Cilium API Filtering and Security Analysis

Albert Bautista

ABSTRACT

Traditional network filtering systems like iptables rely on more static classifiers like IP addresses and ports which falls short for the needs of modern implementations that run applications in containers. A container is an isolated runtime environment that contains every dependency needed to run its application. Images of these containers allow for the application to be replicated into multiple containers to improve reliability. But because each container is mapped to a new IP addresses, iptables requires updates to its rules to maintain the intended connections which comes at the cost of performance. Additionally, modern API interfaces that access different resources communicate over common ports like HTTP, SSH, etc. The usage of common ports can lead to possible security vulnerabilities as iptables fails to discern the underlying access to different resources that goes through the same port. For these problems, Cilium provides a high-performing and flexible solution at the network filtering level by introducing an API-aware filtering system that can map containers uniquely even across replications.

In this paper, I first present the differences in architecture and performance with the traditional system with iptables as opposed to Cilium. Then I demonstrate how Cilium allows for API-level security in a multi-container application and performance overhead running Cilium in a basic container application.

1. INTRODUCTION

Containers provide isolated, lightweight and scalable instances, each encapsulating everything an application needs to run. These properties make containers very useful for purpose of running applications as microservices or a collection of smaller services that together work as an application. With the performance benefits and reliability from scaling, the containerized, microservice-oriented architecture for running applications has been growing in emergence across the modern technology landscape.

Network filtering is the process of controlling the flow between one network to another. Traditional network filtering is usually handled through iptable rules. Iptables is a Linux kernel program that utilizes rules to define allowed connections, dropped or ignored connections, and forwarding connections based on the IP address and the port number of the incoming packet. The issue with utilizing iptables to filter networking traffic for these containerized systems is twofold. First is that IP addresses for containers vary due to the dynamic nature of containers. Containerized instances with a specific IP address can be swapped out for a new instance with a new IP address at a moment's notice. Second, most of these packets target a narrow subset of these ports whereas more specific identifiers like the API calls are not detected in the iptables system which presents a security vulnerability. Though there are workarounds to these issues like addressing IP ranges and handling API filtering at the application developer level, these solutions have overhead costs that contribute to problems with scalability and usability for modern systems. And as more and more technologies adopt a microservice-oriented architecture, there is a growing need to eliminate this possible bottleneck.

Cilium solves this bottleneck by filtering packets based on containers labels, in addition Cilium also extends the filtering to evaluate API calls. To achieve this functionality, Cilium uses the extended Berkeley Packet Filter (eBPF) which is a kernel based virtual machine that provides an instruction set to inject BPF programs into the Linux kernel. As a result, Cilium has extended network filtering to look at multiple points or Layers on the network stack. One is Layer 3 (L3) which includes IP addresses and container labels. Another is Layer 4 (L4) which involves ports. The last is Layer 7 (L7) which is the Application Layer that includes HTTP calls. With network filtering being unbound from just IP addresses and ports, Cilium works well with containers. With network filtering extending to the Application Layer, Cilium provides a better functionality for developers to make their applications secure. These benefits also come with the performance gains of running the filters at the kernel level.

This paper analyzes Cilium from a usability standpoint and a performance standpoint through two experiments /newline through two experiments. The first experiment focuses on running a multi-container application and demonstrating how Cilium can provide filtering security for the application. This experiment shows how usable Cilium is for the purpose of API security through the multi-container system. The second experiment acts as a micro-benchmark focused on testing throughput and latency of a simple server application. This experiment looks at how Cilium policies impact

the performance of the system.

In Section 2, I evaluate a related work that is most relevant to this project concerning the usability of Cilium compared to iptables as motivation for this project along with other motivations. In Section 3, I detail the experiment setup. Section 4 is an evaluation of the results and how the results supports the usability of Cilium for the purposes of security across Layers 3,4, and 7 of the networking stack. Section 5 looks at other related works that look at other implementations of eBPF. Lastly, in Section 6, I discuss open questions and suggest future work.

