GRANULAR CONTAINER SECURITY AND NETWORK FILTERING USING EBPF

Ruturaj Mohite

Master of Technology Thesis
June 2024

International Institute of Information Technology, Bangalore

GRANULAR CONTAINER SECURITY AND NETWORK FILTERING USING EBPF

Submitted to International Institute of Information Technology, Bangalore
in Partial Fulfillment of
the Requirements for the Award of
Master of Technology

by

Ruturaj Mohite
IMT2019518

International Institute of Information Technology, Bangalore
June 2024

Thesis Certificate

This is to certify that the thesis titled **Granular Container Security and Network Filtering Using eBPF** submitted to the International Institute of Information Technology, Bangalore, for the award of the degree of **Master of Technology** is a bona fide record of the research work done by **Ruturaj Mohite**, **IMT2019518**, under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

The thesis conforms to plagiarism guidelines and compliance as per UGC recommendations.

Prof. B. Thangaraju

Bengaluru,
The 4th of June, 2024.

GRANULAR CONTAINER SECURITY AND NETWORK FILTERING USING EBPF

Abstract

Containers have revolutionized application deployment, but security concerns remain a challenge. By attaching programs to kernel hooks, eBPF provides deep visibility into container activity. This enables fine-grained monitoring of network traffic, system calls, and resource usage. Security policies can be enforced at the kernel level, allowing for real-time intrusion detection and prevention.

The thesis examines practical applications of eBPF in container security. The thesis explores uses of eBPF across different kernel subsystems and frameworks like Linux Security Modules (LSMs), Traffic Control (TC) and Secure Computing Mode (Seccomp) to achieve granular network and system call filtering.

The thesis introduces an original method to filter container traffic based on the container and the process of origin. This approach allows for more granular control over network communication within containers. Following the principle of least privilege, the goal is to allow container processes to only talk to services that they need to talk to. Using hook points provided by the kernel and mapping packets to processes, the implementation detailed in this thesis aims to prevent malicious outbound communication from containers. The thesis also discusses challenges in building eBPF filters including portability across kernel versions, BPF CO-RE and operating filters at scale.

Acknowledgements

Firstly, I would like to thank my friends and family for being a constant source of support through this research. I would also like to thank Liz Rice from Isovalent for introducing me to the topic of this research. In many ways, my fascination with eBPF and its applications started in 2022 when Liz gave me her 'What is eBPF' book at the Open Source Summit EU. Thank you Prof. Thangaraju for advising this thesis and helping me write and publish the accompanying paper.

List of Publications

1. Enhancing Container Security with Per-Process Per-Container Egress Packet Filtering using eBPF" by R. Mohite and B. Thangaraju, will be presented at the 4th International Conference on Electrical, Computer, and Energy Technologies (ICECET 2024), Sydney, Australia, from July 25-27, 2024 (to be published)

Contents

List of Figures

List of Tables

List of Abbreviations

BCC . . . . . BPF Compiler Collection

[e]BPF . . . . . [extended] Berkeley Packet Filter

BTF . . . . . BPF Type Format

CO-RE . . . . . Compile Once - Run Everywhere

ICMP . . . . . Internet Control Message Protocol

IPC . . . . . Inter-Process Communication

IPvX . . . . . Internet Protocol Version X

LSM . . . . . Linux Security Module

MAC . . . . . Media Access Control

MTU . . . . . Maximum Transmission Unit

NAT . . . . . Network Address Translation

skb . . . . . Socket Buffer

TC . . . . . Traffic Control (Linux Subsystem)

TOCTTOU . . . . Time of Check to Time of Use

UTS . . . . . Unix Time Sharing

veth . . . . . Virtual Ethernet

CHAPTER 1

BACKGROUND

The rise of containerization has transformed application deployment, offering benefits like portability, scalability, and efficient resource utilization. However, this rapid adoption has also brought security concerns to the forefront. Traditional security solutions often struggle to adapt to the dynamic nature of containerized environments [1].

This thesis investigates the potential of eBPF as a powerful tool to enhance container security. eBPF's unique ability to attach programs to kernel hooks provides a great level of visibility into container activity. With this level of visibility, we can achieve fine-grained monitoring and filtering of network traffic, system calls, and resource usage.

This ability to see "under the hood" of containers unlocks a new level of security control. Security policies can be enforced at the kernel level, enabling real-time intrusion detection and prevention. Malicious activities such as unauthorized access, suspicious connections, and file operations can be identified and blocked before they can compromise the system.

The work in this thesis focuses on the applications of eBPF in container security. Other levels of monitoring and filtering can be achieved with orchestration tools like Kubernetes by extending the work detailed in this thesis. The thesis will set the background knowledge required for proper understanding before exploring existing methods of container security with eBPF. Finally, the thesis will propose an original method for

process level network filtering with eBPF.

1.1 The extended Berkeley Packet Filter

1.1.1 Origin of eBPF

The concept of filtering network traffic at the kernel level has been around since the introduction of Berkeley Packet Filter (BPF) in 1992. BPF acts as a special-purpose virtual machine within the Linux kernel, allowing userspace programs to define custom filtering rules for packets traversing the network stack [2]. These rules are written in a specific bytecode instruction set optimized for efficient packet processing and target specific points within the network stack. BPF's primary function lies in enabling programmatic filtering of network traffic based on predefined criteria, such as source and destination IP addresses, ports, and other packet header information.

The functionality of BPF was earlier limited to this use case of network filtering. Extended BPF (eBPF), introduced in the late 2010s, expands upon the core concept of BPF by transforming it into a general-purpose framework for executing user-defined programs at various hook points within the Linux kernel [3]. This newfound versatility empowers eBPF to go beyond network filtering and delve into a broader scope of system monitoring and security tasks. eBPF also introduces data structures that eBPF programs can use to communicate with each other and with the userspace. This also enables developers to write eBPF programs that relay information to the userspace, that otherwise userspace would not have access to. This is critical in all the observability tools that use eBPF.

1.1.2 eBPF Compiler, Loader and Verifier

eBPF programs are typically written in C and are compiled into eBPF bytecode. The eBPF library, libbpf [4], then loads the program. Since eBPF programs are executed in kernel space, it is of utmost importance that they be safe and secure to use. A developer error in an eBPF program might cause the kernel to error or lock up (for example, due to an infinite loop). A malicious user might also attach eBPF programs that leak kernel data structures which introduce a whole new attack surface. This is where the eBPF verifier [5] comes in. The verifier makes sure that the eBPF program is safe and secure to be executed in the kernel before it is attached. The verifier operates in 2 steps.

The first step checks that there are no unbounded loops or unreachable instructions. This is done by a depth-first search of the program's control flow graph. The second step is a lot more complex and it performs a variety of checks. For the second step, the verifier simulates the execution of the program by going down all paths in the control flow graph and checks that every instruction is legal. An illegal instruction (for example, adding a pointer to another pointer, reading a register that was never written to, or dereferencing a null pointer) causes the verification to fail and the eBPF program is not loaded. The verifier also checks that no kernel pointers are written to the memory in the 'secure mode'. An eBPF program is loaded in the 'secure mode' if a user without the `CAP_SYS_ADMIN` capability (a non-root user) tries to load the program. This makes sure that unprivileged users don't have access to kernel addresses.

After the eBPF program passes the verification, it is compiled to machine code by a just-in-time (JIT) compiler and attached to the kernel hook point. See Figure FC1.1.

