


## Article

# Experimental Analysis of Security Attacks for Docker Container Communications

Haneul Lee <sup>1</sup>, Soonhong Kwon <sup>2</sup> and Jong-Hyoun Lee <sup>2,\*</sup> <sup>1</sup> Protocol Engineering Laboratory, Sejong University, Seoul 143-747, Republic of Korea<sup>2</sup> Department of Computer and Information Security & Convergence Engineering for Intelligent Drone, Sejong University, Seoul 143-747, Republic of Korea

\* Correspondence: jonghyoun@sejong.ac.kr

**Abstract:** Docker has become widely used as an open-source platform for packaging and running applications as containers. It is in the limelight especially at companies and IT developers that provide cloud services thanks to its advantages such as the portability of applications and being lightweight. Docker provides communication between multiple containers through internal network configuration, which makes it easier to configure various services by logically connecting containers to each other, but cyberattacks exploiting the vulnerabilities of the Docker container network, e.g., distributed denial of service (DDoS) and cryptocurrency mining attacks, have recently occurred. In this paper, we experiment with cyberattacks such as ARP spoofing, DDoS, and elevation of privilege attacks to show how attackers can execute various attacks and analyze the results in terms of network traffic, CPU consumption, and malicious reverse shell execution. In addition, by examining the attacks from the network perspective of the Docker container environment, we lay the groundwork for detecting and preventing lateral movement attacks that may occur between the Docker containers.

**Keywords:** containers; container-based virtualization; Docker; network security



**Citation:** Lee, H.; Kwon, S.; Lee, J.-H. Experimental Analysis of Security Attacks for Docker Container Communications. *Electronics* **2023**, *12*, 940. <https://doi.org/10.3390/electronics12040940>

Academic Editors: Juan M. Corchado, Byung-Gyu Kim, Carlos A. Iglesias, In Lee, Fuji Ren and Rashid Mehmood

Received: 11 December 2022

Revised: 9 February 2023

Accepted: 11 February 2023

Published: 13 February 2023



**Copyright:** © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

## 1. Introduction

With the outbreak of COVID-19 and digital transformation, developers are starting to use cloud computing environments to efficiently manage computing resources or to provision on-demand computing resources on the fly [1]. The most important technology to provide a cloud computing environment is virtualization technology, which solves the problems caused by the increase in the amount of data generated by various devices and provides fast software distribution, customized functions, and flexibility for users. Container technology, which is one of these virtualization technologies, is different from the existing virtual machine (VM), namespace, which provides a virtual independent space by separating the resources used by the container in one operating system kernel and the space where the container is isolated isolating processes using Cgroups, which allows resource control on the platform to run applications independently [2].

Docker, an open-source platform used to package and run applications as containers, has become an important platform used by more than half of developers for reasons such as software portability, application operation standardization, and efficient resource management [3]. However, security threats targeting Docker containers began to increase with the activities of malicious hacker teams such as TeamTNT, WatchDog, Kinsing, and Rocke [4]. They mainly infiltrate Docker container environments through misconfigured APIs for the purpose of DDoS or cryptocurrency mining. In the case of applications that are used by multiple users and acquire special privileges, such as Portainer, a web GUI tool to facilitate Docker container management, various attacks can occur in the Docker container environment [5,6]. In addition, as an attacker can enter the Docker container environment through the Docker API exposed on the Internet, the attacker can bypass the

security program and execute various secondary attacks. Further research is needed to detect such attacks from a container network perspective [7].

In [8], a Docker image vulnerability diagnostic system was introduced to analyze Docker images. The system checks Docker images when uploading or downloading the Docker images from a Docker image repository if any known vulnerabilities are included in the images. In [9], the authors introduced a dynamic analysis method to assess the security of Docker images based on their behavior. The dynamic analysis method was shown to complement the static analyses typically used for security assessments for Docker images. The work presented in [10] introduced a Docker image traceability and security detection system based on inheritance graphs, called ZeroDVS, where a basic image graph is built with 160 official Docker images published by Docker Hub. Based on the built image graph, ZeroDVS can identify known vulnerabilities of a Docker image. In [11], various types of base images were compared in terms of security severity.

Among recent container security mechanisms applicable to Docker, major ones addressing container network security are Docker-sec, LiCshield, and Lic-sec.

Docker-sec is a Docker security mechanism based on AppArmor. It creates an AppArmor profile for a container and interacts with the Docker engine to apply it. The default AppArmor profile protects the container only after it is initialized by RunC, but Docker-sec protects the container during its entire life cycle [12]. Docker-sec shows excellent performance in defending against privilege escalation attacks but cannot defend well against privilege escalation attacks on images in the form of Docker in Docker [13].

LiCshield is a framework to automatically apply AppArmor policies to all Docker containers. It tracks all kernel activity with a tool called SystemTap5 while the Docker daemon is running. It can create access rules, mount rules, link rules, and execution rules that Docker-sec cannot create [14]. However, since LiCshield does not restrict malicious functions and commands such as `/bin/bash`, and it cannot defend against all kernel attacks [13].

Lic-sec has been proposed by combining the advantages of Docker-sec and LiCshield. It can create access rules, mount rules, link rules, and execution rules, as well as an audit function, while providing excellent performance in defending against privilege escalation attacks. However, a defense against DoS attacks is not supported [13].

The previous works have provided only defense mechanisms or assumed possible security threats. In other words, no specific details for security threats in the Docker container environment have been published. Rather than proposing a new defense mechanism, in this paper, we present our actual experimental results showing that various attacks are still possible even in the latest Docker container environment. The main contributions of this paper are as follows:

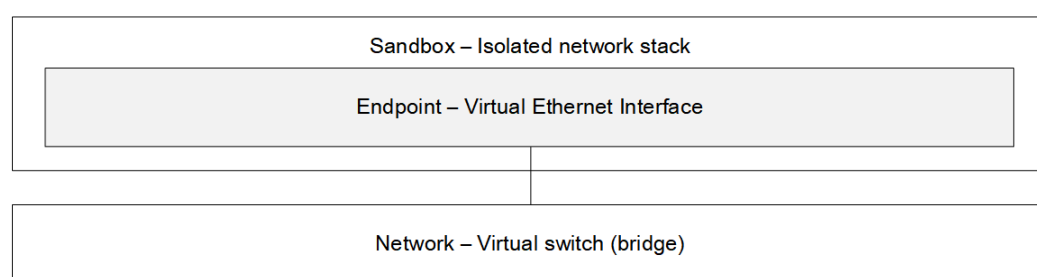
1. It mainly focuses on Docker container networking and presents actual security attacks to show how attackers can execute various attacks even in the latest Docker container environment.
2. It presents detailed experimental steps for executing attacks such as ARP spoofing, DDoS, and elevation of privilege attacks and presents analysis results of the security attacks in terms of network traffic, CPU consumption, and malicious reverse shell execution.

The structure of this paper is as follows. Section 2 describes the Docker container network configuration method based on a Docker container network methodology. Section 3 analyzes network security attacks between independent containers. Section 4 shows the experimental setup of this paper to demonstrate network security attacks in a Docker container environment. Section 5 analyzes the experimental results. Finally, Section 6 concludes this paper.

## 2. Docker Container Communication

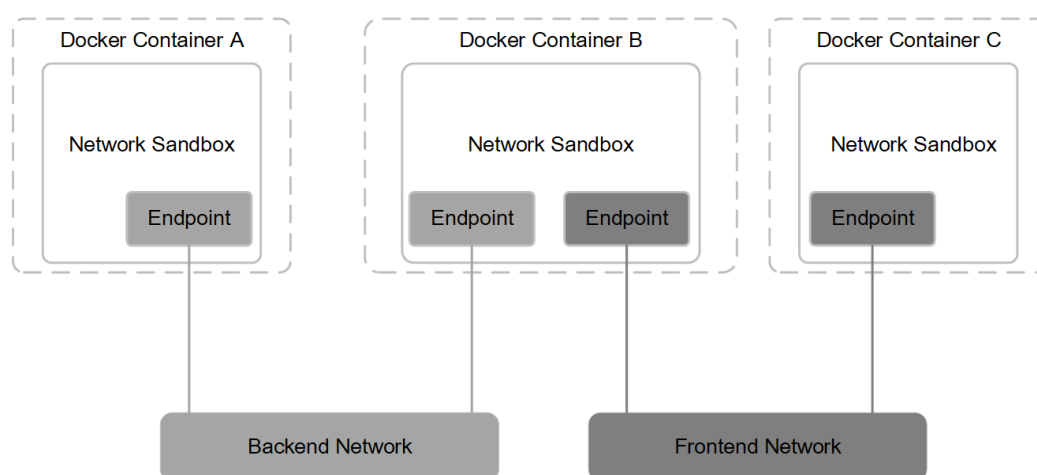
When we implement a specific service based on Docker, we usually configure the service by running an application in a container environment. When composing a service

based on the container, as communication between containers needs to be made, Docker configures the network environment for service configuration by connecting the container-to-container or existing network and VLAN. To clearly understand Docker networking, it is necessary to understand three concepts: container network model (CNM), `libnetwork`, and driver. First, the CNM is a standard for Docker networking and is composed of a sandbox, endpoint, and network, as shown in Figure 1. In the case of the sandbox in Figure 1, it can be defined as an isolated network stack, including Ethernet interfaces, ports, and DNS settings. Additionally, in the case of an endpoint, `veth`, which can be checked when configuring a Docker container network environment with a virtual Ethernet interface, corresponds to this. That is, the main role of the endpoint plays the role of a network interface, and in the CNM, it plays the role of connecting the sandbox to the network. A network is a software application implementation of one switch, and it plays the role of integrating or separating the endpoints for communication [15,16].