2. MOTIVATION

In terms of the works evaluating Cilium, to the best of my knowledge, I only found one work that focused on Cilium. The work by Brujin covers the same space that this paper does, focusing on the performance comparisons between Cilium and iptables as well as the impact of Cilium policy complexity [1]. To differentiate from Brujin, this paper in the first experiment evaluates how usable Cilium is and demonstrates the rules to enforce security throughout the system. The second experiment though goes over the same space as Brujin by the evaluating the performance between Cilium and iptables but utilizes a different performance tool. This experiment acts to evaluate the results of the Brujin though with different configurations in tooling and container architecture.

In choosing this project over the others, my motivation was to explore a new technology that relates to containerized systems while being able to build on prior knowledge on concepts in class like iptables and network filtering. Also, I felt like the deliverables were simple and feasible given my relatively low knowledge of networking, but still interesting and substantive enough to write a detailed report on.

3. EXPERIMENT SETUP

The experiments were ran on a VirtualBox 6.0.14 virtual machine running Ubuntu 19.04 as the guest OS with 4 GB of memory allocated to it. The experiment utilizes Docker containers orchestrated by Kubernetes utilizing the microk8s software package. In microk8s, Cilium is an add-on that I installed through microk8s.

This work consists of two experiments.

For the first experiment, I modified a Cilium demo based on Star Wars to create a demonstration where I went over the vulnerabilities of the system and what Cilium rules can be applied to block exploits for these vulnerabilities.

The Cilium demo consists of a centralized Go server (or as it's labeled the Deathstar) and two containers representing spaceships, labeled tiefighter and xwing respectively. The Deathstar server holds the API according to the OpenAPI or Swagger specification with three accessible APIs. These APIs are: GET /v1 which exposes health characteristics of the Deathstar as well as its APIs, POST /v1/request-landing which allows a ship to 'land', and finally PUT /v1/exhaust-port which causes the server to hang and crash. The demo involves the alliance controlling tiefighter pods to destroy the Deathstar service by attempting to call PUT /v1/exhaust-port to the Deathstar server. The spaceship pods can use curl to create HTTP calls to the Deathstar server with varying effectiveness depending on the Cilium policy set.

To extend the demo, I modified the base images of the Server and the tiefighter pod and added two more containers. The new containers act as remote storage for the Deathstar server. One is an FTP server that is secure from exploits in this demonstration scenario. The other container is an FTP server that is vulnerable and accessible to anyone. For the Deathstar server, I added API logging to simulate the need for sending outgoing traffic to remote store the logs and an FTP client to allow the server to send data to the FTP servers. I also removed the reference to the PUT /v1/exhaust-port API from the GET /v1 call to obfuscate the vulnerability. For the tiefighter pod, I modified the Dockerfile to add an FTP client. Through this setup, I can discuss the vulnerabilities of the system as well as the ways Cilium can block these vulnerabilities. The container setup for the experiment cluster is shown below in Figure 1.

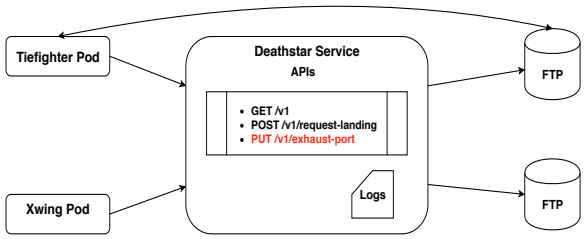

Figure 1: Deathstar Service acts as a server for the ship pods to call with APIs logs to send to remote storage

For the second experiment, it consists of a container running a simple Swagger Go-based server with a single GET API that returns "hello world" and a pod running wrk2 that load tests the API endpoint. The Cilium policy is set to always allow for the GET API to go through to ensure that the benchmark works to tests the Cilium policy configurations alone. To ensure that the server and pod are running correctly I swap the deployment and pods for new instances.