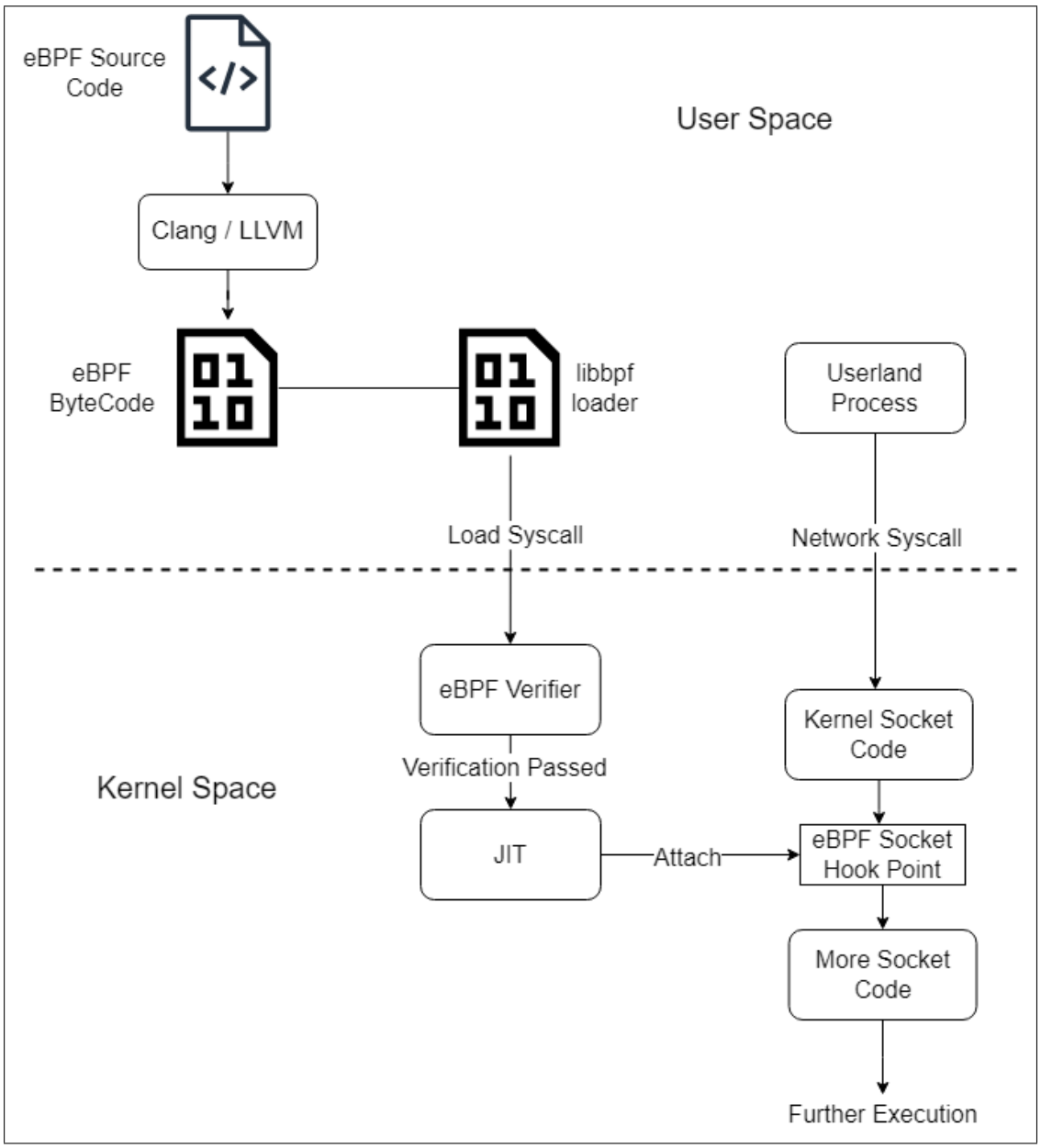


Figure FC1.1: Flow diagram of how an eBPF program is attached to kernel hook points.

1.1.3 eBPF Program Types

eBPF programs can be different types based on the hook point they attach to and the context (kernel data) that they receive. The hook points are specialized locations in the kernel control flow which trigger the attached eBPF programs. The kernel also passes specialized data structures to the program containing data about the event that triggered the program. For example, eBPF programs of type `BPF_PROG_TYPE_SOCKET_FILTER` are used for traditional packet filtering and monitoring, and are triggered inside the kernel networking function `sock_queue_rcv_skb()` (for the ingress event). They receive the `__sk_buff` data structure as context which contains the information about the packet [6].

1.2 BPF Compiler Collection

1.2.1 Convenience

Traditionally, eBPF program development necessitated writing code in low-level languages, requiring a deep understanding of the kernel and eBPF bytecode specifics. This knowledge gap often limited the use of eBPF to a select group of kernel developers. BCC [7] breaks down this barrier by introducing support for higher level languages, most notably C. This shift allows programmers familiar with C syntax to write eBPF programs without needing to delve into the intricacies of kernel development or eBPF bytecode details. Moreover, BCC also introduces support for Python for interacting with the eBPF programs from the userspace. It abstracts away libbpf under very convenient Python methods to load and attach eBPF programs. BCC also provides convenient abstractions around BPF Maps for the BPF programs to communicate with each other and with the userspace. This also does away with a lot of low-level template code that developers writing eBPF programs would have to write earlier.

Other than making it much simpler for developers to write eBPF programs, BCC comes out of the box with prewritten eBPF programs for monitoring, debugging and performance analysis.

1.2.2 Runtime Compilation

BCC packages the Clang/LLVM pair to compile eBPF programs from the source (to eBPF bytecode) on the fly from locally present kernel headers. This is a blessing and a curse at the same time.

It solves one very crucial problem. Accessing kernel data structures is no longer offset dependent at development time. In traditional eBPF programming, accessing a structure member at a specific offset, such as 8, results in compiled code that always targets that offset. Consequently, if the program runs on a different machine where, due to a differing kernel version, the corresponding data resides at offset 12, the eBPF program will incorrectly access the data. However, if the program is compiled on the target machine, it utilizes the local kernel headers, ensuring that the compiled bytecode correctly accesses the data at the appropriate offset, in this case, offset 12 instead of 8 [8].

1.2.3 Drawbacks

This however, is pretty much where the advantages of this approach end. Firstly, shipping Clang/LLVM with every eBPF program is a huge overhead. Then, compiling code at runtime is another overhead. Even with this, BCC doesn't holistically solve portability, since, it's possible that the required kernel headers are not present on the target machine. Another challenge to deal with is that any changes between kernel versions that change the name of the fields of the structures or move the fields around, will not be accommodated automatically [9]. Although, this can be solved by writing

the eBPF code in a manner that caters to all possible versions of the kernel and writing compile time `#ifdef/#else` guards to let the compiler switch between them on the target machine, it is unbelievably tedious. The final, and probably the biggest issue with the BCC approach is that since the compilation happens at runtime, a slightly different-than-expected environment on the target might throw compilation errors. So developers might have to test their eBPF code on all possible environments of the target machine. This is, again, extremely tedious [10].

1.3 eBPF Portability and CO-RE

We can now appreciate that is very important for eBPF programs to be portable across kernel versions without requiring much developer effort. However, with a new mainline kernel release every 10 weeks [11], there are bound to be changes in the in-kernel data structures across different versions.

The earliest method to ensure portability was introducing stable, ABI compliant kernel data structures and functions. The stable data structures are typically a subset of their in-kernel counterparts and are guaranteed to be stable across kernel versions even if their in-kernel counterpart changes. For example, the `__sk_buff` data structure [12, Version 6.2, `include/uapi/linux/bpf.h`, Line 5913] that exposes fields of the unstable, in-kernel `sk_buff` data structure to eBPF programs. This is good enough for relatively simple eBPF programs, but more complex programs (for example, any program that reads `task_struct` that holds information about processes) will not be helped by these since the kernel support for such stable data structures is limited.

The BPF Type Format (BTF) [13] is a big leap towards CO-RE (Compile Once - Run Everywhere). BTF, as the name suggests, is a format for describing types on the surface. However, with the help of the Clang / LLVM compiler and the eBPF verifier, it can help translate data member accesses across different kernel versions.

At compile time, Clang is able to store high level description of data accesses by the eBPF program. These are called BTF Relocations. The kernel also exposes the BTF information of its in-kernel data structures and functions. Based on these two, the eBPF verifier can translate data member accesses in the eBPF program to accesses tailored to the target system. However, this translation is only effective if data members were moved around, consolidated into a nested struct etc. A renamed field will still cause a problem.

BPF portability is an evolving topic, and there are newer solutions to the problems mentioned above like struct flavors and Kconfig variables which leave developers to deal with portability while keeping the eBPF programs compiled ahead-of-time. These will not be discussed in detail in this thesis for the sake of scope.

CHAPTER 2

KERNEL INTERNALS

2.1 Container Internals

Under the hood, Linux containers mainly use 2 important kernel features - Namespaces and Control Groups (cgroups).

2.1.1 Namespaces

Namespaces serve as the foundation for container isolation, creating the illusion of independent environments for each container, even though all containers run on the same kernel as the host. There are a lot more namespaces in Linux kernel, but let's look at a few important ones -

- **Process Namespace:** This namespace isolates the processes running within a container. Processes within a container cannot see or interact with processes in other containers or on the host system. This isolation provides a secure environment for applications to run without interference.
- **Mount Namespace:** Each container possesses its own mount namespace, allowing it to have a separate view of the mounted file systems. This ensures that files

and directories within one container remain invisible to other containers and the host system.

- **UTS Namespace:** This namespace allows containers to have their own hostname and network domain.
- **Network Namespace:** Containers can have their own network namespaces, enabling them to possess their own network interfaces and IP addresses on those interfaces.
- **IPC Namespace:** IPC namespace isolates System V IPC objects (shared memory, semaphores and message queues) and POSIX message queues between processes. This means processes in different IPC namespaces can't directly access each other's IPC resources.

When the host system starts, the pid 1 init process (systemd in case of Linux) attaches to the initial namespaces. Any process created afterwards is a child to the init process and shares its namespaces. Linux provides the `unshare` syscall to enable 'disconnecting' a process from its namespace and 'connecting' it to a new, fresh namespace. This creates a hierarchy of namespaces where a parent namespace has complete visibility of its children, but a child namespace has no visibility of its parent or siblings. Docker uses this syscall to create new mnt, uts, ipc, net and pid namespaces for new containers as evident by the syscall traced using strace -

```
unshare(CLONE_NEWNS| CLONE_NEWUTS| CLONE_NEWIPC|
        CLONE_NEWNET| CLONE_NEWPID)
```

2.1.2 Cgroups

Control groups (cgroups) allow for fine-grained management of computing resources for containers. They act as a hierarchical structure that allows privileged users to define

and enforce resource quotas for processes within a container. Containers are usually associated with a cgroup of their own. Which means, there is a one-to-one mapping between a cgroup and a container.