**Figure 1.** Overview of the container network model.

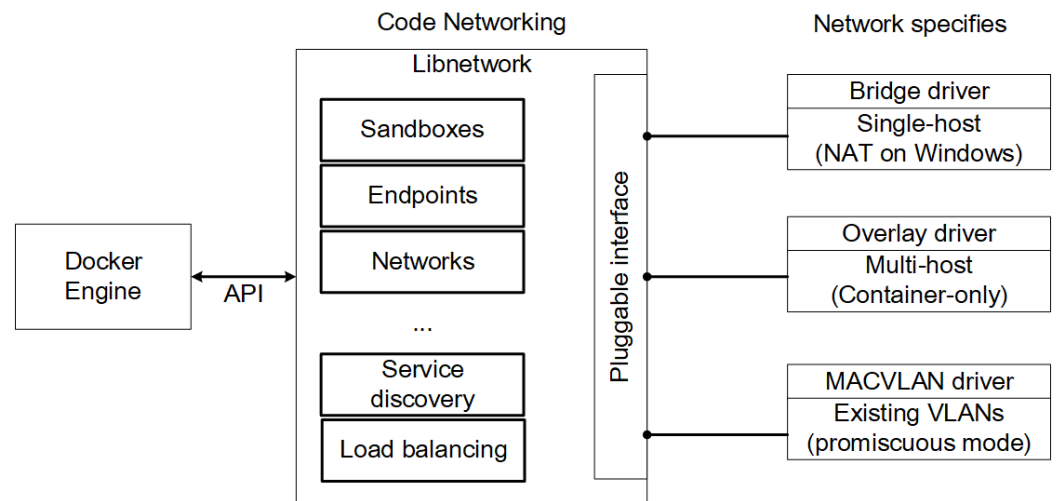
The roles of each component in the CNM are shown in Figure 2. In the case of Container A, it can be seen that the endpoint is connected to the back-end network, and Container B is connected to the back-end network and the front-end network based on the two endpoints. Additionally, the endpoint of Container C is connected to the front-end network. In the case of Figure 2, it can be seen that each container is connected to the back-end network and the front-end network, so communication between them is possible. However, in the case of the two endpoints of Container B, communication is impossible without a Layer 3 router, and communication is possible only through the Layer 3 router [15].



**Figure 2.** Container network model showing the back-end and front-end networks.

Next, in the case of `libnetwork`, it implements the CNM and performs all the roles of each component in the CNM. This can be explained as being responsible for native service discovery, ingress-based container load balancing, and network control and management plane functions. In the early days of Docker, these functions were supported by the Docker daemon, but now it has been refactored and established as `libnetwork` [15].

In the case of a driver, it can be explained compared to `libnetwork`, and if `libnetwork` provides network control and management plane functions, the driver implements the data plane. It can be seen that both the connection and isolation of the Docker container are handled by the driver, which can be seen in Figure 3. In the case of `libnetwork`, as cross-platform is supported, there is a difference in the driver depending on the Linux environment or Windows environment even in the case of the driver. On Linux, it has bridge, overlay, and macvlan drivers, and on Windows, it has nat, overlay, transparent, and 12bridge drivers. Each driver operates in the form of a plug-in and provides a role to manage as well as create all resources on the network [15].

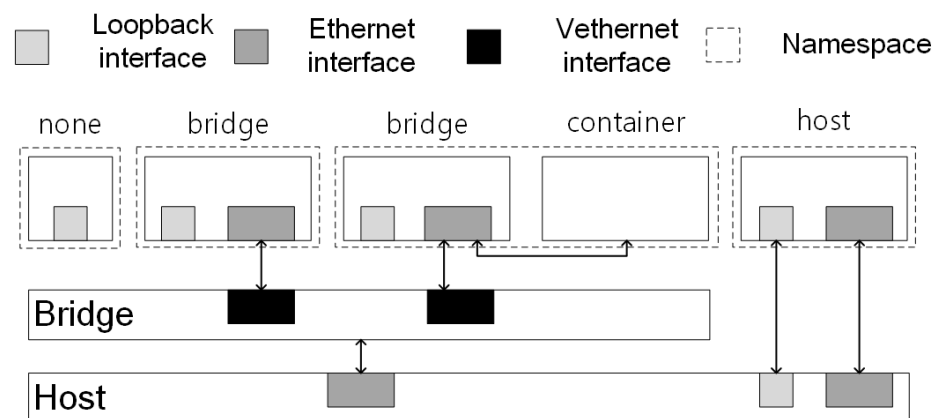


**Figure 3.** Driver components of the container network model.

Docker networking is based on these components, and more specifically, there is single-host communication between containers that share a single-host kernel and multi-host communication between containers on multiple hosts [15].

### 2.1. Single-Host Networking

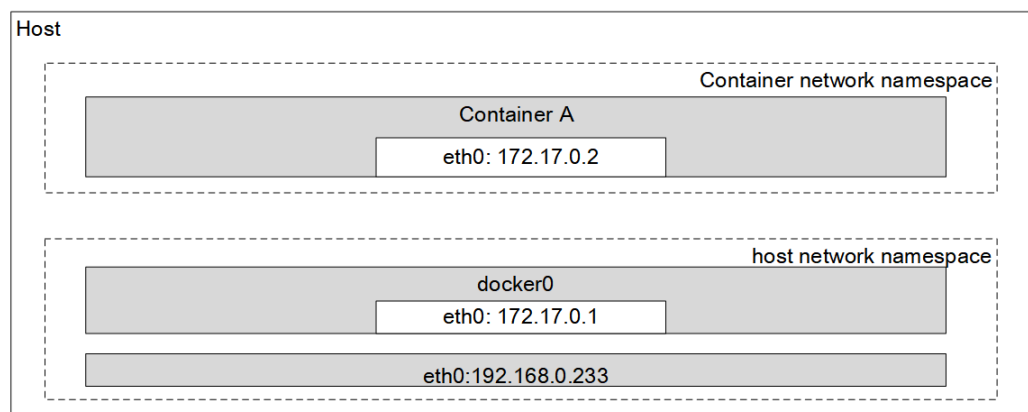
The single-host communication method refers to communication between containers that share the same host kernel. Communication methods include ‘None Mode’, ‘Bridge Mode’, ‘Container Mode’, and ‘Host Mode’ [17]. Figure 4 is a diagram schematically showing the structure of various communication methods appearing in a single-host communication method [18].



**Figure 4.** Single-host networking mode.

First, ‘None Mode’ is a mode that sets the container to a closed network. It cannot connect to other containers on the same host or on an external network, and because of its characteristics, it has the highest isolation and security level of communication methods.

'None Mode' is used for tasks that do not require network access operations such as data calculations, batch processing, or backup operations. Figure 5 shows the network structure of the 'None Mode' method in a single host [18].



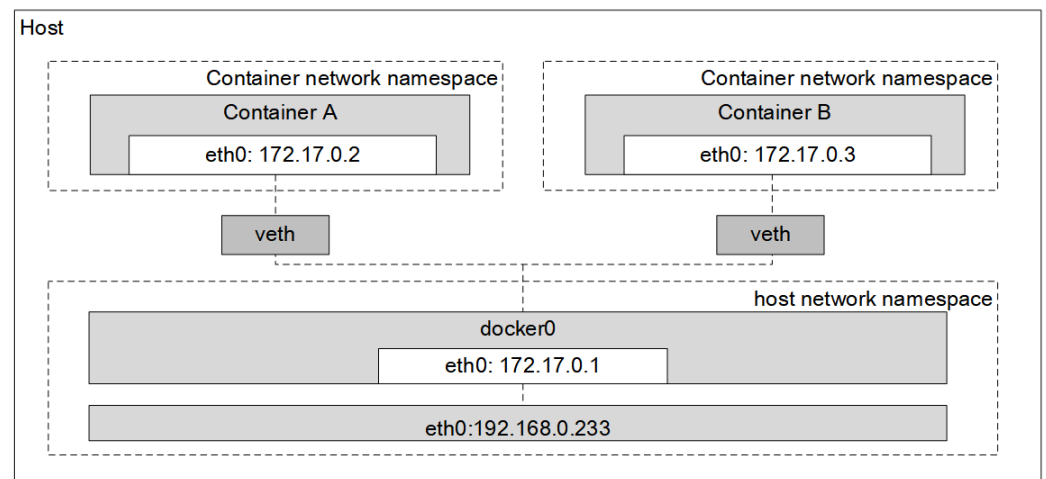
**Figure 5.** 'None Mode' communication method in a single host.

Second, 'Bridge Mode' is a mode in which two networks are used as one network by connecting a container to a host. This is the network mode set by default when a Docker container is created, and it communicates by creating `docker0` or a custom bridged network interface. Each container is isolated by a namespace, and they have a private IP address. Containers communicate by creating a virtual interface `veth` to connect to the bridge network interface that they must go through to communicate with each other. Although each container in 'Bridge Mode' is isolated, it provides relatively weak security compared to 'None Mode' because it is connected by a bridge. Figure 6 shows the 'Bridge Mode' network structure in a single host [19].

