Kubernetes inter-pod container isolation

Aarni Halinen

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 1.6.2023

Thesis supervisor:

Prof. Mario Di Francesco

Thesis advisors:

M.Sc. (Tech.) José Luis Martin Navarro

M.Sc. (Tech.) Jacopo Bufalino

Aalto University
School of Science

AALTO UNIVERSITY
SCHOOL OF SCIENCE

ABSTRACT OF THE
MASTER'S THESIS

Author: Aarni Halinen

Title: Kubernetes inter-pod container isolation

Date: 1.6.2023 Language: English Number of pages: 6+15

Department of Computer Science

Professorship: Computer Science

Supervisor: Prof. Mario Di Francesco

Advisors: M.Sc. (Tech.) José Luis Martin Navarro, M.Sc. (Tech.) Jacopo Bufalino

Your abstract in English. Try to keep the abstract short; approximately 100 words should be enough. The abstract explains your research topic, the methods you have used, and the results you obtained. Your abstract in English. Try to keep the abstract short; approximately 100 words should be enough. The abstract explains your research topic, the methods you have used, and the results you obtained.
Your abstract in English. Try to keep the abstract short; approximately 100 words should be enough. The abstract explains your research topic, the methods you have used, and the results you obtained. Your abstract in English. Try to keep the abstract short; approximately 100 words should be enough. The abstract explains your research topic, the methods you have used, and the results you obtained.

Keywords: Kubernetes, Container, Docker, Security

Preface

I want to thank Professor Pirjo Professori and my instructor Olli Ohjaaja for their good and poor guidance.

Otaniemi, 16.1.2015

Eddie E. A. Engineer

Contents

Symbols and abbreviations

Symbols

$\uparrow$	electron spin direction up
$\downarrow$	electron spin direction down

Operators

$\nabla \times \mathbf{A}$	curl of vectorin $\mathbf{A}$

Abbreviations

K8s	Kubernetes
STRIDE	an object-oriented analog circuit simulator and design tool

1 Introduction

1.1 Problem Statement

While the sidecar pattern makes it easier to add peripheral tasks to applications, it opens up questions about application security. In Kubernetes, there is limited amount of security features available on container-level. Most of the security related policies and capabilities are defined for the Pod, which essentially means that any capability required by the main application is inherited in the sidecar. Any privilege or network policy granted for the main application can be used by the sidecar for escalation and lateral movement.

Most often, developers rely on containers by third parties for the sidecar tasks. The source code of the sidecar containers, even if it were open, can be hard or even impossible to verify for known vulnerabilities. This, combined with the limited security features for sidecars, makes any exploitable security issue in the sidecar an optimal launchpad for attack against the whole cluster. Furthermore, malicious actors can use supply chain attacks and typosquatting to trick victims into installing malicious sidecars to their clusters.

This thesis proposes a solution for limiting capabilities of sidecar without limiting those of the main container, thus extending the principle of least privilege to within the pod.

1.2 Thesis outline

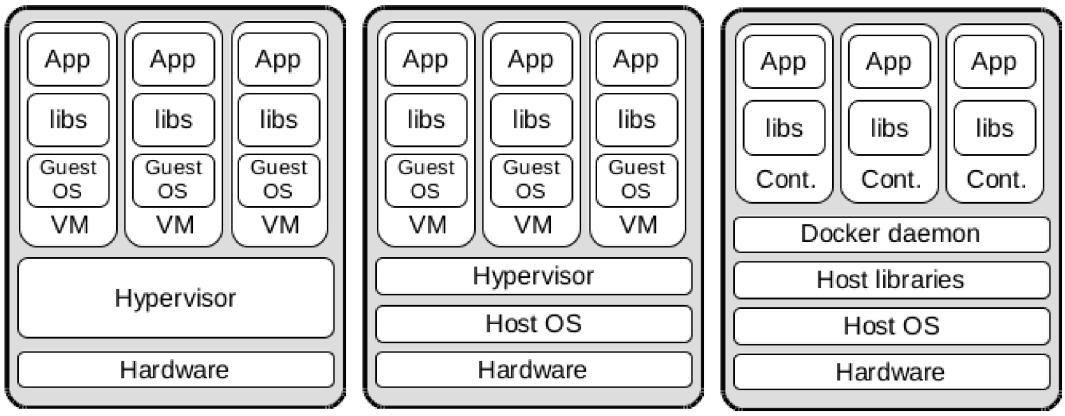
The following chapter 2 gives background about containers, Kubernetes and explains their common attack vectors. It also discusses Kubernetes networking and container network interface plugins. Chapter 3 proposes ideas for isolating sidecars from main application container. The chapter discusses both container and networking security in the context of Kubernetes Pod. Chapter 4 introduces an implementation based on the findings of the previous chapter. The pros and cons of the solution are discussed in Chapter 5. Finally, Chapter 6 discusses future research and concludes the thesis.