For the wrk2, I ran the tool with five fixed arguments. The first was duration time which I set to 300 seconds or five minutes to ensure that I captured as many data points for latency metrics as possible. This is to get a good view of the distribution of latency for each Cilium configuration. The second argument was the threads to run the requests which I set to the cores of my system which was 4. The third argument was rate at 10000 which sets the expected throughput or requests per second (rps) for the performance test. The fourth argument was connections which I set to 2000. Finally, the optional argument called latency which provides advanced statistics for the results including a percentile spectrum for the latency.

I ran the wrk2 tests on five configurations of Cilium: one with Cilium deactivated, one with a simple Cilium policy with one L3 rule, one with a Cilium policy with one L4 rule, one with a Cilium policy with one L7 rule and the last configuration is a complex Cilium policy with forty L7 rules. The deactivated Cilium utilizes iptables by default which works as a baseline. The three single rule policies increase in complexity as L4 has an inherent L3 rule attached likewise with L7 holding both L4 and L3 rules inherently. This is to test whether rules aimed at different layers has varying levels of performance overhead. Finally, the complex Cilium

policy with forty L7 rules aims to test whether scaling the rules influences on the system performance.

4. EXPERIMENT RESULTS

For the first experiment, I go over how the vulnerabilities of a multi-container demo application resulted in the destruction of the Empire's Deathstar server. The cause of the Deathstar's destruction was that the API call PUT /v1/exhaust-port is requested. Seeing this critical oversight, the Empire had hidden the destructive API call from the Alliance which is the adversarial group attempting to take down the Deathstar. But the Empire neglected to implement Cilium policies to enforce incoming and outgoing traffic with the Deathstar.

I'll go over the steps in which the Alliance managed to take down the Deathstar and the Cilium policies that the Empire should have enabled to avoid this breach. As a contrivance for this demonstration, the Deathstar cannot explicitly block the exhaust-port API call with Cilium due to ventilation reasons but can block any other traffic to and from the Deathstar server using Cilium policies.

To infiltrate the Deathstar server with their xwing pods, the Alliance made use of the simple API call used by the tiefighter pods POST /v1/request-landing. This could have been prevented by setting a filter for incoming traffic. Utilizing labels, Cilium can filter at the L3 layer to parse for the matching label that is tied to all traffic from the spaceship container. The Cilium policy that enforces such filter is shown in the code snippet below:

```
...
"ingress": [{
    "fromEndpoints": [{
        "matchLabels": {
            "org": "empire"
        }
    }]
...
```

After infiltrating the server, the alliance rebels found the vulnerability, but had to relay this knowledge to somewhere accessible to the alliance ships as they could not get out of the Deathstar. Fortunately, the Empire has been sending logging data to two storage containers; one of which was an insecure FTP server scheduled for maintenance later that week. This could have been avoided by setting a filter outgoing traffic to avoid the server to await maintenance. In addition to filtering out incoming traffic, Cilium can also monitor and stop outgoing or egress traffic with the container labels as well. The Cilium policy that enforces that traffic is shown in the code snippet below:

```
...
"egress": [{
    "toEndpoints": [{
        "matchLabels": {
            "status": "updated"
        }
    }]
...
```

Through these vulnerabilities, the alliance was able to get the API and send the information for the Alliance xwings to destroy the Deathstar server. Removing the earlier limitation of not blocking the exhaust-port API, the Empire should have simply applied a Cilium policy to specify what HTTP calls can be received by the Deathstar server. This would not be possible with the iptables due to the lack of support for L7 filtering. The simple Cilium policy that enforces the restriction of PUT /v1/exhaust-port while allowing the other calls through is shown below:

```
...
"rules": {
    "http": [
        {
            "path": "/v1/request-landing"
            "method": "POST"
        },
        {
            "path": "/v1"
            "method": "GET"
        }
    ]
...
```

For the second experiment, I tested for the latency and max throughput of a simple swagger Go server in five different configurations. The configurations varying in complexity in the targeted networking layer and rule number show how different Cilium policies can affect the latency and throughput of the system. One configuration involves Cilium disabled to run iptables. This configuration provides a baseline comparison to see how Cilium overall affects performance.