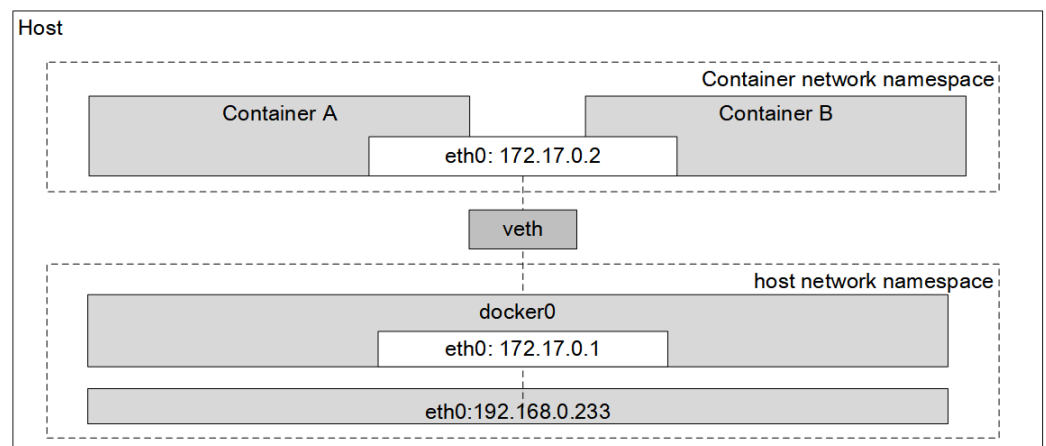
The third 'Container Mode' is a mode to communicate by sharing the namespace of the proxy container. Because containers sharing a single namespace use a common IP, they communicate with each other using Inter-Process Communication (IPC) and can be identified through port numbers. Communication with other containers is possible through 'Bridge Mode'. In 'Container Mode', isolation from groups using different namespaces is high, but isolation between containers using the same group is low. Figure 7 shows the 'Container Mode' communication method in a single host [20].

Finally, 'Host Mode' is the mode in which the container uses the namespace of the host OS. It removes the network isolation between the container and the host and uses the host's network directly. Therefore, this mode provides the lowest level of security. However, communication is as easy as exchanging processes with other hosts. Figure 8 shows the 'Host Mode' communication method in a single host [20].

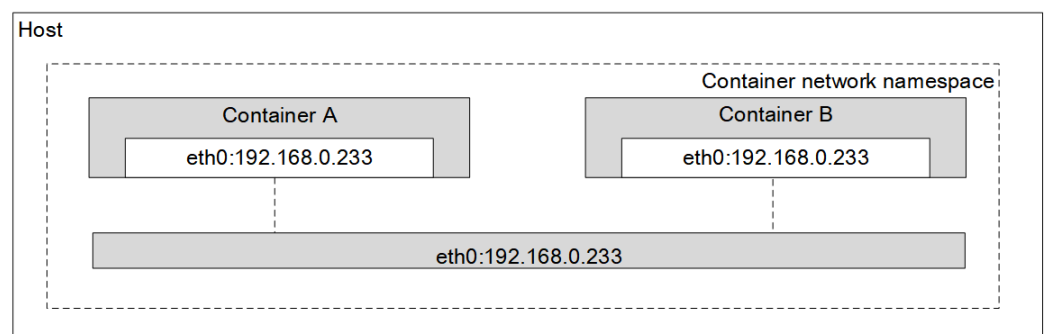
Table 1 summarizes the characteristics of the communication method between containers on a single host according to the method in which each namespace is divided, the type of 'Network Interface/Network Driver' used, and the security strength.



**Figure 6.** 'Bridge Mode' communication method in a single host.



**Figure 7.** 'Container Mode' communication method in a single host.



**Figure 8.** 'Host Mode' communication method in a single host.

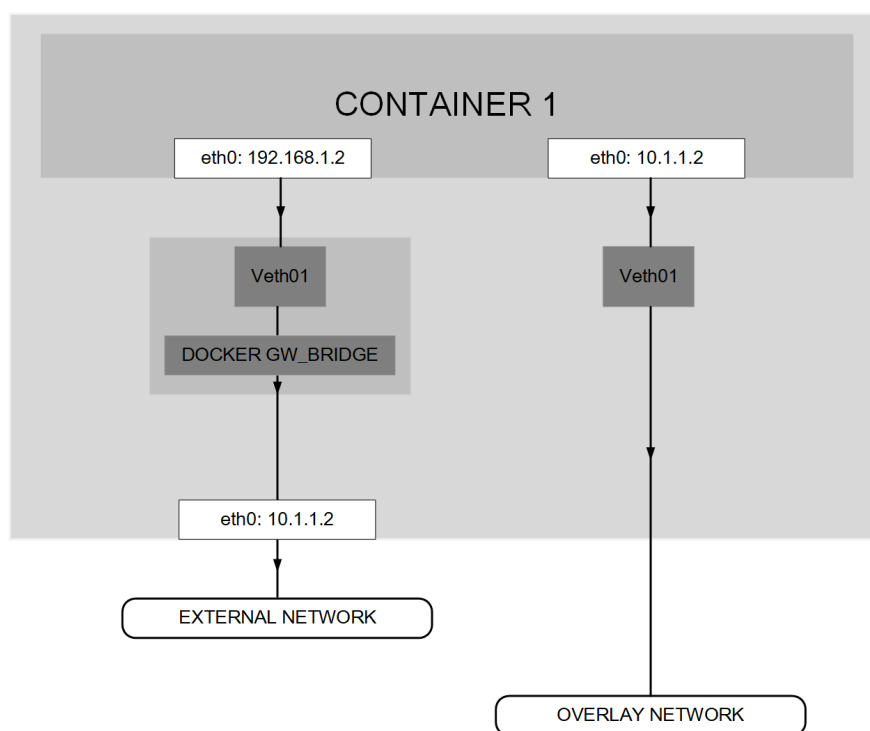
## 2.2. Multi-Host Networking

Docker was developed for the purpose of networking between containers on a single host, so there were difficulties in container communication between different hosts. To solve this problem, solutions such as Calico and Weave began to appear, and after Docker version 1.9, Docker itself began to support overlay networking. Each support method can be selected according to the developer's requirements.

**Table 1.** Comparison according to communication method in a single host.

Mode	Namespace	Network Interface	Network Driver	Security
None	Isolated	-	-	High
Bridge	Isolated	docker0/user-defined bridge interface	Bridge	Medium
Container	Connected with proxy container	Interface of proxy container	-	Medium
Host	Connected with host	Host	Host	Low

First, a Docker overlay network plays a role of integrating into one network for containers running on multiple Docker daemons on one host or multiple Docker daemons on different hosts. In more detail, the Docker overlay network can be defined as a virtual networking solution using `libnetwork` and `libk`. It solves the scalability problem by using Virtual extensible LAN (VxLAN), an encapsulation protocol built into the `libnetwork` library. All management traffic is encrypted using AES algorithm by default, and application data encryption protects messages with IPsec on VxLAN [21]. Figure 9 shows an example of a Docker overlay network.

**Figure 9.** Docker overlay network.

Second, Weave is a virtual networking solution developed by Weaveworks. It is configured as a Weave Net router and communicates by creating a veth-bridge network interface on each host to connect containers. Weave also uses the VxLAN encapsulation protocol, which allows for IP lookups of other containers using DNS queries for container names. In addition, the message is protected using the Networking and Cryptography library (NaCI), which provides network communication, encryption, decryption, and signature [22]. The fast data path in Weave uses the Open vSwitch module to announce to the kernel how to process packets. In addition, in the case of Weave NET, a method of issuing commands directly to the kernel is adopted, and context switching is reduced by using a fast data path, and there are advantages in terms of CPU overhead and latency efficiency. Figure 10 shows a general overview of Weave [22].

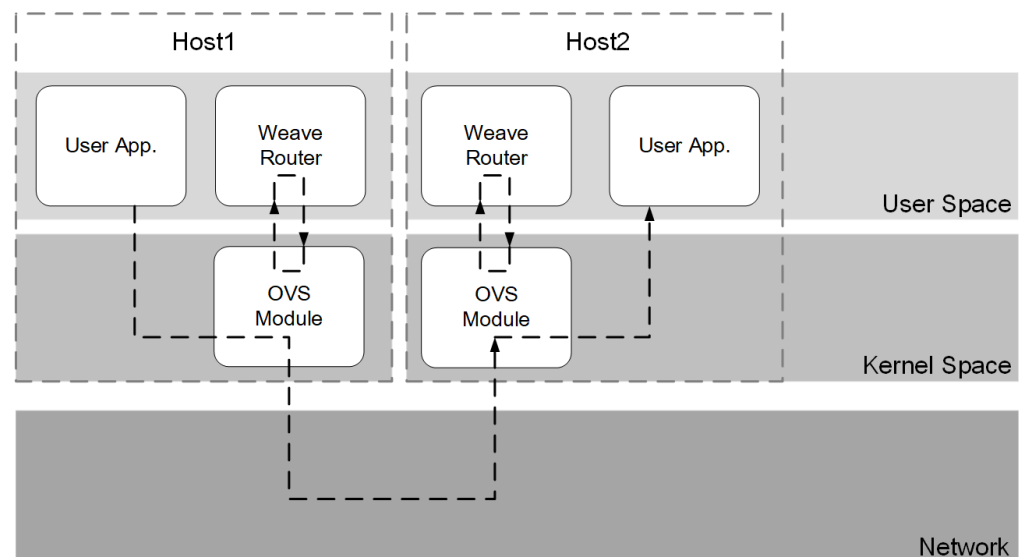


Figure 10. Overview of Weave.

