee, Hyunmin Seo, Phil Porras, Vinod Yegneswaran, and Seungwon Shin. BASTION: A security enforcement network stack for container networks. In *USENIX ATC*, 2020.

[54] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the linux kernel. In *OSDI*, 2020.

[55] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-Informed kernel synchronization primitives. In *OSDI*, 2022.

[56] Liz Rice. *What Is eBPF?* O'Reilly Media, Inc., 2022.

[57] Junxian Shen, Han Zhang, Yantao Geng, Jiawei Li, Jilong Wang, and Mingwei Xu. Gringotts: Fast and accurate internal denial-of-wallet detection for serverless computing. In *CCS*, 2022.

[58] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. Draco: Architectural and operating system support for system call security. In *MICRO*, 2020.

[59] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *OSDI*, 2006.

[60] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, precise, and fast abstract interpretation with tristate numbers. In *CGO*, 2022.

[61] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, and Kui Ren. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. In *CCS*, 2021.

[62] Y. Yang, W. Shen, B. Ruan, W. Liu, and K. Ren. Security challenges in the container cloud. In *TPS-ISA*, 2021.

[63] Kyle Zeng, Yueqi Chen, Haehyun Cho, Xinyu Xing, Adam Doupé, Tiffany Bao, and Yan Shoshitaishvili. Playing for K(H)eaps: Understanding and improving linux kernel exploit reliability. In *USENIX Security*, 2022.

[64] Xiaowei Zhao, Hong Yan, and Jiantong Zhang. A critical review of container security operations. *Maritime Policy & Management*, 2017.

[65] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *OSDI*, 2022.

A eBPF Offensive Features

With the development of Linux eBPF, it can now obtain the kernel status, modify network packets, or alter user space programs. Some eBPF features [2, 34, 37] are dangerous to Linux hosts and containers which may be misused by the attackers. The normal eBPF program types, including socket filter and cgroup (i.e., CGROUP_SKB), do not require any permissions because they can just set filter rules to copy socket packets to user space and cannot alter the system. However, most Linux distributions (e.g., Ubuntu [6], SUSE) disable these features to avoid them being abused by unprivileged users to exploit the kernel vulnerabilities of eBPF. The other eBPF program types all need permissions and can only be used by privileged users. They provide powerful functions such as obtaining the kernel status, managing the network data path, or carrying out offensive operations to affect other processes. We elaborate on all the eBPF features that can be used to perform malicious behaviors as follows:

eBPF XDP and TC Program. The eBPF XDP and TC (traffic control) programs are attached to a network interface for processing packets, such as drop packets, redirect packets, and even modify packets. These functions can affect the user space processes by intercepting their network connections. However, they can not access the network interfaces of the hosts or other containers from the container.

eBPF LIRC Program. The eBPF LIRC [13] programs can implement customize infrared (IR) decoding or encoding extends via eBPF. This feature may be abused to inject keyboard events by the attackers. Fortunately, it is default disabled by most Linux distributions such as Ubuntu.

eBPF LSM Program. In the past, the LSM hooks cannot be set via Linux kernel modules. The eBPF LSM programs introduce a new way to set LSM hooks and define access control and audit policies via eBPF code. This feature can be used inside the container to manage the resources of the whole kernel, including other containers and the host.

eBPF KProbe/Kretprobe and Tracepoint Program (tracing features). The eBPF KProbe/Tracing program can be attached to a specific system call or kernel function and triggered every time when the system call or function is invoked by a process. These programs provide several eBPF helpers (shown in Table 2) which can cross-container affect external processes.

eBPF UProbe. The eBPF UProbe program allows installing hooks at a process's user space addresses. It can only hook the process inside the containers as it needs to obtain the symbols from the program's binary file.

B Detailed Exploits for Local Containers

B.1 DoS Attack

```
SEC("raw_tracepoint/sys_enter")
int tp_enter(struct bpf_raw_tracepoint_args *ctx) {
    if (!is_target_process()) {
        bpf_send_signal(SIGKILL);
    }
    return 0;
}
```

Figure 8: Code snippet for killing specific process via eBPF.

D1: Killing process by sending signals. As shown in Figure 8, malicious eBPF tracing programs can send a `SIGKILL` signal to kill arbitrary processes outside the container, when this process triggers any eBPF hooks. The signal is sent from the kernel, enabling it to terminate processes running under root user privileges.

D2: Abusing LSM rules to crash processes. The eBPF LSM subsystem allows users to attach eBPF programs at various LSM hooks to check permissions for files, processes, and networks. When a kernel function with an LSM hook is invoked, the kernel will execute the eBPF program at this LSM hook

to decide if the current function call is permitted. Attackers can write malicious eBPF LSM programs to set illegal access control rules to prevent a specific process from accessing the necessary resource. For instance, they can paralyze one program by disallowing it to open files, bind ports, or spawn sub-processes.