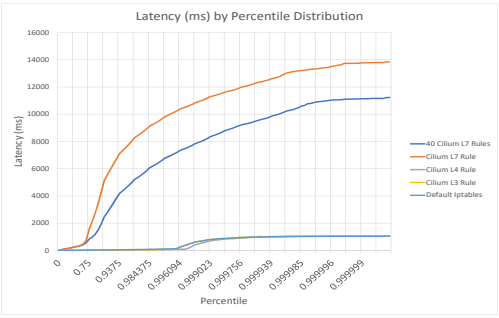

Figure 2: Latency(ms) of the connection as a function of the percentile distribution of all the requests.

Figure 2 shows the percentile distribution of the latency. This distribution gives a better picture of the tail latency and the overall trend for each of the configuration at its extremes. The results clearly show the Cilium policies with L7 complexity have more latency compared to the three less complex policies. With the single L7 having a longer tail latency at close to 14 seconds compared to the 40 rule L7 Cilium policy coming at around 11 seconds, the three less complex policies are drastically lower around 1 second at the tail latency.

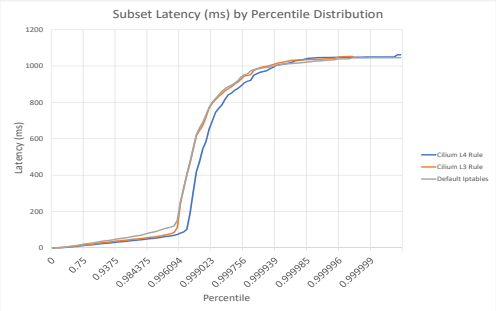

Figure 3: Latency(ms) of the connection as a function of the percentile distribution of all the requests. A subset of the three lower valued latencies.

For clarity, the lower three are separated into a smaller scaled graph in Figure 3. In comparison, the percentile distribution is close for all three with the more complex Layer 4 separating a little at the bottom. Also, the default iptables has higher latency up to the 90th percentile.

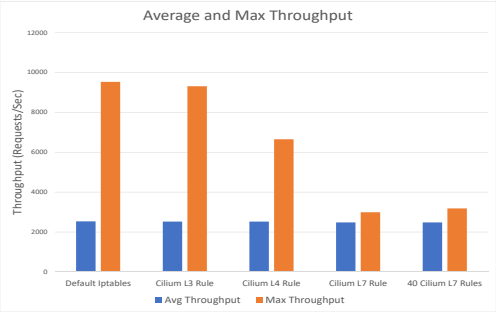

Figure 4: Max Throughput and Average Throughput of the system with all five configurations.

From Figure 4, the throughput averages are relatively stable between the five configurations shown in the blue bars at around 2500 requests per second (rps). From the max throughput of the baseline iptables configuration of 9500 rps down to around 3000 rps, there is a clear decreasing trend for the max throughput as the Cilium policy becomes more complex.

5. DISCUSSION

For this project, I investigated the usability of Cilium for multi-container security and tested for latency and throughput with different Cilium configurations.

The first experiment demonstrated the flexibility of Cilium policies for container security to allow for fine-grained filtering of specific parameters. Cilium has extended the L3 and L4 to capture labels for containers which is well-suited for the dynamic deployment environment that most containers are currently being utilized. In addition, Cilium provides L7 security for API endpoints. Still Cilium has gaps in flexibility. For example, L7 Cilium policies are still bound to HTTP calls with some other L7 protocols in a beta. In addition, labels and IP addresses do not work seamlessly in Cilium as adding IP matching requires containers to be mapped as to the matching IP despite the correct labels.