2 Background

2.1 Principle of least priviledge, Zero trust...

2.2 Microservices architecture

2.3 Containerization and Docker



(a) Type1 hypervisor (b) Type2 hypervisor (c) Container

Figure 1: Virtualization models [6]

Figure 1 illustrates common virtualization models. Whereas traditional virtualization techniques virtualize workloads on top of a hypervisor which shares hardware resources between the virtual machines, containerization is a technique where virtualization happens on a operating system level [22]. Processes executing in containers run on the host machine kernel. However, each container is isolated to its own network, process namespace and so on; two containers on the same host OS do not know that they share resources. Furthermore, containers are similarly isolated from accessing host OS resources.

BSD jails and *chroot* can be considered early forms of containerization technology, so the idea of containers is not new [6]. Recent Linux container solutions rely on two main implementations: Linux Containers (LXC) -based solution that relies on kernel features such as control groups (cgroups) and namespaces, and a custom kernel and Linux distribution called Open Virtuozzo (OpenVZ). Docker [14] is a hugely popular container runtime that is based on LXC and provides an easy-to-use API and tooling for creating and managing containers. Docker also provides containerization for other OSes as well. However, in this thesis we focus only on the Linux implementation.

2.3.1 Linux containers

The Linux containers technology implements container isolation and containement using Linux kernel feature called namespaces [20]. Namespaces [11] are a construct that wraps a global system resource in an abstraction which makes it appear to the processes in the namespace that they have their own, isolated, instance of the global resource. There are total of eight namespaces: i) Cgroup which is used for resource management, ii) Inter-process communication (IPC) which isolates POSIX message queues etc., iii) Network which isolates network devices, stack ports etc., iv) Mount for file system isolation, v) Process ID (PID), vi) Time, vii) User for isolating user and group identifiers and viii) UTS which isolates hostnames and NIS domain names. For example, network namespace provides each container their own loopback device and even `iptables` rules. In another example, mount namespace makes sure that container has no visibility nor access to the host's or other container's file system. Compared to other namespaces that concern isolation of kernel data, cgroups focuses on limiting available system resources per namespace [20]. Each namespace can be setup with their own limits on CPU and memory usage and available devices. Using Docker as an example, setting *–cpu*, *–memory* and *–devices* options will limit available resources for the container.

Since all containers and the host machine run on same kernel, any container that manages to breakout from isolation may compromise other containers, the host and the whole kernel. To combat this container breakout, several Linux kernel security mechanisms are adopted to constrain the capabilities of containers [20]. The mechanisms include Discretionary Access Control (DAC) mechanisms like Capability [10] and Secure computing mode (Seccomp) [12], and Mandatory Access Control (MAC) mechanisms like Security-Enhanced Linux (SELinux) and AppArmor [1]. With Capability, the superuser (i.e. the root user) privilege is divided into distinct units, each of which represent a permission to process some specific kernel resources. The feature turns the binary "root/non-root" security mechanism into fine-grained access control system, which makes it easier to follow the principle of least privilege. For example, processes like web servers that just need to bind on a Internet domain privileged port (numbers less than 1024) do not need to run as root; they can just be granted with `CAP_NET_BIND_SERVICE` capability instead [15]. The Seccomp mechanism constrains which system calls a process can invoke. The available system calls are defined for a container through Seccomp profile which is defined as a JSON file. The Docker default Seccomp profile [13] includes over 300 system calls. SELinux is integrated to CentOS/RHEL/Fedora distributions and utilizes a label-based enforcement model, while AppArmor is available in Debian and Ubuntu and adopts a path-based enforcement model [20].