2.1.3 Networking in Docker Containers

The previous section talked about how containers have separate network namespaces. This means that, as far as the container is concerned, the host is a completely different machine that it needs to communicate with over a network. Similarly, any other containers running on the same host are also different machines for the container. This section briefly describes how Docker sets up network communication between containers and the host.

In the physical world, a bunch of computers are connected together by a physical bridge with physical Ethernet ports using RJ45 connectors. Virtually, containers are connected to each other (if they are on the same docker network) and to the host by a virtual bridge and virtual Ethernet interfaces. The bridge here is similar to a switch, connecting two network devices on layer 2. Docker also creates a network and adds all the devices connected to the bridge on this network and assigns them IP addresses so that they are reachable via layer 3. The interface for this network on the host machine is called `docker0` [14].

When a new container is created, a pair of veths (Virtual Ethernets) are created and registered with the kernel. One of these veths hooks into the container, and another into the Docker bridge on the host. See Figure FC2.1.

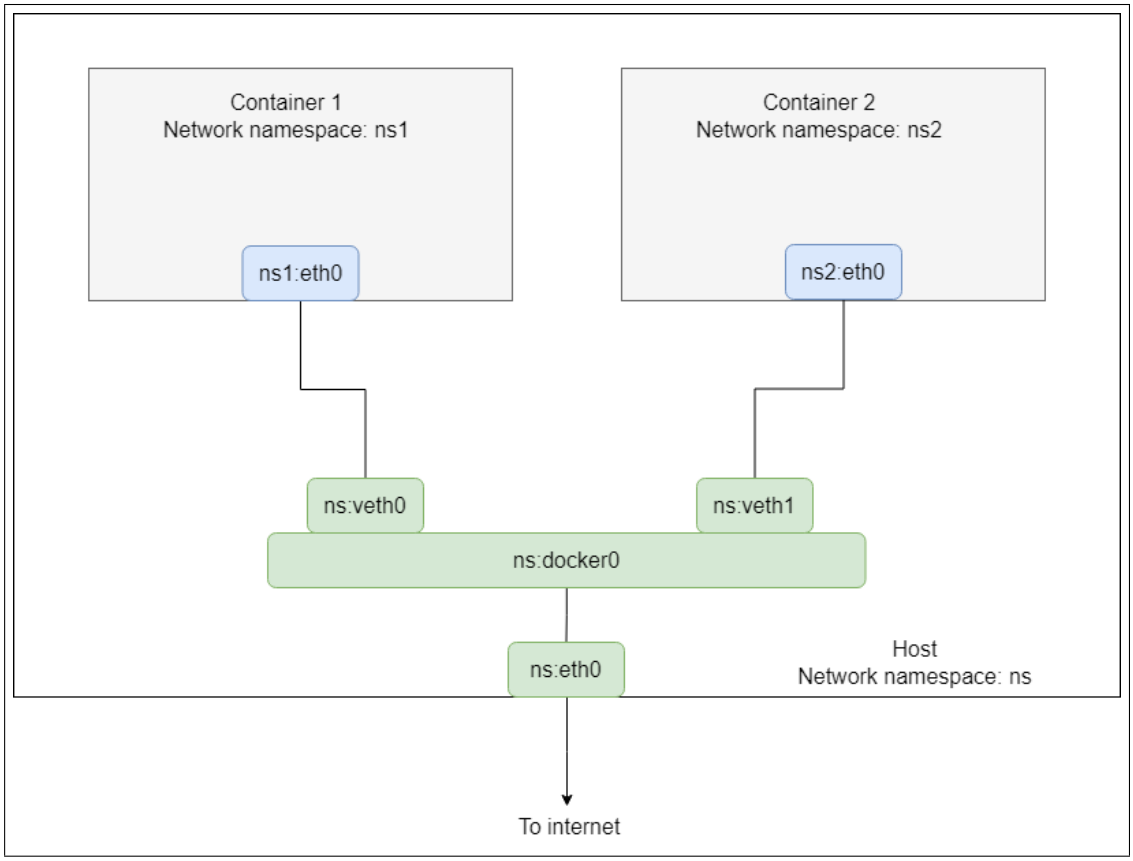


Figure FC2.1: Structure of Docker's bridge network for 2 containers and a host.

2.2 Linux Kernel Networking

In this section, we will explore how IPv4 packets go from being received at the Network Interface Card of a Linux machine, to reaching the destination machine and process and vice versa.

The Linux networking flow can be broken into two flows - Ingress (incoming packets) and Egress (outgoing packets).

2.2.1 Ingress Flow

When a packet is received at the NIC, the kernel calls the `NET_RX_SOFTIRQ`. This is a softirq, similar to an IRQ (Interrupt Request) which is a request sent out by a hardware device to the CPU to service a hardware interrupt. The service routine for this softirq eventually calls `ip_rcv()` [15] [16].

`ip_rcv()` checks that the packet that was received was intended for the current host unless the host is in promiscuous mode (promiscuous mode allows a host to intercept all traffic flowing through it, regardless of whether the traffic was intended for the host). This function then triggers the `NF_INET_PREROUTING` hook and calls `ip_rcv_finish()`. `ip_rcv_finish()` then populates the route specific fields in the socket buffer (skb) structure of the packet and calls one of the three functions depending on the destination of the packet -

1. `ip_local_deliver()` is called if the current host is the destination for the packet. This function sends the packet up the networking stack for layer 4 processing (even if the packet is raw and has no layer 4 header information) [17]. This function is also responsible for defragmentation of packets.

2. `ip_forward()` is called if the packet is not intended for the current host, but the

current host is able to route it its destination. It checks that the packet still has a non-zero TTL before forwarding. If the packet's TTL hits zero, an ICMP control message for the same is sent back to the source host.

3. `ip_mr_input()` is called if the packet is a multicast packet, which further calls `ip_local_deliver()` or `ip_mr_forward()` depending on the destination host.

Figure FC2.2 shows the ingress flow.

2.2.2 Egress Flow

When a process wants to send packets over the network, it first writes to a socket file. `__sock_sendmsg()` then checks permissions and forwards this data to Layer 4 protocol specific functions. For TCP, the data is forwarded to `tcp_sendmsg()` [15] [16].

`tcp_sendmsg()` copies this data from userspace to the kernel-space socket buffer (skb) data structure. Once populated, the skb is sent further with `tcp_transmit_skb()`. This function builds the Layer 4 header for the packet, clones the packet and sends it for Layer 3 processing via the `ip_queue_xmit()` function [17].

`ip_queue_xmit()` is responsible for populating the routing information in the skb in the `skb->dst` field. If the routing information is already present in the skb, this step is skipped. Otherwise, `__ip_route_output_key()` function is called to check for an available route in the routing cache. If the cache misses, `ip_route_output_slow()` is called for a full lookup in the routing table. If no route to the destination is found despite this, the packet is dropped. If a route is found, `ip_local_out()` is called that sets the packet length and checksum, and triggers the `NF_INET_LOCAL_OUT` netfilter. If the packet passes the netfilter checks, it is then passed to the `dst_output()` function which (in case of a unicast packet) calls `ip_output`. This is also the function where ingress packets that are to be routed further end up. This function further triggers another