Finally, Calico is a virtual networking solution developed by Tigera. It uses two types of encapsulation, IP in IP (IPIP) and VxLAN, to explore routes to individual containers. IPIP is an IP tunneling protocol that encapsulates one IP packet in another IP packet, and routing information is exchanged between Calico nodes using the Border Gateway Protocol (BGP). VxLAN is applied to some environments such as Azure where IPIP is not applied and BGP is not used. Figure 11 shows an example of multi-host networking configuration for MySQL containers based on Calico [23].

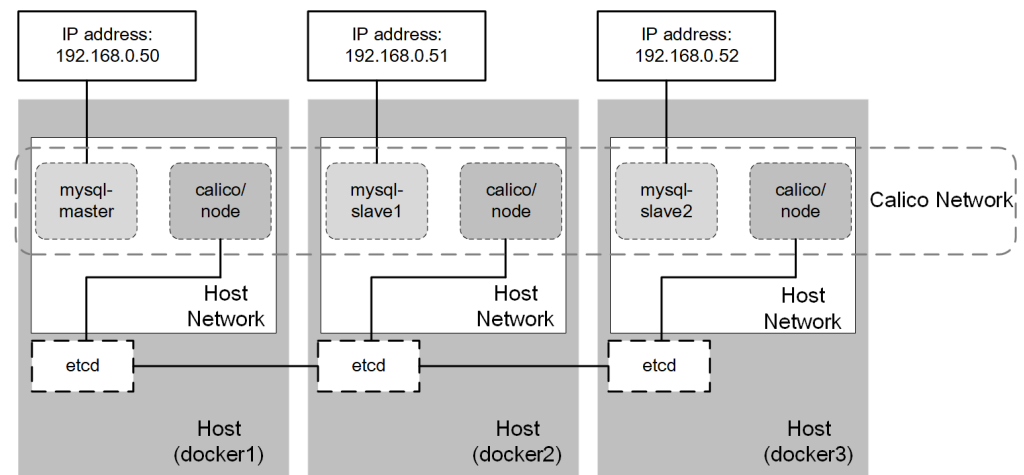


Figure 11. Multi-host networking configuration for MySQL containers based on Calico.

Table 2 is a table summarizing the characteristics of virtual networking solutions between containers on different hosts. It indicates the encapsulation protocol used, whether the name service is supported, the encryption channel used, and the supported protocol types.

Table 2. Characteristics of virtual networking solutions [18].

Overlay	Encapsulation Protocol	Name Service Support	Encryption Channel	Protocol Support
Docker overlay	VxLAN	N	N	ALL
Weave	VxLAN	Y	Y	ALL
Calico	VxLAN/IPIP	N	N	TCP, UDP, ICMP

### 3. Security Attacks Exploiting the Docker Container Network

In this section, we present our analysis of the network security attacks associated with communications between Docker containers. Container network security challenges can be expressed as the five challenges shown in Figure 12 below [24].

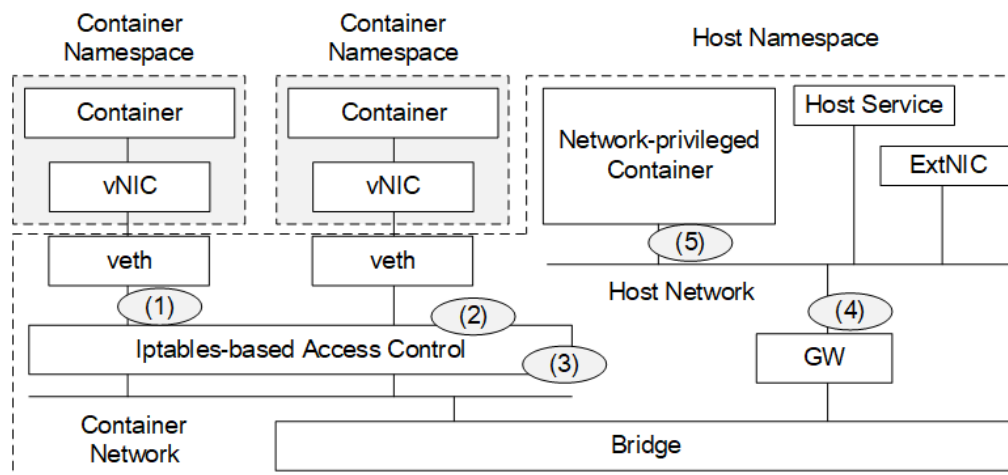


Figure 12. Container network security challenges [24].

The description of challenges shown in Figure 12 is summarized in Table 3 below.

(1) Each container has its own virtual network interface through which Docker can bridge other containers. However, the security challenge is that inspection of packets can only be performed in each container, and packet inspection in the host network namespace must rely on packet header information. If a malicious container exists on the bridge, this container can forge packets of other containers and enables lateral movement, making it impossible to inspect normal container packets.

(2) Containers have dynamic IP changes using Dynamic Host Configuration Protocol (DHCP), making it difficult to update security policies using iptables and are vulnerable to Layer 2 attacks such as ARP spoofing.

(3) Each container is subdivided and a security policy suitable for the container is required. iptables is a centralized management mechanism for host network interfaces. When a large number of containers are created, management of each container's security policy becomes difficult and performance suffers [25].

(4) Container networks have an external gateway interface. Containers can access that gateway, which allows them to access services running on the host and also access other connected hosts. If a malicious container exists on container network, the malicious container may perform an attack that violates the availability of the host, such as DoS.

(5) In the case of a container that has network authority over the host through Host Mode, the container's isolation is low, so it can monitor the traffic of all containers connected to the host and inject malicious attacks.

Table 4 describes the network security threats that can occur between containers among the security problems that can occur on the above Docker container network and the resulting attacks [26].

#### 3.1. Poor Independence between Containers

The network connection between Docker containers can weaken the independent container characteristics. For this reason, attacks such as DDoS and man-in-the-middle (MITM) attacks can occur. A DDoS attack is one of the availability breach attacks that depletes the victim's computer resources and prevents normal operation. When a malicious container is inserted into the Docker network and connected to other containers through a network, it shares the same CPU core resource on a single host and may infringe the area of other connected containers. An attacker may use this maliciously to lead to a DDoS attack that consumes service resources. In an MITM attack, when two or more containers sharing the same network interface communicate with each other on a single host, assuming that

there is a malicious container using the same network interface, the malicious container can intervene in communication between the two containers. Typical MITM attacks include sniffing that eavesdrops on communications, ARP spoofing that pretends to be another user, and tampering that can modify information in other containers without permission [27].

**Table 3.** Challenges in Docker container networks [24].

No.	Challenges	Description
1	Loss of container context	Malicious containers can forge packet header information through the bridge network, enabling attacks such as spoofing
2	Limitations of IP-based access control	Because of the dynamic change of container IP, updating the policy table of iptables can be difficult and vulnerable to Layer 2 attacks
3	Network policy explosion	As each container requires a different security policy setting the security policy through the centralized mechanism iptables can result in severe performance degradation
4	Unrestricted host access	A container can access it through a gateway for external access connected to the host network, allowing lateral movement to access other containers
5	No restriction on network privileges container	It monitors the network traffic of all containers connected to the host and can inject malicious attacks

**Table 4.** Network security threats and related attacks for an inter Docker container.

Threats	Attacks	Description	Ref.
Poor independence between Docker containers	DDoS	A malicious Docker container depletes the resource of another container	[28–31]
	Sniffing	Communication between two containers can be eavesdropped	-
	ARP Spoofing	ARP spoofing is possible for containers running in the kernel	[29–31]
	Tampering	In the case of host mode, a malicious Docker container can modify information because the Docker container user the same namespace as the host	[29–31]
		In the case of container mode, information of containers sharing a namespace can be modified	[29–31]
Lack of security protocols	Sniffing	Unencrypted packets can be eavesdropped	-
	Tampering	Unencrypted packets can be modified	-
	Privilege Escalation	A malicious Docker container can access the endpoint of a Rest API, allowing elevation of privilege	[28–31]

### 3.2. Lack of Security Protocols

The Docker communication system consists of a client–server structure. It is represented as a Docker client and a Docker daemon, respectively, which communicate with each other via the REST API. As the REST API uses the HTTP protocol, there is a possibility that it may be exposed to vulnerabilities in the protocol. Since the HTTP protocol is weak in security as security protocols such as TLS/SSL are not applied, MITM attacks such as sniffing and tampering are easy, and further security attacks that infringe confidentiality and integrity may occur.

In addition, the firewall of the REST API protects the intrusion of external attackers, but does not respond to security threats that occur inside when a malicious container is inserted. Recently, privilege escalation attacks using this vulnerability have occurred

frequently, and through this, cases where the target host is exploited for cryptocurrency mining are increasing [32].