2.3.2 Container breakout scenarios

[4]

1. Priviledged container
2. CAP_SYS_ADMIN, mounting /proc and chroot

3. CAP_SYS_PTRACE, shellcode injection to running program, nc 172.17.0.1 on port running shell

4. Mounted docker socket, creating priviledged containers

2.4 Kubernetes system components, control plane

2.4.1 apiserver

2.4.2 etcd

2.4.3 scheduler

2.4.4 controller-manager

2.5 Kubernetes resources

2.5.1 Namespaces

2.5.2 Pods

2.5.3 Services

2.5.4 Admission control

2.6 Sidecar pattern

2.7 Kubernetes network model

Integral part of Kubernetes cluster is how nodes and resources are networked together. Specifically, the networking model needs to address four different type of networking problems: i) intra-Pod (ie. container-to-container within same Pod) communication, ii) inter-Pod communication between Pods, iii) Service-to-Pod communication and iv) communication from external sources to Services [2]. The model also requires that each Pod is IP addressable and can communicate with other Pods without network address translation (NAT), even when Pods are scheduled on different hosts [24]. All agents on a host should also be able to communicate with Pods on the same host. The implementation of this model is not part of Kubernetes, but is handed to special plugins that implement Container Network Interface (CNI) specification.

2.7.1 Container Network Interface

The Container Network Interface (CNI) [7] is a networking specification, which has become de facto industry standard for container networking. It is backed by Cloud Nativce Computing Foundation (CNCF) [24]. CNI was first developed for the container runtime `rkt`, but it is supported by all container runtimes and there is a large number of implementations to choose from [18]. Most of the container orchestrators have adopted the specification as their networking solution. The biggest outlier is Docker Swarm, which instead implements `libnetwork` [16].

The specification has five distinct definitions: i) a format for network configuration, ii) a execution protocol between the container runtimes and the plugin binary, iii) a procedure for the runtime to interpret the configuration and execute the plugins, iv) a procedure for deletegating functionality between the plugins and v) data types for plugins to return their results to the runtime [7]. The configuration is defined as a JSON file and it includes a list of plugins and their configuration. The file is consumed at plugin execution time by the runtime, and passed to the plugins. The execution protocol defines a set of operations (ADD, DEL, CHECK) for adding and removing containers from the network, while also defining a set of OS environment variables that are used as parameters by the plugins. When the runtime mutates a container network, it results in a series of ADD, DELETE or CHECK executions. These are then executed in same order as defined in the `plugins` list, or reversed order for DELETE executions. Each plugin then returns either `Success` or `Error` JSON object. The execution of a series of operations ends when it encounters the first Error response, or when all the operations have been performed.

2.7.2 Network policies

Since Kubernetes does not provide networking between the Pods, it has no capabilities to enforce network isolation between workloads. Thus, another key feature for CNI plugins is enforcing network traffic rules. Kubernetes provides a common resource called `NetworkPolicy` for CNI plugins to consume. The NetworkPolicy specification consists of a `podSelector` that specifies pods that are subject to the policy and `policyTypes` to specify Ingress and Egress rules for the traffic [3] to the target Pod. Each rule includes `to` or `from` field for selecting Pod, Namespace or IP address block in CIDR notation on the other side of the connection, and `ports` field for explicitly specifying which ports and protocols are part of the rule. The policies are additive; when multiple rules are defined for a Pod, the traffic is restricted to what is allowed by the union of the policies. Many CNI plugins also introduce Custom Resource Definitions for their own, more granular, network policy rules.

2.7.3 Cilium and other CNI plugins

While all CNI plugins meet the requirements listed above, they may differ in architecture significantly. The plugins can be classified based on which OSI model network layer they operate on, which Linux kernel features they use for packet filtering and which encapsulation and routing model they support for inter-host and intra-host communication between Pods.

In this thesis, we focus on three different CNI plugins: Cilium, Calico and Multus.