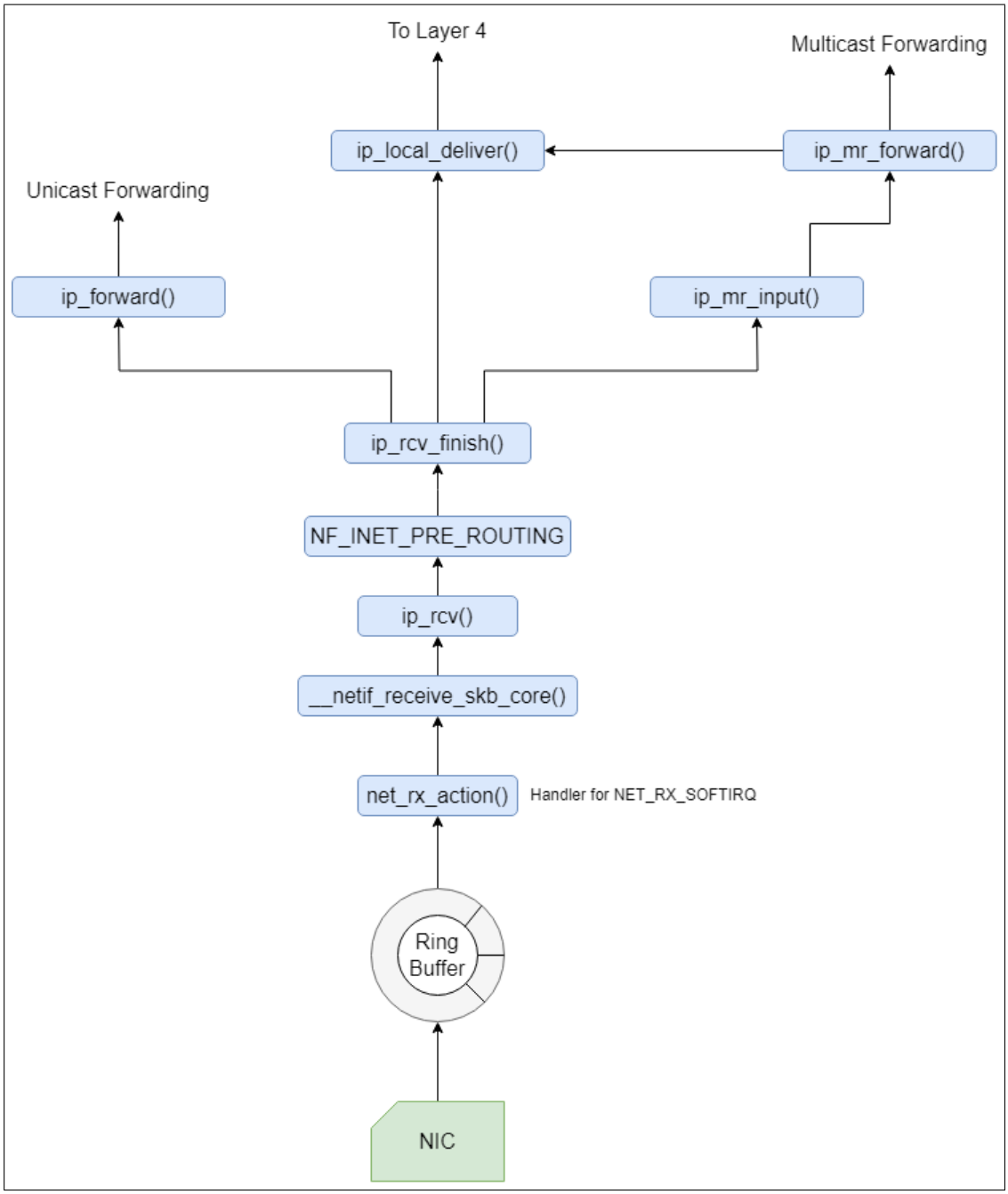


Figure FC2.2: Ingress flow for IPv4 packets

netfilter hook and calls `ip_finish_output()`.

At this point, the skb contains all necessary information about the source, destination and the route of the packet. `ip_finish_output()` function is responsible for fragmenting packets that exceed MTU. It also calls the `BPF_CGROUP_RUN_PROG_INET_EGRESS` macro which triggers any eBPF programs attached to the `CGROUP_SKB` egress hookpoints [12, Version 6.2, `net/ipv4/ip_output.c`, Line 313].

Finally, the packet reaches the `ip_finish_output2()` function that looks up the MAC address of the neighbour in the packet's route and populates the address in the skb and sends it for Layer 2 processing.

Figure FC2.3 shows the egress flow.

2.3 System Calls and Seccomp

System calls are the primary mechanism through which user-space applications interact with the kernel. They provide a controlled interface for accessing low-level system resources such as file systems, networking, and process control. Each system call triggers a context switch from user mode to kernel mode, allowing the operating system to perform privileged operations on behalf of the application. While essential, system calls also present potential security risks, as malicious programs can exploit them to gain unauthorized access to system resources or escalate privileges.

To mitigate these risks, the Linux kernel includes a feature called seccomp (secure computing mode). Seccomp provides a mechanism for restricting the set of system calls that a process can make, significantly reducing the attack surface available to malicious code. When a process enters seccomp mode, it can only make a predefined set of system calls, and any attempt to make a prohibited system call results in the kernel terminating the process.

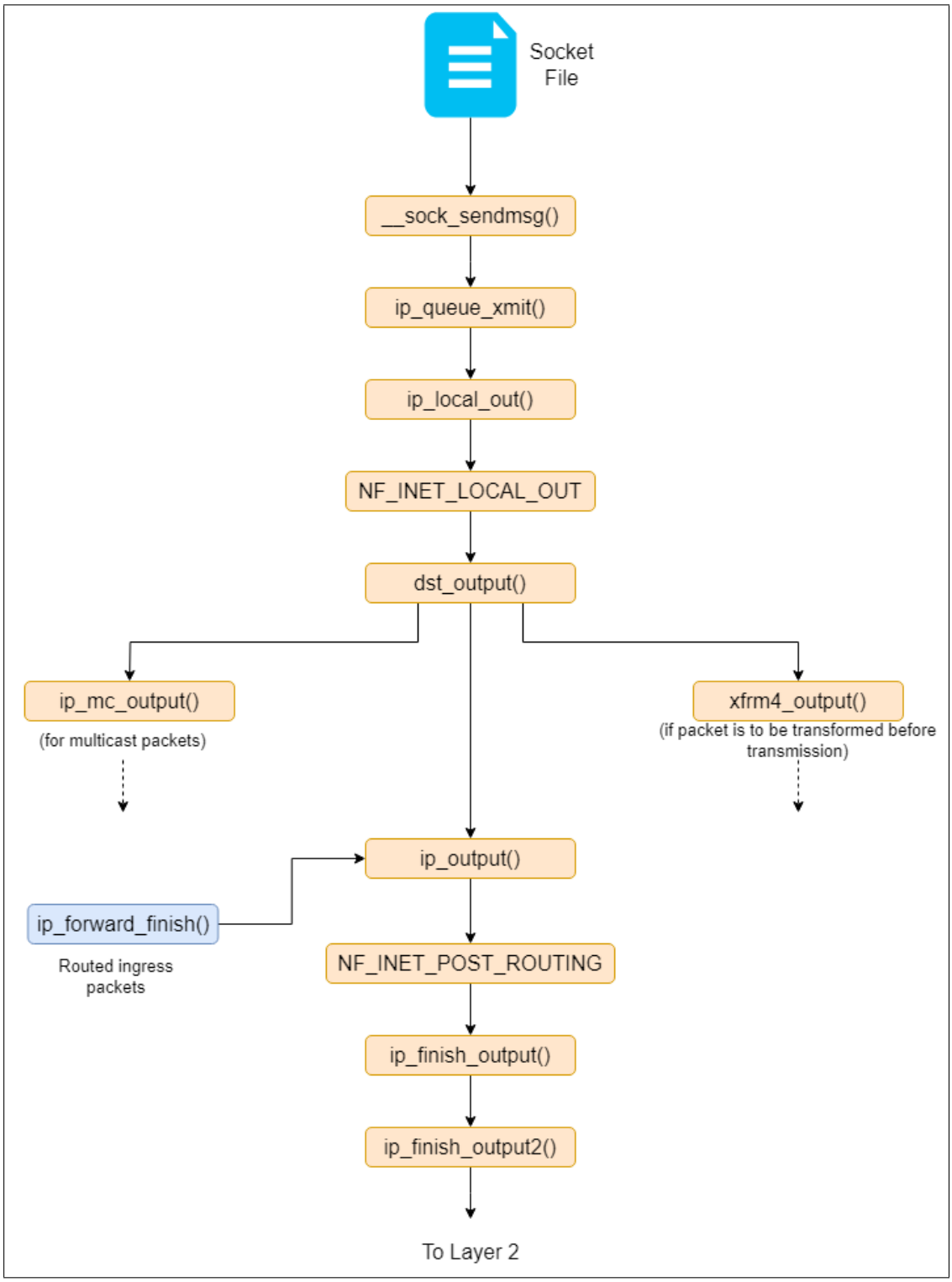


Figure FC2.3: Egress flow for TCP IPv4 packets

2.4 Kprobes and Tracepoints

Kprobes (Kernel Probes) [18] are a powerful debugging and performance monitoring tool in the Linux kernel. They allow developers to dynamically insert probes into running kernel code, which can be used to collect diagnostic information, trace kernel execution, and monitor system performance without needing to modify the kernel source code or reboot the system.

Kprobes can be inserted at almost any location within the kernel, including at the entry and exit points of functions, making them extremely flexible. When a probe is hit, it can execute a user-defined handler function that can perform tasks such as logging, modifying registers, or collecting data.

Tracepoints are points in the kernel code at key locations, providing predefined hooks that can be used for tracing and logging events. Unlike Kprobes, which are dynamic and can be placed almost anywhere, tracepoints are predefined and placed at strategic points in the kernel source code by kernel developers. This makes them more stable and less likely to cause performance issues compared to dynamically inserted probes.

CHAPTER 3

NETWORK AND SYSTEM CALL FILTERING FOR CONTAINERS

In this chapter, we will take a look at current studies that focus on container security. Particularly, the ones that focus on system call and network filtering.