D3: Altering system call arguments or return code. Similar to the Iago [32] attack, victim processes' system call arguments can be modified by eBPF Tracepoint programs when entering or exiting the system calls. Attackers can disrupt process functionality by passing invalid system call arguments. Moreover, malicious eBPF tracing programs can exploit the kernel's error injection feature to prevent victim processes from invoking any system calls by overriding their return codes.

B.2 Information Theft Attack

```
SEC("raw_tracepoint/sys_exit")
int tp_exit(struct bpf_raw_tracepoint_args *ctx) {
    unsigned long svc;
    struct pt_regs *regs=(struct pt_regs*)(ctx->args[0]);
    bpf_probe_read(&svc, sizeof(svc), &regs->orig_ax);
    // step-1: record the fd of sensitive file
    if (svc == NR_openat) {
        save_fd_for_sensitive_file(regs);
    }
    // step-2: steal sensitive file content
    if (svc == NR_read) {
        if (is_sensitive_fd(regs)) {
            char *p = NULL; // ptr for read buf
            char buf[LOG_ENTRY_SIZE] = {0};
            int sz = 0; // read size
            bpf_probe_read(&p, sizeof(p), &regs->si);
            bpf_probe_read(&sz, sizeof(sz), &regs->ax);
            BOUNDED_LOOP (sz > 0) {
                sz -= LOG_ENTRY_SIZE;
                p += LOG_ENTRY_SIZE;
                bpf_probe_read_str(&buf, sizeof(buf), p);
                // send to user space via eBPF map
                record_data(EVENT_FILE, buf);
            }
        }
    }
}
```

Figure 9: Code snippet for stealing sensitive file via eBPF.

T1: Stealing processes' opened files. As shown in Figure 9, attackers can attache a Tracepoint eBPF program to the system call exit point inside the container and observes the read system call. This eBPF program will be triggered whenever a read system call is finished. By checking the `pid` and program name, the attackers can identify the sensitive processes both inside or outside of the container and obtain the reading buffer from the system call arguments. Then, the attackers can use the eBPF `bpf_probe_read` helper function, which allows reading arbitrary memory of both kernel space and user space, to obtain the sensitive data from the reading buffer pointers. Finally, attackers can transfer these data to their user space eBPF control program via eBPF map. Thereafter, the attackers can get the content of any opened files even if these files are isolated by the mount namespace of the container's file system, e.g., extracting the private key when it is opened by OpenSSL.

T2: Stealing kernel data structure addresses. Attackers can read the kernel data structure addresses by attaching eBPF KProbe program to various kernel functions with eBPF helpers. These kernel data structures can assist attackers to bypass the KALSR and stably obtain the kernel base address and sensitive kernel functions' addresses, such as some mostly used kernel exploitation functions [19] (e.g., `commit_cred` and `prepare_kernel_cred`). With these functions, they can construct ROP chains for privilege escalation. When using the traditional kernel exploit technologies [63], it is challenging to leak these addresses via spraying the unstable kernel heap. Note that the user space memory of the current process can also be dumped via the `bpf_probe_read` helper. Attackers can steal the secrets in memory or dump the code and stack for further exploitation, e.g., inject shellcode and construct ROP chain.

B.3 Container Escape Attacks

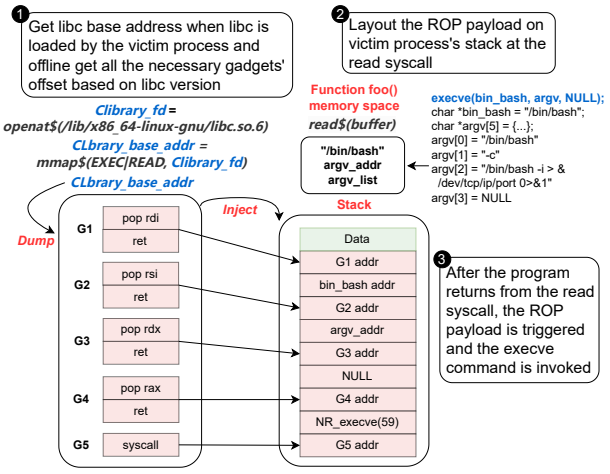

Figure 10: The workflow and stack layout of eBPF-based ROP attack.