Looking at the requirements for this project, I needed to demonstrate how Cilium works for API filtering as well as security. With a demonstration of a security situation and analysis, I was able to build an application that geared more towards showing step-by-step where Cilium policies would be appropriate. Therefore, I used the Cilium demo was a good base because it had an understandable framework of the server and attack vectors that was directly mitigated through Cilium.

The goals of the first experiment were to make the policies understandable and usable. For me, I thought to make follow the naming conventions of the Star Wars demo while extending parts to demonstrate the capabilities of the Cilium policy. To extend the demo further, I looked at emulating the attack vector but with the server sending out traffic and the end result. This allowed me to also demonstrate egress policies that have the same capabilities as ingress policies and keep the policies tied to just the Deathstar container.

To extend the existing Kubernetes deployment of the demo, I had to make modifications to the Docker images used in the demo to make the functionality of sending out traffic be more sensible for the demo and build it locally. To ensure the demonstration was replicable, I followed the steps of exploiting the demo vulnerabilities with the Cilium policies off and tested being blocked at the points where Cilium policies filtered out malicious traffic. The experiment could have involved building a multi-container application, but I had a hard time thinking of balancing explaining the application's features while demonstrating how Cilium fits into the picture. An application utilizing Cilium would be a possibility for further exploration and research.

The second experiment showed performance benchmarks of throughput and latency between iptables and varying levels of complexity of Cilium policies, with L7 level Cilium policies performing considerably slower and with less throughput. From the results, there is two levels of involving latency and max throughput. On one side is L7 Cilium policies and the other side is L3, L4 Cilium policies as well as the baseline iptables. This divide shows considerable overhead involving L7 policies. The results also show the performance benefits of eBPF for the L3 and L4 Cilium policies as the latency distribution is below iptables for below 90 percent that supports the claims of Brujin et al [1]. On the other hand, the throughput averages were flat across the five configurations which partly disproves Brujin et al's claims of throughput loss for Cilium. I do not strongly refute Brujin's results as my results for average throughput might need a deeper analysis before deriving conclusions for throughput, possibly into different performance tools and server setup.

In terms the setup for second experiment, there were decisions that I made for the configuration that could influence the results. To start with, wrk2's system of setting a constant expected throughput rate for the purpose of benchmarking latency might skew the average throughput results. Still I found the tool useful for the percentile distribution of the latency gives a substantial view of the latency from the outliers to the targets. Still the latency distribution graphs

are skewed to the outlier percentiles above 90 percent making it difficult to view the averages completely. I can assure that latency averages in terms of comparison between the subsystems are in line with the tail latencies shown in the graph, so I thought to just show the graphs to avoid redundancy.

I admit that there were some difficulties in getting together the experiment correctly. To start with, wrk2 for my experience is not as precise as I needed it to be. That can be seen especially comparing the L7 policies with the larger sized 40 rule L7 Cilium policy beating out the single rule L7 Cilium policy. I resorted to making multiple runs and making best estimations on which results were reasonable. To offset my choosing, I made the runtime long and restarted the test server and wrk2 pod to ensure that the results are either stable or easily discernible. I also noticed that Cilium L7 policies for wrk2 returned with some error response whether that comes from the high load or spurious denial errors is another topic for further research.

Putting into context the fine-grained security benefits of Cilium compared to iptables as well as the usability of Cilium shown in the first experiment, the throughput and latency overhead results from the second experiment is within reasonable bounds that I still support using Cilium for container packet filtering as a part of containerized security.

6. RELATED WORK

For related works, I looked into extended Berkeley Packet Filters (eBPF), which Cilium also utilizes. eBPF is a virtual machine-like program with a simplified instruction set that allows for Just-In-Time (JIT) compilation of programs to run at the kernel level.