3.1 Seccomp Notify - Making Seccomp Dynamic

3.1.1 Motivation

The classic cBPF way of writing Seccomp filters serves a very rudimentary purpose. With classic Seccomp policies, syscalls can be allowed - in which case, the kernel performs the syscall and reports the result to the caller process, or denied - in which case, kernel is able to return a user-specified error code. Therefore, once a Seccomp filter is loaded, the policy defined by the filter is used to filter all syscalls. It is not possible to make a case-by-case decision for every syscall. Moreover, it is also not possible to allow another process to make the syscall on behalf of the caller process.

The ability to let another process make syscalls on behalf of the calling process ties into the case-by-case decision making. This is useful if a more privileged process is allowed to make or deny syscalls made by a lesser privileged process. For example, a

container manager could be allowed to make or deny syscalls made by the containers it created. This would make for a more dynamic container system call security filter as the filter logic can be more nuanced with the container manager having a borader and real-time view of the system.

3.1.2 Implementation

To enable Seccomp notify [19], the container must set the `SECCOMP_RET_USER_NOTIF` flag on its Seccomp filter. Upon setting this flag, the kernel returns a file descriptor to the calling process (the container) when the filter is loaded. The container can then hand over this file descriptor to the container manager. This file descriptor is an object shared between the kernel and the container manager, which facilitates the interaction between the two to share metadata about the syscall.

With Seccomp notify enabled, the kernel, instead of allowing or denying the syscall made by the container, notifies the container manager using the notify file descriptor. While the container manager processes the syscall, the kernel blocks the container process that initiated the syscall.

To read notify events on the fd, the container manager polls the notify fd with an `epoll(7)` event loop and waits for the fd to become readable, i.e. for the kernel to write any syscall metadata on it. This syscall metadata is written in a struct `seccomp_notif`. Once the fd becomes readable, the container manager reads the syscall metadata (like syscall name, arguments) and copies the syscall arguments to a local buffer and checks the syscall with the policy. It then either denies the syscall (in which case, it can return any error code or return no error code and pretend as if the syscall happened) or makes the syscall on the behalf of the container. If the syscall is successfully made, the container manager wraps the response in a `seccomp_notif_resp` struct. Alternatively, the response can also contain the flag `SECCOMP_USER_NOTIF_FLAG_CONTINUE` which

instructs kernel to perform the syscall, an equivalent of allowing the syscall in classic Seccomp.

3.1.3 Drawbacks

Seccomp notify has a few major drawbacks. Further studies to address these drawbacks will be explored in a later section.

1. **Requires knowledge of the Linux source code:** Since the arguments to the syscall are passed in an array in the `seccomp_notif` data structure, the container manager is expected to know the layout of the arguments of the specific syscall. This can only be found in the Linux source code as the man pages usually list the libc wrapper provided by the kernel rather than actual syscall.

2. **No pointer arguments:** If any of the syscall arguments are userspace pointers (which is the case for many syscalls), it is unsafe to dereference them for policy verification due to the potential for a time-of-check-to-time-of-use (TOCTTOU) race condition. It is possible that before the check, the pointers point to arguments that pass the Seccomp check, the container manager then sends back the `SECCOMP_USER_NOTIF_FLAG_CONTINUE` flag, instructing the kernel to execute the syscall. Between the time the response is sent from the container manager and the kernel reads it from the fd, an attacker might change the values at the pointers, now pointing to illegal arguments, which are used when the syscall is executed.

3.2 Seccomp eBPF

3.2.1 Motivation and Implementation

The goal of Seccomp eBPF [20] is to allow implementation of granular system call policies while taking care of most of the shortcomings of Seccomp notify. Since eBPF provides a way to write complex code right inside the kernel, Seccomp eBPF introduces a new eBPF program type that lets users write complex logic to filter syscalls. This completely eliminates the need for a higher privileged process that in Seccomp notify, contained bulk of the filtering logic. Effectively, this 'more privileged process' is now just the kernel instead of a container manager. The main goals that the work focusses on are -

1. Providing safe userspace memory access. As discussed in the section about Seccomp notify, a lot of Linux syscalls use userspace pointers as arguments. In Seccomp notify, it was unsafe to filter such syscalls because of TOCTTOU race conditions while accessing userspace pointer values. Seccomp eBPF provides safe userspace memory access by copying values on the userspace memory into kernel memory and making the area on the userspace memory write-protected.

2. Allowing stateful filtering policies. With classic Seccomp, it is not possible to implement stateful policies like rate limiting (where you have to store the count of syscalls made). While with Seccomp notify, the state could be maintained with the container manager, eBPF allows for state to maintained in eBPF maps.

3. Flexible and granular filter policies. Just like Seccomp notify, the goal of Seccomp eBPF is to allow for granular and logically complex syscall filters. eBPF allows for writing logically complex code inside the kernel with relatively low risk (due to the verifier). This allows for the filter itself to contain complex logic rather than delegating

the logic to an external process.

4. Serialization of system calls. The Linux kernel has had increasing number of reports of bugs due to race conditions [21]. Although these bugs are eventually resolved, it may take considerable time for kernel developers to address them. In environments where these bugs have a high impact, this delay may be unacceptable. Serialization of syscalls allows users to make sure that two syscalls vulnerable to race conditions [22] do not execute at the same time. Seccomp eBPF provides a helper function `void bpf_wait_syscall(int curr_nr, int target_nr)` which waits for the target syscall to finish before executing the current syscall.

3.3 Process Level Network Security

3.3.1 Motivation

With the growth of containerization and orchestration, it is a common practice to run multiple services on the same host machine. Orchestration tools like Kubernetes also make deployment of services completely host agnostic. Because of this, it is not an easy job to define network filtering policies on hosts since the developers do not know what services are going to be run on a particular host.

Therefore, it is important for developers to be able to define network filtering policies per container. With eBPF's granular control, it is also possible to map network activity to the processes that generated it. Fournier's [23] work explores methods for implementing process and container level network filtering using eBPF.

3.3.2 Implementation

On the surface, the filter is an eBPF program hooked to the Traffic Control (TC) [24] subsystem. TC eBPF programs allow for filtering packets based on their L2/L3 header information and also allow resolving container NAT at runtime. A program hooked to TC subsystem gets the `__sk_buff` structure as context and can extract header information from there. Based on the return value from the filter, the packet can be forwarded or dropped by the kernel.

Mapping packets to their processes served as a major challenge as process context (such as functions like `bpf_get_current_pid_tgid()` or `bpf_get_current_comm` are not available to eBPF programs hooked to the TC subsystem. To map packets to their processes, Fournier uses a kprobe to the Linux Security Module (LSM) [25] function `security_sk_classify_flow()` after trying multiple approaches. This works fine for IPv4 and IPv6 packets over a layer 4 protocol such as TCP or UDP, but is not able to map packets without layer 4 header information (such as ICMP packets). This is logical for ingress ICMP packets as they are not directed at a specific process but are served by the kernel itself. However, it should theoretically be possible to map egress ICMP traffic to the originating process.

In the next chapter, we propose a novel way of implementing such a filter which primarily makes use of a different type of eBPF program which is specialized for containers.

CHAPTER 4

NOVEL METHOD FOR PROCESS-CONTAINER LEVEL NETWORK FILTERING

4.1 Motivation

The goal of this section is to implement a per-process per-container network filter using eBPF that filters network traffic sent out by processes within containers according to a user policy. This type of filtering is a crucial safeguard against containers that could be potentially compromised. A compromised container can communicate with the attacker's server and be utilized as part of a DDoS botnet, to exfiltrate application data, or to hijack computing resources.

Fournier's work [23] had limitations with filtering ICMP packets (or raw packets without layer 4 information) which made sense for ingress packets but inadvertently ended up limiting the filtering of egress packets. It is essential to filter these packets when they are transmitted by a process to prevent attacks like ICMP tunneling [26]. We focus our efforts on egress packets and propose a method that does not have any such limitation.

This chapter is referenced from the thesis author's original work. The source code for the program is open source and publicly available [27].

4.2 Design

4.2.1 Overview

We implemented an eBPF-based packet filtering system to manage network traffic from containers, filtering packets based on Layer 3 (IP) and Layer 4 (Transport) headers and process of origin according to user-defined rules.

Our primary eBPF program attaches to the cgroup file descriptor of a container and operates on the socket buffer structure. This program compares packet headers against rules stored in BPF arrays. A matching packet is forwarded, while a non-matching packet is dropped.

To identify the process that created a packet, we use kprobes to probe into functions that are called in the kernel network code flow and relay the process information back to the filter.

We also probe into kernel functions that register network devices to monitor creation of new containers at runtime and attach the eBPF filter to the new containers.

Refer to Figure FC4.1 for the overview of the flow of the program.