#### 4. Experimental Setup

We now demonstrate some of the Docker network security attacks described in Section 3. We use Ubuntu version 20.04.3, and the experiments are carried out in Docker version 20.10.12. The host IP and container addresses are shown in Table 5.

**Table 5.** Summary of Docker network security threats [26].

Attack	OS	Docker	Classification	Host IP	Container	IP Address
ARP spoofing	Ubuntu 20.04.3	Docker 20.10.12	Target host	192.168.19.130	busybox1	172.17.0.2
					busybox2	172.17.0.3
					ARP spoofer	172.17.0.4
DDoS			MySQL	172.18.0.2		
				phpMyAdmin	172.18.0.3	
				Ubuntu	172.18.0.4	
Privilege escalation			Malicious host	192.168.19.129	my_registry	172.17.0.2
					ubuntu_attacker	172.17.0.3
				Target host	192.168.19.130	ubuntu_target

At this time, the MAC address of busybox1 is 02:42:ac:11:00:02, and the MAC address of busybox2 is 02:42:ac:11:00:03. Additionally, the MAC address of ARP Spoofer is 02:42:ac:11:00:04. The configuration of environment information for the DDoS attack consists of the MySQL container required for web construction, the phpMyAdmin container acting as the victim, and 20 Ubuntu containers acting as the attacker. After preparing two hosts, malicious host and target host, for the containers required for privilege elevation, the Ubuntu container required for attack on the malicious host, the registry to upload the image, and the Ubuntu container required for the target host to be attacked are prepared.

The ARP spoofing, DDoS, and privilege escalation attacks are performed based on the experimental environment specified in Table 5, and the main outline of each attack execution is as follows.

- ARP spoofing

For the ARP spoofing experiment in communication between Docker containers, it is assumed that a malicious container ARP spoofer exists on a single host. The ARP spoofer intercepts communication between busybox1 and busybox2 and disguises its IP as the IP of busybox1 and busybox2. Figure 13 shows the environment of the ARP spoofing experiment.

- DDoS

To experiment with a DDoS attack in Docker container-to-container communication, it is assumed that a malicious container exists on a single host. In the experiment, when the phpMyAdmin container and the MySQL container communicate with each other, 20 malicious Ubuntu containers simultaneously conduct a SYN flood attack through the hping3 tool. Figure 14 is a diagram showing the corresponding experimental environment.

- Privilege escalation

For privilege escalation experiments in communication between Docker containers, experiments are conducted on multiple hosts. First we need to set up a malicious host, which is the host of the attacker, and the target host, which is the host of the victim. The malicious host uploads a malicious Ubuntu image file to its registry my\_registry and attacks the target host through its own Ubuntu. The target host downloads the malicious ubuntu image file from the attacker's registry. When the target host executes the ubuntu

file, the reverse shell is executed. Figure 15 shows the experimental environment briefly, and Figure 16 shows the overall workflow between the attacker and the victim in detail.

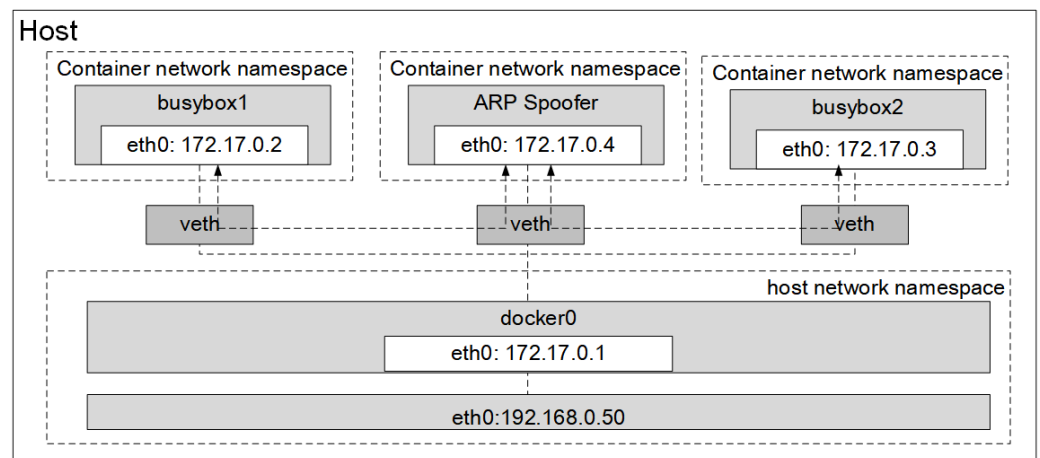


Figure 13. Experimental environment for the ARP spoofing.

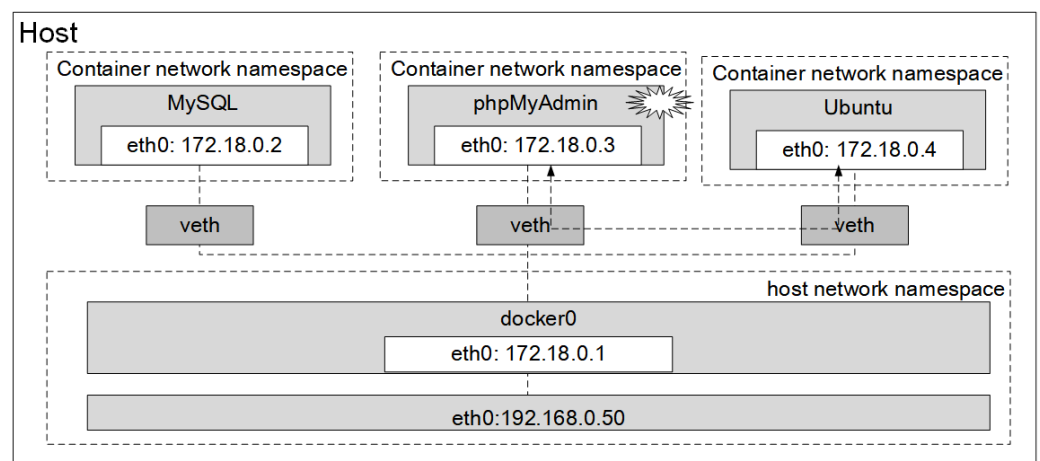


Figure 14. DDoS experimental environment.

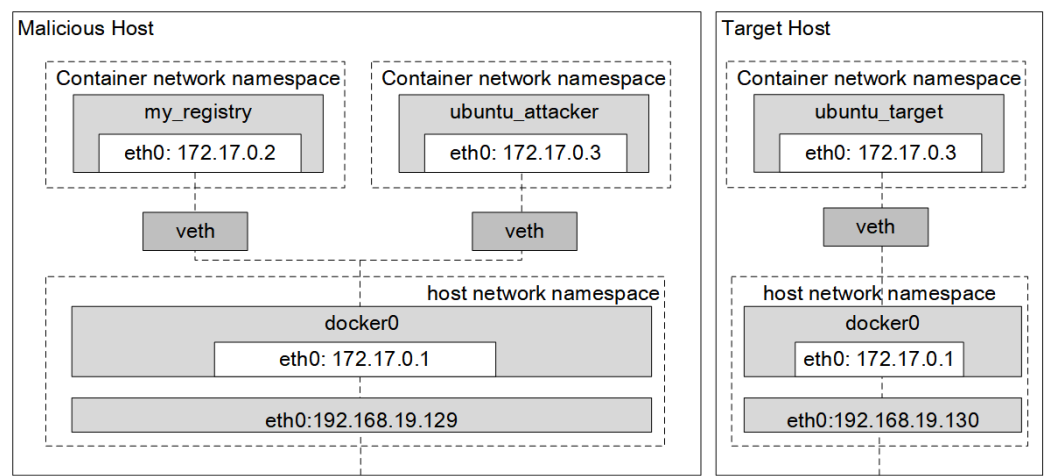
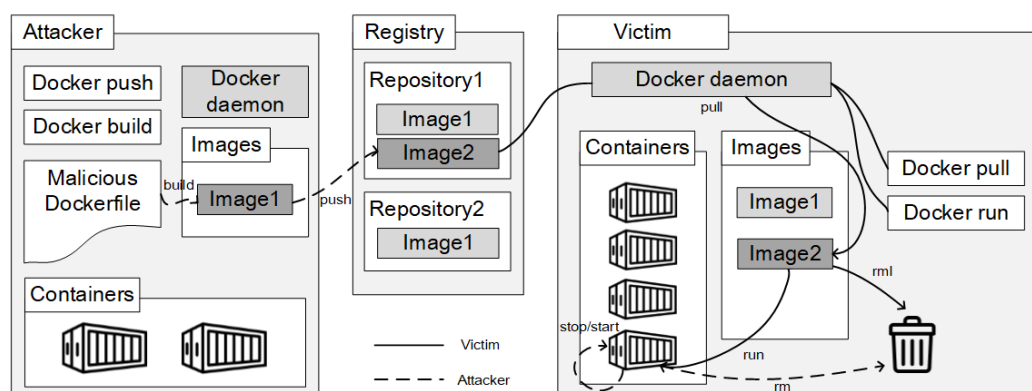


Figure 15. Experimental environment of privilege escalation.