Cilium [8] is one of the most advanced and powerful CNI plugins for Kubernetes. It works by creating virtual ethernet device for each Pod and sets one side of the link into Pod's network namespace [17]. Cilium then attaches extended Berkeley Packet Filter (eBPF) programs to ingress traffic control (`tc`) hooks of these virutal ethernet devices for intercepting all incoming packets from the Pod. The packets are intercepted and processed before the network stack and thus `iptables`, reducing latency 20%-30% and even doubling the throughput of packets in some scenarios [3].

Cilium provides Custom Resource Definition `CiliumNetworkPolicy` that supports policies in layers 3-7 instead of standard L3/L4. With `CiliumClusterwideNetworkPolicy`, network rules can applied to every namespace in the cluster, or even to nodes when using `nodeSelector`.

2.7.4 Extended Berkeley Packet Filter

Berkeley Packet Filter (BPF, or nowadays often cBPF) was originally developed in early 1990s as a high-performance tool for user-space packet captures [21]. BPF works by deploying the filtering part of the application, `packet filter`, in the kernel-space as an agent. The `packet filter` is provided with a program (often denoted as BPF program) consisting of BPF instructions, which works as a set of rules for selecting which packets are of interest in the user-space application and should be copied from kernel-space to user-space. The instuctions are executed in a register-based pseudo machine. Since network monitors are often interested only in subset of network traffic, this limits the number of expensive copy operations across the kernel/user-space protection boundary only to packets that are of interest in the user-space application. A notable usecase for BPF is *libpcap* library, which is used by network monitoring tool called `tcpdump`.

Later in the 2010s the Linux community realized that BPF and it's ability to instrument the kernel could benefit other areas than packet filtering as well [25]. This reworked version of BPF was first merged in to Linux kernel in 2014 and is publicly called extended Berkeley Packet Filter (eBPF) to distinguish it from the original cBPF. The kernel development community continues to call the newer version BPF, but instead of the original acronym consider it a name of a technology. Similarly to the kernel community, the term BPF always refers to the eBPF in this thesis.

The eBPF programs are compiled to bytecode and loaded to kernel with `bpf()` system call [23]. Most often programs are written in restricted C and compiled with LLVM Clang compiler to bytecode. It also possible to use eBPF assembly instructions and `bpf_asm` utility for converting instructions to bytecode. eBPF programs follow a event-driven architecture: a loaded eBPF program is hooked to a particular type of event and each occurence of the event triggers the program execution.

For networking purposes, there are two eBPF hooks available for intercepting and mangling, forwarding or dropping network packets: eXpress Data Path (XDP) and Traffic Control (TC) [23]. In Cloudflares DDoS testing benchmark [5], XDP program was capable to drop 10 million and TC program 2 million packets per second, while common `iptables` INPUT rule was able to drop less than one million packets per second.

XDP programs are attached to a network interface controller (NIC) and can handle only incoming packets [19]. The programs are called directly by the NIC driver if it has XDP support, thus executing before packets enters the network stack. This skips expensive packet parsing and memory allocation operations, and allow XDP programs to run at very high throughput. Thus, even the main networking buffer *skbuff* is not populated. Some SmartNICs even support offloading the program to the NIC's own processor from host CPU, improving host machine performance

even further [9]. If the driver does not support XDP, generic XDP is used and the programs run after the packet has been parsed by the network stack.

XDP programs can read and modify contents of the packets [25]. Since the packets are not parsed the network stack, the programs have to work with raw packets and implement own parsing functionality. The program's return value determines how the packet should be processed further. With `XDP_DROP` and `XDP_PASS` return values, the packet can be dropped or passed further to the networking stack respectively. The packet can also be bounced back to the same NIC it arrived on with `XDP_TX`, usually after modifying the packet contents. `XDP_REDIRECT` is used for redirecting the packet to a different NIC, CPU or even to another socket.