4.3 Implementation

4.3.1 Packet Filter

We use an eBPF program of type `BPF_PROG_TYPE_CGROUP_SKB` to filter packets. This program is attached to a container's cgroup file descriptor and operates within the context of the socket buffer structure that contains network packet information originat-

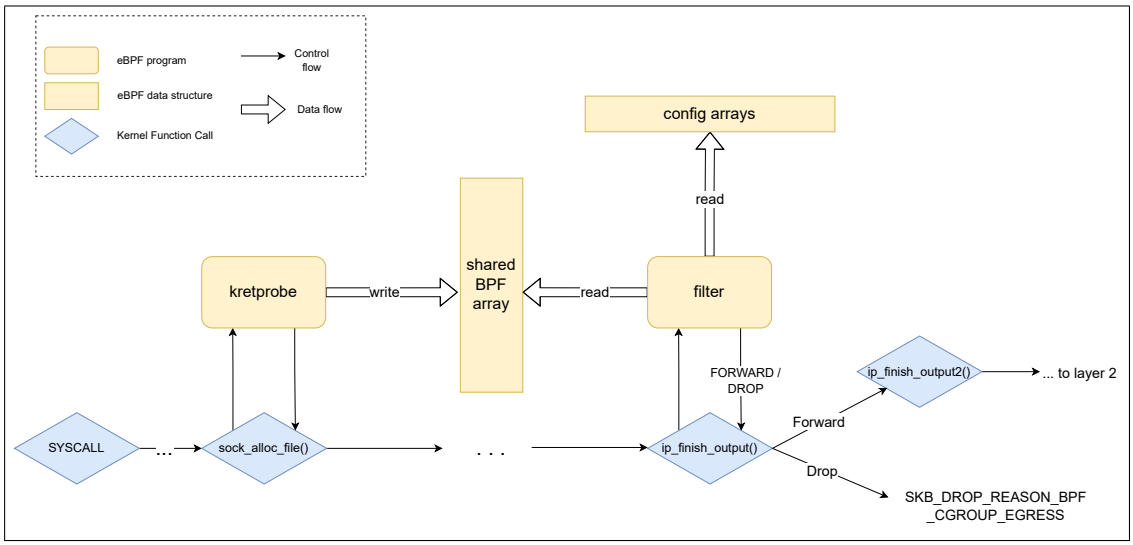


Figure FC4.1: Flow of the filter program with respect to kernel function calls.

ing from the container. For egress traffic, the program is invoked in the `ip_finish_output()` function in the kernel [6].

Recall from Chapter 2 that `ip_finish_output()` function is responsible for fragmenting packets that exceed the MTU in size and is one of the final steps for the outgoing packet before it's processed at layer 2. This function includes a call to the `BPF_CGROUP_RUN_PROG_INET_EGRESS` macro, which executes the eBPF programs attached to the egress hook.

If the eBPF program returns 0, the packet is dropped with the reason `SKB_DROP_REASON_BPF_CGROUP_EGRESS`. A non-zero return value leads to the call of `__ip_finish_output()`, which performs fragmentation and allows the packet to pass through [12, Version 6.2, `net/ipv4/ip_output.c`, Line 340].

Our eBPF program accepts user-defined specifications via eBPF maps. It matches incoming packets' L3 and L4 headers against these specifications to determine whether to forward or drop the packet (refer Appendix A).

4.3.2 Packet to Process Mapping

We have established packet filtering based on L3 and L4 headers for containers, but we also need to identify the process that created each packet. The `bpf_get_current_comm()` function retrieves the command of the current process, but it cannot be called from a `BPF_PROG_TYPE_CGROUP_SKB` program as it does not run in process context.

To overcome this, we use a separate eBPF program with access to the process context. We select an identifier accessible to both programs and store this identifier and the process command in a shared map. This enables the filter program to access the process command indirectly.

We attach a kretprobe to the `sock_alloc_file()` function, which allocates and returns a file structure representing a socket [28]. By capturing the pointer to this file structure at the function's return, we use it as a key in our map.

4.3.3 Container Detection

Our `BPF_PROG_TYPE_CGROUP_SKB` program attaches to a cgroup file descriptor. To automatically attach our filter program to new containers, we must detect their creation and obtain their cgroup file descriptors.

As in Fournier's work [23], containers can be detected at runtime using kprobes. Kprobes, similar to kretprobes, execute code at the beginning of a function call, providing access to the function's arguments. By attaching kprobes to functions called during a container's network interface registration, we can obtain the interface index (ifindex) of the veth interface. We then use the ifindex to find the container's ID and the cgroup file descriptor path.

We attach our kprobe to the `register_netdevice()` function, which registers net-

work devices and can access the device's ifindex [29] [28]. Given that this function is invoked twice for each container—once for the container's veth and another time for the host's veth—a state machine is used within the kprobe to specifically capture the registration of the container's veth.

Similarly, a kprobe on `unregister_netdevice_queue()` captures container shutdown events to detach the filter from the container.

The program pushes the ifindex and event (container start or stop) to a queue shared with a userspace agent. The agent then identifies the container ID using the ifindex.

4.3.4 Userspace Agent

The userspace program performs the following tasks:

- Ingests the filter allow-list.
- Listens for events from the container detection program and attaches the filter to containers.
- Translates container ifindexes to container IDs.

The allow-list is specified in JSON format as specified in Listing 1. When ifprobe detects a new event, the userspace program reads the ifindex and event type, then finds the container ID using the ifindex.

Using the container ID, the agent retrieves the cgroup path and attaches or detaches the filter program to/from the container's cgroup.

```
[
  {
    "container_name": "curler",
    "policy": [
      {
        "process": "curl",
        "allow": [
          {
            "cidr4": "1.1.1.0/24",
            "ports": [80, 443]
          }
    }
]
```

Listing 1: Allow-list JSON format (some trailing brackets are omitted for clarity)

```
/ # curl 1.1.1.1:3000
curl: (28) Failed to connect to 1.1.1.1 port 3000 after 129410 ms:
 ↪  Operation timed out
```

Listing 2: attempting to curl a non-allow-listed service

4.4 Result

When a process running in a target container makes a network request to an allowlisted service, the request proceeds without issue. However, when a non-allowlisted service is requested, the kernel returns an `EPERM` causing the request to fail. For a TCP request, this triggers retransmissons which all fail and the request finally times out. For an ICMP request, `EPERM` is propagated to the requesting process. A detailed table of behaviours is provided in Appendix B.

Listings 2 - 4 demonstrate the behavior of processes in the container when making allowed and non-allowed network requests.

```
/ # ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
ping: sendto: Operation not permitted
```

Listing 3: attempting to ping a non-allow-listed IP

```
/ # ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1): 56 data bytes
64 bytes from 1.1.1.1: seq=0 ttl=53 time=5.264 ms
64 bytes from 1.1.1.1: seq=1 ttl=53 time=14.970 ms
^C
--- 1.1.1.1 ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max = 5.264/10.117/14.970 ms
```

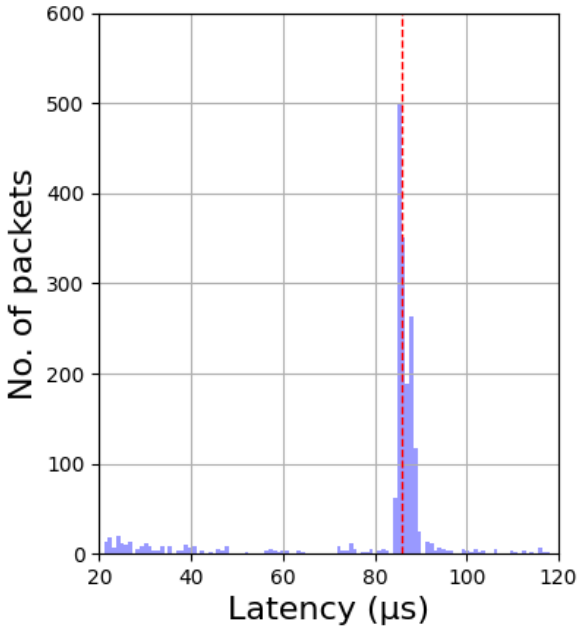
Listing 4: attempting to ping an allow-listed IP

4.5 Performance Analysis

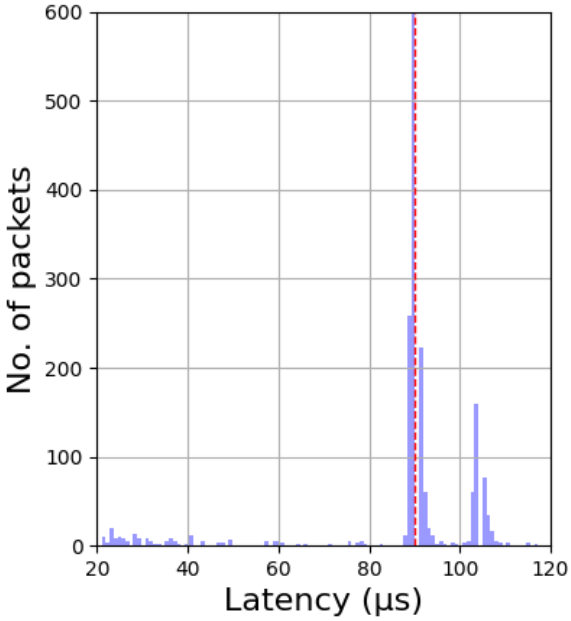
We measured the performance overhead by sending ping requests to an allow-listed IP (127.0.0.1) at 100ms intervals and recording round-trip times for different filter rule counts. The median latency increased by approximately 4µs upon enabling the filter, with an insignificant further increase ($\tilde{1}$µs) when the rule count rose from 20 to 100. This indicates acceptable overhead and good scalability compared to iptables and nftables, which degrade more with increased rule counts [30].