**Figure 16.** Overall attack and victim workflow.

## 5. Analysis of the Attack Results

- ARP spoofing

The ARP spoofing attack is performed based on the attack method configured in Section 4. First, in order to determine the success of the ARP spoofing attack, the packets of busybox1 and busybox2 performing general ICMP communication are checked using Wireshark. From Figure 17, it is confirmed that busybox1 and busybox2 communicate normally with source and destination MAC addresses of 02:42:ac:11:00:02 and 02:42:ac:11:00:03, respectively.

No.	Time	Source	Destination	Protocol	Length	Info
+	1	0.000000000	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=21/5376, ttl=64
	2	0.000125999	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=21/5376, ttl=64
	3	1.001611443	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=22/5632, ttl=64
	4	1.001734442	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=22/5632, ttl=64
	5	2.003897079	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=23/5888, ttl=64
	6	2.004021279	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=23/5888, ttl=64
	7	3.006975215	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=24/6144, ttl=64
	8	3.007116914	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=24/6144, ttl=64
	9	4.009710953	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=25/6400, ttl=64
	10	4.009834452	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=25/6400, ttl=64
	11	5.011840493	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=26/6656, ttl=64
	12	5.011962692	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=26/6656, ttl=64
	13	6.017815116	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=27/6912, ttl=64
▶ Frame 1: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface veth81e2501 id 0						
▶ Ethernet II, Src: 02:42:ac:11:00:03 (02:42:ac:11:00:03), Dst: 02:42:ac:11:00:02 (02:42:ac:11:00:02)						
▶ Internet Protocol Version 4, Src: 172.17.0.3, Dst: 172.17.0.2						
▶ Internet Control Message Protocol						
No.	Time	Source	Destination	Protocol	Length	Info
+	1	0.000000000	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=21/5376, ttl=64
	2	0.000125999	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=21/5376, ttl=64
	3	1.001611443	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=22/5632, ttl=64
	4	1.001734442	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=22/5632, ttl=64
	5	2.003897079	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=23/5888, ttl=64
	6	2.004021279	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=23/5888, ttl=64
	7	3.006975215	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=24/6144, ttl=64
	8	3.007116914	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=24/6144, ttl=64
	9	4.009710953	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=25/6400, ttl=64
	10	4.009834452	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=25/6400, ttl=64
	11	5.011840493	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=26/6656, ttl=64
	12	5.011962692	172.17.0.2	ICMP	98	Echo (ping) reply id=0x0012, seq=26/6656, ttl=64
	13	6.017815116	172.17.0.3	ICMP	98	Echo (ping) request id=0x0012, seq=27/6912, ttl=64
▶ Frame 2: 98 bytes on wire (784 bits), 98 bytes captured (784 bits) on interface veth81e2501 id 0						
▶ Ethernet II, Src: 02:42:ac:11:00:02 (02:42:ac:11:00:02), Dst: 02:42:ac:11:00:03 (02:42:ac:11:00:03)						
▶ Internet Protocol Version 4, Src: 172.17.0.2, Dst: 172.17.0.3						
▶ Internet Control Message Protocol						

**Figure 17.** General ICMP communication process.

However, when the ARP spoofer is involved in inter-container communication, if busybox1 (02:42:ac:11:00:02) and busybox2 (02:42:ac:11:00:03) communicate with each other, the destination MAC address 02:42:ac:11:00:04 can be seen through Wireshark. On the IP address, it is confirmed as the original communication between busybox1 and busybox2, but when it is checked through the MAC address, it can be seen from Figure 18 that the MITM attack that ARP spoofer intervened in is executed.

- DDoS

As previously described in Section 4, in the case of a DDoS attack, the attack is performed based on 20 Ubuntu containers and is performed based on the hping3 tool.

Figure 19 shows that the hping3 command executed in each Ubuntu container acting as an attacker performs a SYN flooding attack on port 80 of the container whose IP address is 172.18.0.3. In the case of attacking the target by executing the command in Figure 19 in the 20 Ubuntu containers, we can confirm the network traffic through Wireshark as shown in Figure 20. It can be seen that when phpMyAdmin and MySQL communicate with each other, the Ubuntu container continuously sends the SYN messages to the target container, phpMyAdmin, and the Ubuntu container sends the RST messages to phpMyAdmin after receiving the SYN and ACK messages in response. Additionally, we can confirm that the traffic is classified as malicious (bad) TCP traffic in Figure 21.

No.	Time	Source	Destination	Protocol	Length	Info
12	1.043778127	02:42:ac:11:00:04	02:42:ac:11:00:02	ARP	42	172.17.0.3 is at 02:42:ac:11:00:04 (duplicate us
13	2.001905499	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=37/9472, tt
14	2.001938549	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=37/9472, tt
15	2.001978221	172.17.0.2	172.17.0.3	ICMP	98	Echo (ping) reply id=0x0011, seq=37/9472, tt
16	2.001987158	172.17.0.2	172.17.0.3	ICMP	98	Echo (ping) reply id=0x0011, seq=37/9472, tt
17	3.002873425	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=38/9728, tt
18	3.002890579	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=38/9728, tt
19	3.002910895	172.17.0.2	172.17.0.3	ICMP	98	Echo (ping) reply id=0x0011, seq=38/9728, tt
20	3.002914579	172.17.0.2	172.17.0.3	ICMP	98	Echo (ping) reply id=0x0011, seq=38/9728, tt
21	3.044825091	02:42:ac:11:00:04	02:42:ac:11:00:03	ARP	42	172.17.0.2 is at 02:42:ac:11:00:04
22	3.044860541	02:42:ac:11:00:04	02:42:ac:11:00:02	ARP	42	172.17.0.3 is at 02:42:ac:11:00:04 (duplicate us
23	4.003512471	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=39/9984, tt
24	4.003537056	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=39/9984, tt

No.	Time	Source	Destination	Protocol	Length	Info
12	1.043778127	02:42:ac:11:00:04	02:42:ac:11:00:02	ARP	42	172.17.0.3 is at 02:42:ac:11:00:04 (duplicate us
13	2.001905499	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=37/9472, tt
14	2.001938549	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=37/9472, tt
15	2.001978221	172.17.0.2	172.17.0.3	ICMP	98	Echo (ping) reply id=0x0011, seq=37/9472, tt
16	2.001987158	172.17.0.2	172.17.0.3	ICMP	98	Echo (ping) reply id=0x0011, seq=37/9472, tt
17	3.002873425	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=38/9728, tt
18	3.002890579	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=38/9728, tt
19	3.002910895	172.17.0.2	172.17.0.3	ICMP	98	Echo (ping) reply id=0x0011, seq=38/9728, tt
20	3.002914579	172.17.0.2	172.17.0.3	ICMP	98	Echo (ping) reply id=0x0011, seq=38/9728, tt
21	3.044825091	02:42:ac:11:00:04	02:42:ac:11:00:03	ARP	42	172.17.0.2 is at 02:42:ac:11:00:04
22	3.044860541	02:42:ac:11:00:04	02:42:ac:11:00:02	ARP	42	172.17.0.3 is at 02:42:ac:11:00:04 (duplicate us
23	4.003512471	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=39/9984, tt
24	4.003537056	172.17.0.3	172.17.0.2	ICMP	98	Echo (ping) request id=0x0011, seq=39/9984, tt

Figure 18. General ICMP communication process with the ARP spoofing attack.

```
root@2a7b0e3d8f9d:/# hping3 -S 172.18.0.3 -p 80 -flood
```

Figure 19. hping3 command for the SYN flooding attack.

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	172.18.0.4	172.18.0.3	TCP	54	47717 → 80 [RST] Seq=1 Win=0 Len=0
2	0.000076700	172.18.0.4	172.18.0.3	TCP	54	47718 → 80 [SYN] Seq=0 Win=512 Len=0
3	0.000143400	172.18.0.3	172.18.0.4	TCP	58	80 → 47718 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
4	0.000227499	172.18.0.4	172.18.0.3	TCP	54	47718 → 80 [RST] Seq=1 Win=0 Len=0
5	0.000592898	172.18.0.3	172.18.0.4	TCP	58	80 → 47719 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
6	0.000623498	172.18.0.4	172.18.0.3	TCP	54	47719 → 80 [RST] Seq=1 Win=0 Len=0
7	0.000787598	172.18.0.4	172.18.0.3	TCP	54	47720 → 80 [SYN] Seq=0 Win=512 Len=0
8	0.000853097	172.18.0.3	172.18.0.4	TCP	58	80 → 47720 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
9	0.000881997	172.18.0.4	172.18.0.3	TCP	54	47720 → 80 [RST] Seq=1 Win=0 Len=0
10	0.001166096	172.18.0.4	172.18.0.3	TCP	54	47721 → 80 [SYN] Seq=0 Win=512 Len=0
11	0.001235296	172.18.0.3	172.18.0.4	TCP	58	80 → 47721 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
12	0.001264406	172.18.0.4	172.18.0.3	TCP	54	47721 → 80 [RST] Seq=1 Win=0 Len=0
13	0.001341396	172.18.0.4	172.18.0.3	TCP	54	47722 → 80 [SYN] Seq=0 Win=512 Len=0
14	0.001405696	172.18.0.3	172.18.0.4	TCP	58	80 → 47722 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
15	0.001434596	172.18.0.4	172.18.0.3	TCP	54	47722 → 80 [RST] Seq=1 Win=0 Len=0
16	0.001500696	172.18.0.4	172.18.0.3	TCP	54	47723 → 80 [SYN] Seq=0 Win=512 Len=0
17	0.001572095	172.18.0.3	172.18.0.4	TCP	58	80 → 47723 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
18	0.001601295	172.18.0.4	172.18.0.3	TCP	54	47723 → 80 [RST] Seq=1 Win=0 Len=0
19	0.001669395	172.18.0.4	172.18.0.3	TCP	54	47724 → 80 [SYN] Seq=0 Win=512 Len=0
20	0.001734795	172.18.0.3	172.18.0.4	TCP	58	80 → 47724 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
21	0.001763795	172.18.0.4	172.18.0.3	TCP	54	47724 → 80 [RST] Seq=1 Win=0 Len=0
22	0.001831294	172.18.0.4	172.18.0.3	TCP	54	47725 → 80 [SYN] Seq=0 Win=512 Len=0
23	0.001904894	172.18.0.3	172.18.0.4	TCP	58	80 → 47725 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
24	0.001933994	172.18.0.4	172.18.0.3	TCP	54	47725 → 80 [RST] Seq=1 Win=0 Len=0
25	0.002011194	172.18.0.4	172.18.0.3	TCP	54	47726 → 80 [SYN] Seq=0 Win=512 Len=0

Figure 20. Wireshark execution screen with the DDoS attack.