TC programs are executed when both incoming and outgoing packets reach kernel traffic control function within the Linux network stack [25]. The ingress hook executes after the packet is parsed to *skbuff* but before most of the network stack. On egress the stack is traversed in reverse, thus the hook executes after most of the network stack. TC programs can read and write directly to packet in memory. Similarly to XDP programs, the return value of the program determines further processing of the packet. The packet can be passed furhter in the stack with `TC_ACT_OK`, dropped with `TC_ACT_SHOT`, or the modified packet can be redirected back to the start of the classification with `TC_ACT_RECLASSIFY`, among others.

3 Research material and methods

3.1 Prevent container breakout with Pod Security Admission

3.2 IPTables

- If executed from containers in Pod (init, lifecycle, wrapper container), it breaks Security admission rules (root user and NET_ADMIN)
- Can be executed from Node itself, using DaemonSet (sort of a CNI plugin), but a bit hacky.
- Use owner module for catching egress packets (userId, groupId, processId)

3.3 eBPF program firewall

- XDP works only for ingress
- TC needs some way to catch egress from sidecar

3.4 Own pod for sidecar

- Guaranteed to work, since own network namespaces. What type of issues arise from this?
- Need to force pods to same node, for common volumes (if using host as storage)
- Implementation by hand, or Admission controller that catches sidecars?
- Loopback is not the same anymore. DNAT that changes localhost to external? => impossible with IPTables!

3.5 Multus

- Requires breaking sidecar pattern with multiple Pods
- Allows use of custom IP addresses
- Hard to implement between nodes, affinity rules
- Loopback not easy to hijack for forwarding to new IPs

4 Evaluation

5 Discussion

6 Conclusion

References

[1] AppArmor. *AppArmor*. 2022. URL: https://apparmor.net/ (visited on 03/31/2023).

[2] The Kubernetes Authors. *Cluster Networking*. 2022. URL: https://kubernetes.io/docs/concepts/cluster-administration/networking/ (visited on 03/09/2023).

[3] Gerald Budigiri et al. "Network policies in kubernetes: Performance evaluation and security analysis". In: *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. IEEE. 2021, pp. 407–412.

[4] Thanh Bui. "Analysis of docker security". In: *arXiv preprint arXiv:1501.02967* (2015).

[5] Cloudflare. *How to drop 10 million packets per second*. 2018. URL: https://blog.cloudflare.com/how-to-drop-10-million-packets/ (visited on 03/15/2023).

[6] Theo Combe, Antony Martin, and Roberto Di Pietro. "To docker or not to docker: A security perspective". In: *IEEE Cloud Computing* 3.5 (2016), pp. 54–62.

[7] CoreOS. *CNI—The Container Network Interface*. 2023. URL: https://github.com/containernetworking/cni/blob/v1.1.2/SPEC.md (visited on 03/10/2023).

[8] Cilium Developers. *Cilium*. 2023. URL: https://cilium.io/ (visited on 03/14/2023).

[9] Cilium developers. *Program types*. 2018. URL: https://docs.cilium.io/en/latest/bpf/progtypes/ (visited on 03/16/2023).

[10] Linux Developers. *capabilities(7) — Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man7/capabilities.7.html (visited on 03/31/2023).

[11] Linux Developers. *namespaces(7) — Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 03/31/2023).

[12] Linux Developers. *seccomp(2) — Linux manual page*. 2023. URL: https://man7.org/linux/man-pages/man2/seccomp.2.html (visited on 03/31/2023).

[13] Docker. *Docker default Seccomp profile*. 2022. URL: https://github.com/moby/moby/blob/23.0/profiles/seccomp/default.json (visited on 03/31/2023).

[14] Docker. *Docker overview*. 2018. URL: https://docs.docker.com/get-started/overview/.

[15] Docker. *Docker security*. 2023. URL: https://docs.docker.com/engine/security/ (visited on 03/31/2023).

[16] Docker. *libnetwork*. 2023. URL: https://github.com/moby/moby/tree/master/libnetwork (visited on 03/10/2023).

[17] The Kubernetes Networking Guide. *Cilium*. 2023. URL: https://www.tkng.io/cni/cilium/ (visited on 03/14/2023).

[18] Michael Hausenblas. *Container Networking*. O'Reilly Media, Incorporated, 2018.

[19] Toke Høiland-Jørgensen et al. "The express data path: Fast programmable packet processing in the operating system kernel". In: *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 2018, pp. 54–66.