E3: Shellcode injection during mprotect system call. Shellcode injection may still be practical under the modern Linux NX. We notice in some situations, programs need to change their memory executable/write attribute at runtime. For instance, the programs with Javascript runtime, such as NodeJS and Chrome, use Just-In-Time (JIT) compilation to generate binary code which is stored in heap and is changed to executable during runtime. The UPX parcker programs also need to change the writable memory to execution only after unpacking the real program to memory. With the help of eBPF, we can capture a timing to inject the shellcode right before the memory is changed from writable to executable. As the memory attribute is changed by the mprotect system call, we can use a eBPF tracepoint program to hook the mprotect of all

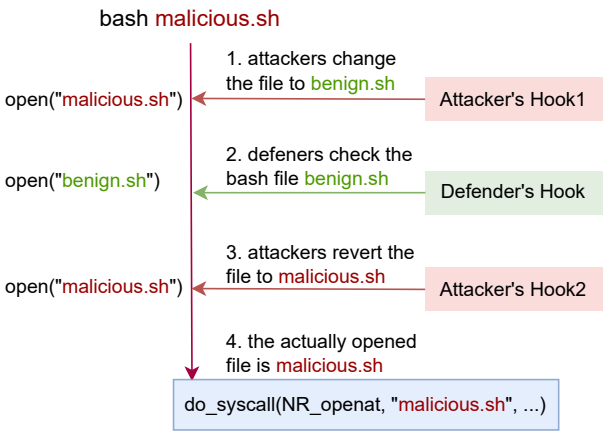

Figure 11: Attackers can revert the system calls arguments after the defender checks by placing a hook (Attacker's Hook2) after the defender's hook.

the processes and inject shellcode to them before the memory is set to readonly. If the affected process is running in the host by root user, the attackers can escape the container to execute malicious code by hijacking this process.

E4: Forging credentials to login as root via SSH. The sudo and ssh password verification need to read the access control configures (i.e., /etc/passwd, /etc/shadow, /etc/sudoers). Attackers can forge the file content by hijacking the read system call of these processes and inserting fake passwords into these files. Thus, the sudo and ssh can be deceived, and the attackers login the host or use sudo to change to root with false password. Even though the attackers already have the root permission inside the container, they still need to login to the host to obtain real root permission. Thus, this attack requires the host to have an accessible IP for ssh login.

C Bypass the Defenders' Kernel Modules

Time of check to time of use (TOCTOU). The defenders receive massive tracing messages from the kernel modules. To reduce CPU usage, the defenders usually process the messages periodically and give way to other OS tasks after finishing a scan, e.g. the *AliYunDun* starts a scan every six seconds. The log buffer (i.e., trace pipe, eBPF ringbuffer map, and perf events) may be full, and the old logs are overwritten by the new content. Attackers can launch attacks immediately after *AliYunDun* finishes reading the trace pipe and starts checking the logs. They first need to produce numerous benign behaviors to exhaust the log buffers and then perform malicious behaviors, e.g., invoking the bpf to load malicious eBPF programs. The sensitive events are ignored by the defenders due to message loss.

Hook Order Interference Attack. As shown in Figure 11, attackers can install their hooks after the defender's hook at the same place. They can inject malicious payload into a benign process by violating the system arguments after checking by the defenders. In particular, they use two hooks to conceal their own behaviors by converting the input from malicious to benign before the checking and revert it after checking. This attack is practical for most defenders who use `KProbe` or `Tracepoint` hooks. At the same hook point, attackers can install a `RawTracepoint` hook which is definitely triggered before the defenders' `KProbe` or `Tracepoint` hook and install a the same type hook with the defenders which is triggered after the defenders' hook. However, if the defenders also use `RawTracepoint` hook, attackers need first kill the defender's process to install hooks earlier than the defenders. Note that this attack can only hide the system call arguments rather than the system call itself. The defenders can still obtain the system call sequences.

D Reasons for Insecure Container Configs

We further study the root causes of using the aforementioned three insecure container configs. First, some containers require kernel-level capabilities to conduct kernel-level operations, such as mounting file systems [1,25] and modifying network settings [7]. Specifically, telegraf requires docker.sock to monitor other docker containers, dynatrace/oneagent uses privileged option to set environment credentials, and docker in docker utilizes *CAP_SYS_ADMIN* to configure the network. Second, security tools are installed inside containers to create an isolated environment and have access to the host system. For example, newrelic/nrsysmond runs New Relic's free Linux Server Monitor (LSM) via requesting privileged option and docker.sock. Third, some containers require privileges for debugging and troubleshooting issues on the host system. For instance, Cross builds in privileged docker when the target platform is Android arch_32bit because unprivileged docker doesn't support thumb instructions and toolchains, and the open big data serving engine Vespa also needs privileged docker for perf to instrument kernel surveillance.