In [2], Scholz et al. study the performance of the eBPF programs at different levels of the network stack for the purposes of security. The paper detailed two different packet filtering applications, Express Data Path (XDP) and their own application-specific packet filtering configuration using eBPF at the socket level. In their research, XDP's performance compared to iptables showed a significant performance increase due to the usage of eBPF particularly the enabled JIT compiler. There is a tradeoff as XDP performance not as viable as bypassing the kernel. Still XDP provides flexibility for kernel integration which the paper finds suitable for Denial of Service protection. The system with XDP is fast enough for DoS protection while also allowing for packets to efficiently pass through the network stack. The application-specific filtering configuration was created for the purposes of research into the filtering viability and performance. The paper also discusses the previous approach of Global TCP wrappers and how this eBPF-focused configuration allows for optimized control that circumvents the issue Global TCP wrappers had that required explicit application support from a system administrator. This work demonstrates the flexibility of eBPF to provide security efficiently.

Van Tu et al. [3] details an architecture hybridizing eBPFs for network virtualization that is another important factor to account for in the modern network architecture. Utilizing the flexibility of eBPF in the kernel space, the architecture adds a user facing program to parse through the eBPF packet to build a Virtual Network Function (VNF) which is a piece of software that mocks hardware appliances like routers, load balancers, etc. Through this hybridized approach, the architecture leverages the speed advantages of eBPF in the kernel space while allowing for user space flexibility. This is demonstrated in the paper through the construction of two VNFs, a load balancer, and Deep Packet Inspection utilizing the proposed architecture.

Bertone et al. [4] uses eBPFs to make a simple eBPF-based iptables prototype to demonstrate the usefulness of eBPFs for the purposes of scalability. As stated earlier, the issue with the traditional iptables is that iptable rules are not scalable for modern requirements. Contrasting the eBPF-based Cilium that is geared to support a microservices ecosystem, the BPF-Iptables prototype is more focused on the just the scaling benefits of eBPFs. Therefore, the prototype is traditional in filtering traffic based on IP addresses and has less features overall than iptables. The results shown in the testing favored the BPF-Iptables prototype performing TCP traffic at an order of magnitude higher than iptables when scaling up to use a high number of rules. Thus, showing how even eBPF alone provides significant performance benefits even with IP based network filtering.

From another view on containerized systems in [5], Makowski et al. focused on the viability of the modern container networking systems to support wider adoption that ranges from scientific applications to digital marketplaces. The paper looks at the building blocks that involve Cilium to allow for container networking with multi-tenancy or allowing different virtualized systems to be contained inside the same machine. I found that this work approaches a holistic view of containerized security than just container packet filtering focusing on a larger use case to support widespread adoption with the advancements that Cilium also contributes to.

7. CONCLUSIONS

The project evaluated the usability of Cilium policies for security and performance overhead between Cilium and iptables. The results show that Cilium is more flexible and effective in securing containerized workloads with some performance overhead compared to iptables. In this paper, there is no insight for the causes of overhead and performance testing for running more substantive applications. Further research can investigate optimizing the connections to lower the overhead. Also, one can test different methods of network filtering combining different security filtering that together closely compares with the features that Cilium has. Currently, comparing Cilium policies with iptables rules is unbalanced due to the functionality of API filtering handled in Cilium being handled instead by the programmer.

8. REFERENCES

[1] N de Bruijn. ebpf based networking. 2017.

[2] 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 1, pages 209–217. IEEE, 2018.

[3] Nguyen Van Tu, Kyungchan Ko, and James Won-Ki Hong. Architecture for building hybrid kernel-user space virtual network functions. In *2017 13th International Conference on Network and Service Management (CNSM)*, pages 1–6. IEEE, 2017.

[4] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110. ACM, 2018.

[5] Łukasz Makowski and Paola Grosso. Evaluation of virtualization and traffic filtering methods for container networks. *Future Generation Computer Systems*, 93:345–357, 2019.