The experiment was conducted on a device with an Intel Core i7-8750H @ 2.2 GHz and 16 GB of RAM running Ubuntu 22.04.2 LTS with Linux 6.2.0-39-generic.

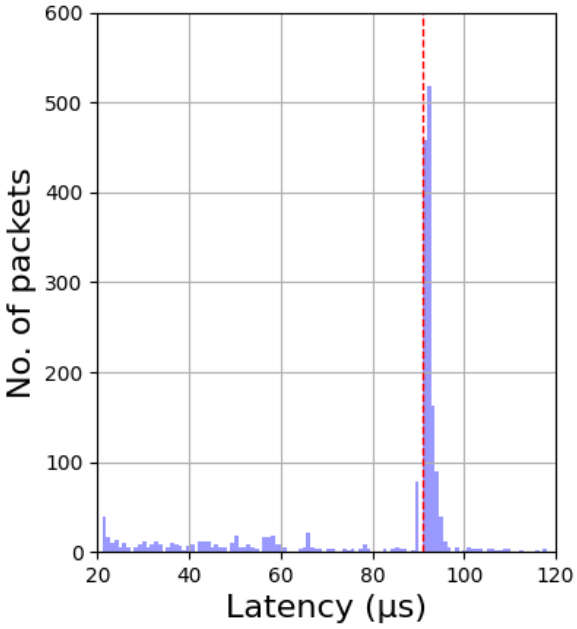
Plots in Figure FC4.2 show the distribution of round-trip times for 2000 ICMP packets with different rule counts.



(a) No filter (baseline)



(b) Filter enabled with 20 rules



(c) Filter enabled with 100 rules

Figure FC4.2: Distribution of round-trip times for 2000 ICMP packets for increasing rule counts. Red dotted lines mark the medians of the distributions. The medians are a) 86μs, b) 90μs, c) 91μs

CHAPTER 5

CONCLUSIONS

The goal of this work was to explore applications of eBPF for container security. As the use of containers in production systems grows, traditional security solutions are not able to provide the required granularity of policies that developers would want. The work explored 2 applications very crucial to container security - system call filtering and network filtering.

We explored how eBPF can aid Seccomp for filtering system calls [20] by providing the much needed expressiveness in the filter policies while leveraging the power of in-kernel filtering. The impact of eBPF was highlighted in this study by exploring an alternate solution with Seccomp Notify [19] that does not use eBPF. We studied the drawbacks of this solution and how eBPF is able to solve a lot of the issues with Seccomp Notify.

We then explored container network security starting with the work of Fournier [23]. We examined how eBPF provides process-level granularity for container network security and analyzed the methods Fournier employs to achieve this granularity using TC programs and kprobes into LSM functions. We also study the drawbacks (and their impact) of the solution proposed by Fournier and proposed an original method to fix the important drawbacks.

In our original method, we limit the scope of network filtering to outgoing packets

and explore the use of cgroup specific eBPF program to filter outgoing packets specific to processes within the target containers. We also use kprobes in the kernel networking functions to map packets back to their processes, achieving the desired process level granularity.

Finally, we look at the performance implications of such a filter and conclude that the performance overhead of the eBPF network filter is acceptably low and does not increase significantly with increase in number of filter rules.

5.1 Future Work

While the implementation presented in this work establishes a basis for enhanced container security, several critical considerations remain before it can be fully production-ready.

5.1.1 Kubernetes Integration

Currently, the daemon that runs the ifprobe and attaches the filter to containers needs to be manually started on each target machine. To improve deployment efficiency, future work should focus on integrating this process with Kubernetes, which inherently supports machine-agnostic deployment. Utilizing Kubernetes components such as DaemonSets will enable engineers to deploy the filter seamlessly across Kubernetes clusters.

5.1.2 BPF CO-RE

As discussed in Chapter 1, CO-RE considerations in eBPF programs make them portable across kernel environments and are crucial for packaging eBPF programs to

run on different target machines. Without CO-RE considerations, eBPF programs have to be developed with target machine in mind and are specific to the environments they were developed for.

In this work, we utilized BCC, which packages a compiler collection that compiles BPF source code at runtime on the target host. This approach has performance drawbacks due to the need for runtime compilation and relies on the presence of appropriate kernel headers on the target host. As a result, our current BPF code is tied to the specific kernel version and the environment of the target host.

Bibliography

[1] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Containers' security: Issues, challenges, and road ahead. *IEEE Access*, PP:1–1, 04 2019. doi:10.1109/ACCESS.2019.2911732.

[2] The Kernel Development Community. Bpf documentation. Retrieved from https://docs.kernel.org/bpf. Accessed May 28 2024.

[3] The Kernel Development Community. Classic bpf vs ebpf. Retrieved from https://www.kernel.org/doc/html/v6.2/bpf/classic_vs_extended.html. Accessed May 28 2024.

[4] The Kernel Development Community. libbpf overview. Retrieved from https://docs.kernel.org/bpf/libbpf/libbpf_overview.html. Accessed May 28 2024.

[5] The Kernel Development Community. ebpf verifier. Retrieved from https://docs.kernel.org/bpf/verifier.html. Accessed May 28 2024.

[6] Alan Maguire. Bpf: A tour of program types, 2019. Retrieved from https://blogs.oracle.com/linux/post/bpf-a-tour-of-program-types. Accessed May 28 2024.

[7] IO Visor Project. Bpf compiler collection. GitHub Repository. Retrieved from https://github.com/iovisor/bcc. Accessed May 28 2024.

[8] Aniket Bhattacharyea. A practical guide to btf (bpf type format), 2021. Retrieved from https://web.archive.org/web/20230531183204/https://www.airplane.dev/blog/btf-bpf-type-format. Accessed May 28 2024.

[9] Wenbo Zhang. Why we switched from bcc to libbpf for linux bpf performance analysis, 2020. Retrieved from https://www.pingcap.com/blog/why-we-switched-from-bcc-to-libbpf-for-linux-bpf-performance-analysis. Accessed May 28 2024.

[10] Andrii Nakryiko. Bpf portability and co-re, 2020. Retrieved from https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html. Accessed May 28 2024.

[11] The Kernel Development Community. Linux kernel releases. Retrieved from https://www.kernel.org/category/releases.html. Accessed May 28 2024.

[12] Linus Torvalds. Linux [operating system]. Retrieved from https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git. Accessed May 28 2024.

[13] The Kernel Development Community. Bpf type format (btf). Retrieved from https://www.kernel.org/doc/html/latest/bpf/btf.html. Accessed May 28 2024.

[14] Kun Suo, Yong Zhao, Wei Chen, and Jia Rao. An analysis and empirical study of container networks. *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 189–197, 2018. doi:10.1109/INFOCOM.2018.8485865.

[15] Fida Ullah Khattak. Ip layer implementatoin of linux kernel stack. Retrieved from https://wiki.aalto.fi/download/attachments/70789059/linux-kernel-ip.pdf. Accessed May 28 2024.

[16] René Serral and Marisa Gil. A linux networking study. *SIGOPS Oper. Syst. Rev.*, 38(3):1–11, jul 2004. doi:10.1145/1035834.1035835.

[17] The Linux Foundation. Networking kernel flow, 2023. Retrieved from https://wiki.linuxfoundation.org/networking/kernel_flow. Accessed May 28 2024.

[18] The Kernel Development Community. Kernel probes (kprobes). Retrieved from https://docs.kernel.org/trace/kprobes.html. Accessed May 28 2024.

[19] Christian Brauner. Seccomp notify – new frontiers in unprivileged container development. Retrieved from https://people.kernel.org/brauner/the-seccomp-notifier-new-frontiers-in-unprivileged-container-development. Accessed May 28 2024.

[20] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. Programmable system call security with ebpf. 02 2023. doi:10.48550/arXiv.2302.10366.

[21] Sishuai Gong, Deniz Altinbüken, Pedro Fonseca, and Petros Maniatis. Snowboard: Finding kernel concurrency bugs through systematic inter-thread communication analysis. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 66–83, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3477132.3483549.

[22] Robert N. M. Watson. Exploiting concurrency vulnerabilities in system call wrappers. In *First USENIX Workshop on Offensive Technologies (WOOT 07)*, Boston, MA, August 2007. USENIX Association. doi:10.5555/1323276.1323278.

[23] Guillaume Fournier. Process level network security monitoring and enforcement with ebpf. *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, 2020. url:https://actes.sstic.org/SSTIC20/sstic-2020-actes.pdf.