[20] Xin Lin et al. "A measurement study on linux container security: Attacks and countermeasures". In: *Proceedings of the 34th Annual Computer Security Applications Conference*. 2018, pp. 418–429.

[21] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture." In: *USENIX winter*. Vol. 46. 1993.

[22] Dirk Merkel et al. "Docker: lightweight linux containers for consistent development and deployment". In: *Linux j* 239.2 (2014), p. 2.

[23] Sebastiano Miano et al. "A framework for eBPF-based network functions in an era of microservices". In: *IEEE Transactions on Network and Service Management* 18.1 (2021), pp. 133–151.

[24] Shixiong Qi, Sameer G Kulkarni, and KK Ramakrishnan. "Assessing container network interface plugins: Functionality, performance, and scalability". In: *IEEE Transactions on Network and Service Management* 18.1 (2020), pp. 656–671.

[25] Marcos AM Vieira et al. "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications". In: *ACM Computing Surveys (CSUR)* 53.1 (2020), pp. 1–36.

A Esimerkki liitteestä

Liitteet eivät ole opinnäytteen kannalta välttämättömiä ja opinnäytteen tekijän on kirjoittamaan ryhtyessään hyvä ajatella pärjäävänsä ilman liitteitä. Kokemattomat kirjoittajat, jotka ovat huolissaan tekstiosan pituudesta, paisuttavat turhan helposti liitteitä pitääkseen tekstiosan pituuden annetuissa rajoissa. Tällä tavalla ei synny hyvää opinnäytettä.

Tämän tekstin lähteenä oleva tiedosto on opinnäytteen pohja, jota voi käyttää kandidaatintyössä, diplomityössä ja lisensiaatintyössä. Tekstin lähteenä oleva tiedosto on kirjoitettu LaTeX-tiedoston rakenteen opiskelemista ajatellen. Tiedoston kommentit sisältävät tietoa, joka on hyödyllistä opinnäytettä kirjoitettaessa.

Johdanto selvittää samat asiat kuin tiivistelmä, mutta laveammin. Johdannossa kerrotaan yleensä seuraavat asiat

- Tutkimuksen taustaa ja tutkimusaiheen yleisluonteinen esittely
- Tutkimuksen tavoitteet
- Pääkysymys ja osaongelmat
- Tutkimuksen rajaus ja keskeiset käsitteet.

Lyhyiden opinnäytteiden johdannot ovat yleensä liian pitkiä, joten johdannon paisuttamista on vältettävä. Diplomityöhön sopii johdanto, joka on 2–4 sivua. Kandidaatintyön johdannon on oltava diplomityön johdantoa lyhyempi. Sopivasti tiivistetty johdanto ei kaipaa alaotsikoita.

Tässä osassa kuvataan käytetty tutkimusaineisto ja tutkimuksen metodologiset valinnat, sekä kerrotaan tutkimuksen toteutustapa ja käytetyt menetelmät.

Tässä osassa esitetään tulokset ja vastataan tutkielman alussa esitettyihin tutkimuskysymyksiin. Tieteellisen kirjoitelman arvo mitataan tässä osassa esitettyjen tulosten perusteella.

Tutkimustuloksien merkitystä on aina syytä arvioida ja tarkastella kriittisesti. Joskus tarkastelu voi olla tässä osassa, mutta se voidaan myös jättää viimeiseen osaan, jolloin viimeisen osan nimeksi tulee »Tarkastelu». Tutkimustulosten merkitystä voi arvioida myös »Johtopäätökset»-otsikon alla viimeisessä osassa.

Tässä osassa on syytä myös arvioida tutkimustulosten luotettavuutta. Jos tutkimustulosten merkitystä arvioidaan »Tarkastelu»-osassa, voi luotettavuuden arviointi olla myös siellä.

Opinnäytteen tekijä vastaa siitä, että opinnäyte on tässä dokumentissa ja opinnäytteen tekemistä käsittelevillä luennoilla sekä harjoituksissa annettujen ohjeiden mukainen muotoseikoiltaan, rakenteeltaan ja ulkoasultaan.