As a result of observing the amount of accumulated traffic through the Wireshark I/O graph, it can be observed that there is no change in terms of packets per second (PPS), as shown in Figure 22 when the DDoS attack is not executed.

2429_14.998009799	172.18.0.4	172.18.0.3	TCP	54 63169 → 80 [RST] Seq=1 Win=0 Len=0
2429_14.998102497	172.18.0.4	172.18.0.3	TCP	54 [TCP Port numbers reused] 63170 → 80 [SYN] Seq=0 Win
2429_14.998186996	172.18.0.3	172.18.0.4	TCP	58 80 → 63170 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
2429_14.998223796	172.18.0.4	172.18.0.3	TCP	54 63170 → 80 [RST] Seq=1 Win=0 Len=0
2429_14.998315495	172.18.0.4	172.18.0.3	TCP	54 [TCP Port numbers reused] 63171 → 80 [SYN] Seq=0 Win
2429_14.998401693	172.18.0.3	172.18.0.4	TCP	58 80 → 63171 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
2429_14.998437693	172.18.0.4	172.18.0.3	TCP	54 63171 → 80 [RST] Seq=1 Win=0 Len=0
2429_14.998523892	172.18.0.4	172.18.0.3	TCP	54 [TCP Port numbers reused] 63172 → 80 [SYN] Seq=0 Win
2429_14.998609491	172.18.0.3	172.18.0.4	TCP	58 80 → 63172 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
2429_14.998645591	172.18.0.4	172.18.0.3	TCP	54 63172 → 80 [RST] Seq=1 Win=0 Len=0
2429_14.998731090	172.18.0.4	172.18.0.3	TCP	54 [TCP Port numbers reused] 63173 → 80 [SYN] Seq=0 Win
2429_14.998815688	172.18.0.3	172.18.0.4	TCP	58 80 → 63173 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
2429_14.998852188	172.18.0.4	172.18.0.3	TCP	54 63173 → 80 [RST] Seq=1 Win=0 Len=0
2429_14.998941687	172.18.0.4	172.18.0.3	TCP	54 [TCP Port numbers reused] 63174 → 80 [SYN] Seq=0 Win
2429_14.999025786	172.18.0.3	172.18.0.4	TCP	58 80 → 63174 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
2429_14.999061785	172.18.0.4	172.18.0.3	TCP	54 63174 → 80 [RST] Seq=1 Win=0 Len=0
2429_14.999145484	172.18.0.4	172.18.0.3	TCP	54 [TCP Port numbers reused] 63175 → 80 [SYN] Seq=0 Win
2429_14.999231083	172.18.0.3	172.18.0.4	TCP	58 80 → 63175 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
2429_14.999267783	172.18.0.4	172.18.0.3	TCP	54 63175 → 80 [RST] Seq=1 Win=0 Len=0
2429_14.999526979	172.18.0.4	172.18.0.3	TCP	54 [TCP Port numbers reused] 63176 → 80 [SYN] Seq=0 Win
2429_14.999647578	172.18.0.3	172.18.0.4	TCP	58 80 → 63176 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
2429_14.999685377	172.18.0.4	172.18.0.3	TCP	54 63176 → 80 [RST] Seq=1 Win=0 Len=0
2429_14.999860875	172.18.0.4	172.18.0.3	TCP	54 [TCP Port numbers reused] 63177 → 80 [SYN] Seq=0 Win
2429_14.999952774	172.18.0.3	172.18.0.4	TCP	58 80 → 63177 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MS
2429_14.999992674	172.18.0.4	172.18.0.3	TCP	54 63177 → 80 [RST] Seq=1 Win=0 Len=0

Figure 21. Traffic classified as malicious in Wireshark.

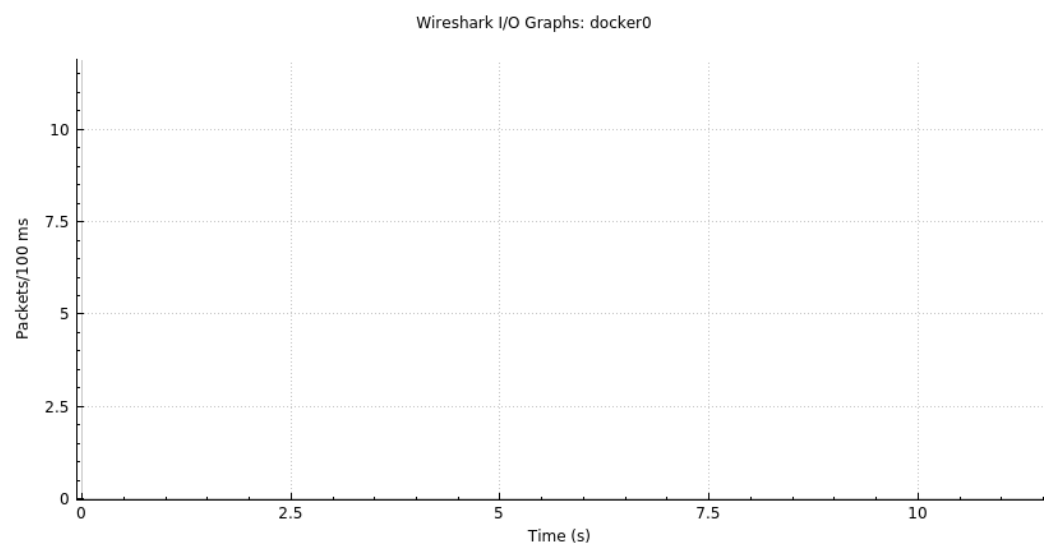


Figure 22. PPS result without the DDoS attack.

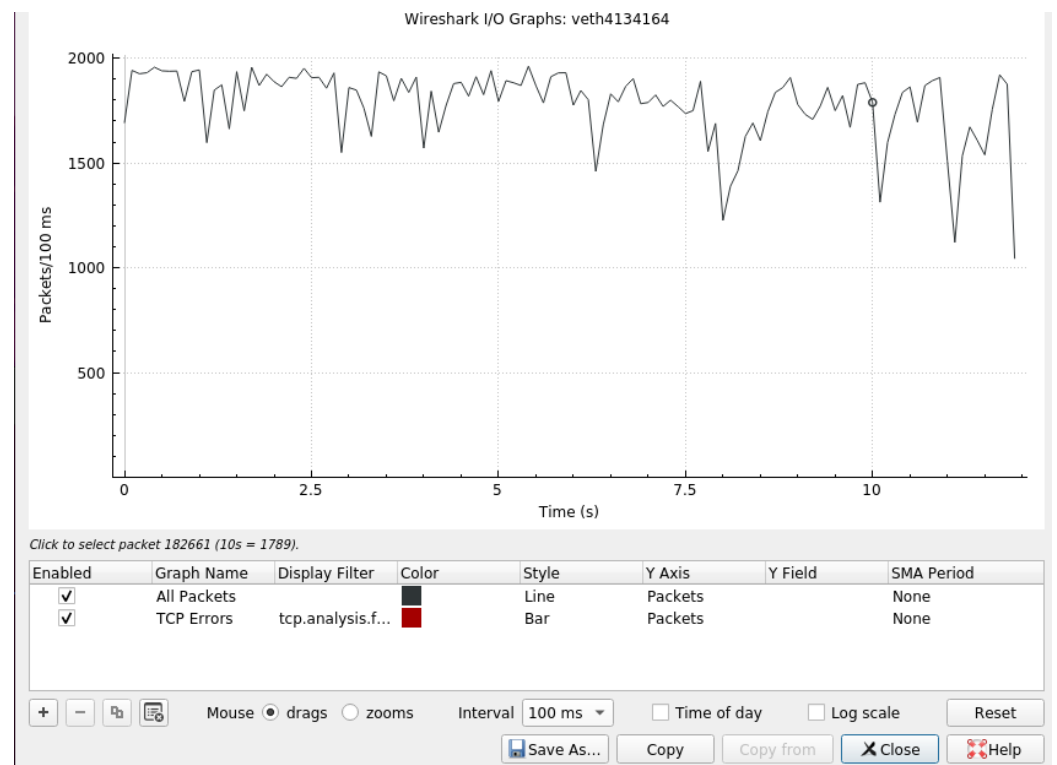
In the case of abnormal traffic confirmed by the execution of the DDoS attack, it is confirmed that the amount of packets per 100 ms is between 1500 and 2000, as shown in Figure 23. This confirms that the availability of the target environment is violated.