[24] Martin A. Brown. Traffic control howto - the linux documentation project, 2006. Retrieved from https://tldp.org/HOWTO/pdf/Traffic-Control-HOWTO.pdf. Accessed May 28 2024.

[25] The Kernel Development Community. Linux security module usage - linux kernel documentation. Retrieved from https://www.kernel.org/doc/html/v6.2/admin-guide/LSM/index.html. Accessed May 28 2024.

[26] Abhishek Singh, Ola Nordström, Chenghuai Lu, and André Santos. Malicious icmp tunneling: Defense against the vulnerability. volume 2727, pages 226–235, 07 2003. doi:10.1007/3-540-45067-X_20.

[27] Ruturaj Mohite. Contain - bpf based per-process per-container egress filter. GitHub Repository. Retrieved from https://github.com/gr455/contain. Accessed May 28 2024.

[28] The Kernel Development Community. The linux kernel (version 6.2) documentation, 2023. Retrieved from https://www.kernel.org/doc/html/v6.2/networking/kapi.html. Accessed May 28 2024.

[29] Alessandro Rubini. *Linux Device Drivers*. O'Reilly Media, 1998.

[30] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 01, pages 209–217, 2018. doi:10.1109/ITC30.2018.00039.

APPENDIX A

APPENDIX: EBPF FILTER SOURCE CODE

This appendix contains source code for the main eBPF filter program. Some unimportant parts of the code have been omitted. The full source code is open source and available at https://github.com/gr455/contain.

A.1 Network filter

```
// bpf/cgroup_sock_filter.c
#include <net/sock.h>
#include <net/inet_sock.h>
#include <linux/ip.h>

#define SOCK_DROP 0
#define SOCK_PASS 1

#define PROC_EXECNAME_MAX 16
#define EXECNAMES_COUNT_MAX 10
#define IP_COUNT_PER_EXECNAME_MAX 10
#define PORT_COUNT_PER_IP_MAX 5
#define TOTSIZE ( EXECNAMES_COUNT_MAX * IP_COUNT_PER_EXECNAME_MAX *
    PORT_COUNT_PER_IP_MAX )
```

```
/**
 *
 * This BPF file consists of 2 programs
 *  - A PROG_TYPE_CGROUP_SKB to forward or drop packets based on allow list
 *  - A kretprobe to map a process to socket file struct
 *
 * PROG_TYPE_CGROUP_SKB is used to forward or drop packets through the
    ip_finish_output() function.
 * A BPF handler function attached to CGROUP_SKB is expected to return 1
    (SOCK_PASS) to forward.
 * Any other return value will result in an EPERM propagating an EPERM to
    ip_finish_output(). This
 * BPF program, however does not get process context and cannot access process
    comm.
 *
 * Kretprobe pushes a map between process comm and file struct. This file
    struct can be accessed
 * through the sock object in the CGROUP_SKB BPF program. So this map helps
    CGROUP_SKB to access
 * process comm.
 *
 * */

struct in6_addr_u64 {
    __u64 addr_hi;
    __u64 addr_lo;
};

// 16 byte process executable name (comm)
struct Execname {
    __u64 execname_hi;
    __u64 execname_lo;
```

```
};

// Array of execnames
BPF_ARRAY(allowlist_execname, struct Execname, TOTSIZE);
// Array of ips
BPF_ARRAY(allowlist_ip, __u32, TOTSIZE);
// Array of CIDRs
BPF_ARRAY(allowlist_cidr_for_ip, __u32, TOTSIZE);
// Array of ports
BPF_ARRAY(allowlist_port_for_ip, __u32, TOTSIZE);

// kprobe pushes to sockfile execname to this map
BPF_HASH(sockfile_pname, struct file *, struct Execname);

static bool ip_in_net(__u32 candidate_ip, __u32 net_ip, __u32 cidr4_mask) {
    return ((candidate_ip >> (32 - cidr4_mask)) == (net_ip >> (32 -
        cidr4_mask)));
}

// Socket filter BPF program
int sock_filter(struct __sk_buff *bpf_skb) {
    if (bpf_skb == NULL) return SOCK_PASS;

    struct sk_buff skb;

    // Base addresses for __sk_buff and sk_buff are the same, so reading
    // sk_buff from kernel mem
    int succ = bpf_probe_read_kernel(&skb, (__u32)sizeof(struct sk_buff),
        bpf_skb);
    if (succ != 0) return SOCK_PASS;

    // Read sock object from kernel
```

```
struct sock *sk = skb.sk;

// Read socket object
struct socket *sock;
succ = bpf_probe_read_kernel(&sock, (__u32)sizeof(struct socket*),
    &sk->sk_socket);
if (succ != 0) return SOCK_PASS;

// Read file object for socket file
struct file *file;
succ = bpf_probe_read_kernel(&file, (__u32)sizeof(struct file*),
    &sock->file);
if (succ != 0) return SOCK_PASS;

// Read fowner
struct fown_struct f_owner;
succ = bpf_probe_read_kernel(&f_owner, (__u32)sizeof(struct fown_struct),
    &file->f_owner);
if (succ != 0) return SOCK_PASS;

// sk->daddr and such are not populated when using raw sockets. Need to
    obtain
// daddr from iphdr.
struct iphdr *iph = (struct iphdr *)skb_network_header(&skb);

__be32 daddr;
succ = bpf_probe_read_kernel(&daddr, (__be32)sizeof(__be32), &iph->daddr);
if (succ != 0) return SOCK_PASS;

if (bpf_skb->sk == 0) return SOCK_PASS;

// Host and port converted to network byte order
```

```
__u32 port = htons(bpf_skb->sk->dst_port);
__u32 ip = htonl(daddr);

// Lookup for current process name and check if the process name has a
    filter
struct Execname *e = sockfile_pname.lookup(&file);
if (e == NULL) return SOCK_PASS;

// Check for match in arrays
int i = 0;
#pragma clang loop unroll(full)
while (i < TOTSIZE) {
    int idx = i;
    struct Execname *candidate_exec = (struct Execname *)
        allowlist_execname.lookup(&idx);
    if (candidate_exec == NULL) { i++; continue; }

    if (candidate_exec->execname_hi == e->execname_hi &&
        candidate_exec->execname_lo == e->execname_lo){
        // Found execname, check ip4 cidr
        __u32 *bl_ip = allowlist_ip.lookup(&idx);
        __u32 *cidr4_mask = allowlist_cidr_for_ip.lookup(&idx);
        if (bl_ip == NULL || cidr4_mask == NULL) { i++; continue; }

        if (ip_in_net(ip, *bl_ip, *cidr4_mask)) {
            // IP found, check port
            __u32 *bl_port = allowlist_port_for_ip.lookup(&idx);
            if (bl_port == NULL) { i++; continue; }

            if (*bl_port == port) {
                // Matched
                return SOCK_PASS;
```

```
            }
        }
    }
    i++;
}

bpf_trace_printk("BLOCKED for %s%s connection to %d", &e->execname_lo,
    &e->execname_hi, ip);
return SOCK_DROP;
}


// Get executable file name that calls sock_alloc_file and push to map
int kprobe_map_sockfile_pname(struct pt_regs *ctx) {
    // Get return value (file pointer) from pt regs
    struct file *file = (struct file *) PT_REGS_RC(ctx);

    // struct Execname execname;
    struct Execname execname;
    execname.execname_lo = 0;
    execname.execname_hi = 0;

    // get comm from the task struct of current program
    bpf_get_current_comm((char *)&execname, PROC_EXECNAME_MAX);
    // Push execname to map
    sockfile_pname.update(&file, &execname);
    return 0;
}
```

APPENDIX B

APPENDIX: TABLE DETAILING THE BEHAVIOUR OF EBPF NETWORK FILTER

Table TA2.1: Table detailing the behaviour of the filter for various possible events

Event	Action (What happens)	Effect (Behaviour within the container)
New container comes up	1. ifprobe pushes 'up' event to the shared event queue along with the container's veth's ifindex. 2. Userspace agent then obtains the container's name and ID to check against the policy. 3. If the container is in the policy, filter is attached to the container's cgroup	None
Container process makes request to an allow-listed service	1. Process' command name is obtained during socket file creation and pushed to a shared map. 2. filter program reads the packet's destination IP, port and process name to check against the policy. 3. Since the service is allow-listed, filter returns a zero exit code to let the packet pass.	The request is successful as expected.
Container process makes request to a non-allow-listed service	1. Process' command name is obtained during socket file creation and pushed to a shared map. 2. filter program reads the packet's destination IP, port and process name to check against the policy. 3. Since the service is disallowed, filter returns a non-zero exit code to drop the packet.	The request is unsuccessful. The packet is repeatedly retransmitted until timeout in case of TCP.
A container is stopped	1. ifprobe pushes a 'down' event to the shared event queue along with the container's veth's ifindex. 2. Userspace agent then obtains the container's ID. 3. filter, if attached, is detached from the container's cgroup.	None