In more detail, the cAdvisor tool [33] is used to compare a CPU usage when a DDoS attack does not occur and when it occurs. The CPU usage in a normal state where the DDoS attack does not occur showed mainly usage 0% to 20% as shown in Figure 24.

However, as a result of measuring the CPU usage based on the time of the DDoS attack, the result shown in Figure 25 is confirmed, which means a high usage of 25% to 100%. This result shows that attackers can waste host resources regardless of intention when using Docker, which is a fatal security threat that harms availability.

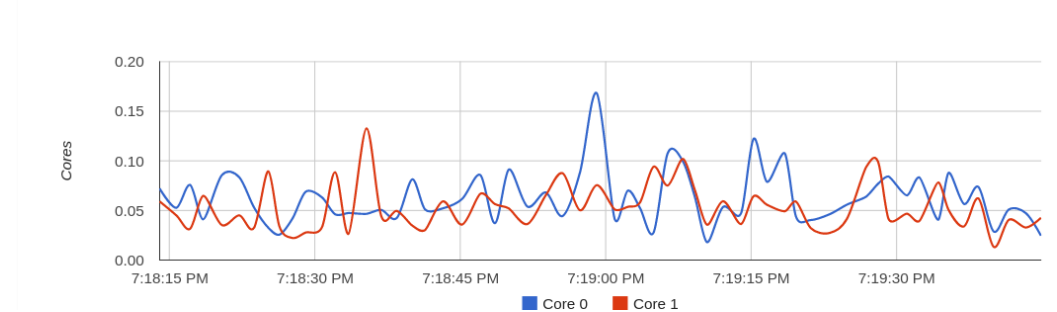
- Privilege escalation

The privilege escalation attack described in Section 4 is performed. First, as mentioned above, it is assumed that the malicious host has uploaded a malicious image with a backdoor code inserted to access the target host to the registry. In addition, the target host is the situation in which the malicious image file uploaded by the attacker is downloaded, and the container is executed based on the malicious image. At this time, the ubuntu\_attacker container of the malicious host is connected to the ubuntu\_target of the target host. Figure 26 shows that the malicious host's ubuntu\_attacker container is connected to the target host's ubuntu\_target container.



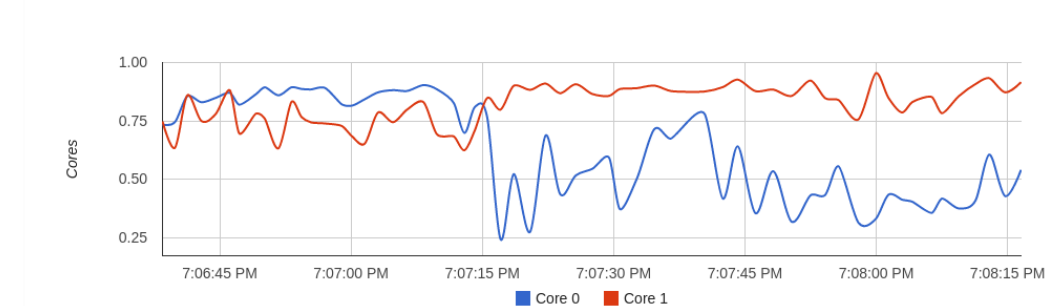
**Figure 23.** Confirmation of the PPS result after the DDoS attack is executed.

#### Usage per Core



**Figure 24.** Confirmation of the CPU consumption when the DDoS attack does not occur.

#### Usage per Core



**Figure 25.** Confirmation of the CPU consumption when the DDoS attack occurs.

When the `ubuntu_target` container is executed on the target host, that the reverse shell can be seen to have been executed as follows. To check this, it can be seen from Figures 27 and 28 that the same result is obtained after inputting the same command into the `ubuntu_attacker` container of malicious host and the `ubuntu_target` container of the target host.

```

root@9cd9903e655e:/# nc -v -k -l 172.17.0.3 8888
Listening on 9cd9903e655e 8888
Connection received on 192.168.19.130 39936
Connecting people

```

**Figure 26.** Privilege escalation attack success screen.

```

$ sudo docker ps -a
CONTAINER ID   IMAGE                                COMMAND                  NAMES
93b2e3317de6   192.168.19.129:5000/ubuntu_backdoor3:latest  "/bin/bash"            ubuntu_target
352193c9fde6   192.168.19.129:5000/ubuntu_backdoor:latest  "/bin/bash"            serene_rubin
0bd728c6372c   ubuntu:latest                          "/bin/bash"            ubuntu_real

```

**Figure 27.** ubuntu\_target docker container list accessed from ubuntu\_attacker (1).

```

neul@ubuntu:~/Desktop$ sudo docker ps -a
CONTAINER ID   IMAGE                                COMMAND                  NAMES
93b2e3317de6   192.168.19.129:5000/ubuntu_backdoor3:latest  "/bin/bash"            ubuntu_target
352193c9fde6   192.168.19.129:5000/ubuntu_backdoor:latest  "/bin/bash"            serene_rubin
0bd728c6372c   ubuntu:latest                          "/bin/bash"            ubuntu_real

```

**Figure 28.** ubuntu\_target docker container list (1).

Figure 29 shows that the container of the ubuntu\_target container is deleted from the ubuntu\_attacker container through privilege escalation.

```

$ docker rm -f 352193c9fde6
352193c9fde6

```

**Figure 29.** Delete the docker container in the ubuntu\_target container.

As a result, it can be confirmed through Figures 30 and 31 that the information of the ubuntu\_target container identified in the ubuntu\_attacker container of the malicious host and the information of the ubuntu\_target identified in the target host are the same.

```

$ sudo docker ps -a
CONTAINER ID   IMAGE                                COMMAND                  NAMES
93b2e3317de6   192.168.19.129:5000/ubuntu_backdoor3:latest  "/bin/bash"            ubuntu_target
0bd728c6372c   ubuntu:latest                          "/bin/bash"            ubuntu_real

```

**Figure 30.** ubuntu\_target docker container list accessed from ubuntu\_attacker (2).

```

neul@ubuntu:~/Desktop$ sudo docker ps -a
CONTAINER ID   IMAGE                                COMMAND                  NAMES
93b2e3317de6   192.168.19.129:5000/ubuntu_backdoor3:latest  "/bin/bash"            ubuntu_target
0bd728c6372c   ubuntu:latest                          "/bin/bash"            ubuntu_real

```

**Figure 31.** ubuntu\_target docker container list (2).

## 6. Conclusions

This paper demonstrated through experiments that security attacks, i.e., ARP spoofing, DDoS, and privilege escalation attacks, can occur in communication between the Docker containers. We have analyzed the impacts of the security attacks in terms of network traffic, CPU consumption, and malicious reverse shell execution. Security attacks occurring in inter-container communication must be prevented, and efforts to demonstrate and analyze attacks based on the security mechanisms presented above are required. For future research, in order to conduct research on security systems usable on inter-container communication, we will demonstrate and analyze various Docker network attacks to which security mechanisms are applied and propose new ways to overcome the limitations of security mechanisms.

**Author Contributions:** H.L. conducted experiments on security attacks between Docker containers and drafted this article. S.K. reviewed the article and provided inputs on experimental analysis. J.-H.L. revised this article and supervised all the work. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No.2021-0-00796, Research on Foundational Technologies for 6G Autonomous Security-by-Design to Guarantee Constant Quality of Security).

**Data Availability Statement:** Not applicable.

**Acknowledgments:** Authors would like to give thanks for the support of Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No.2021-0-00796, Research on Foundational Technologies for 6G Autonomous Security-by-Design to Guarantee Constant Quality of Security).

**Conflicts of Interest:** The authors declare no conflict of interest.

## References

1. Alashhab, Z.R.; Anbar, M.; Singh, M.M.; Leau, Y.-B.; Al-Sai, Z.A.; Alhayjaa, S.A. Impact of coronavirus pandemic crisis on technologies and cloud computing applications. *J. Electron. Sci. Technol.* **2021**, *19*, 100059. [\[CrossRef\]](#)
2. Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Comput.* **2014**, *1*, 81–84. [\[CrossRef\]](#)
3. 2022 Developer Survey. Available online: <https://survey.stackoverflow.co/2022> (accessed on 24 October 2022).
4. I Am Your Defense against Cloud Threats: The Latest Unit 42 Cloud Threat Research. Available online: <https://unit42.paloaltonetworks.com/iam-cloud-threat-research> (accessed on 24 October 2022).
5. Kim, B.S.; Lee, S.H.; Lee, Y.R.; Park, Y.H.; Jeong, J. Design and implementation of cloud docker application architecture based on machine learning in container management for smart manufacturing. *Appl. Sci.* **2022**, *12*, 6737. [\[CrossRef\]](#)
6. Open Source Container Management GUI for Kubernetes, Docker, Swarm. Available online: <https://www.portainer.io/> (accessed on 9 January 2023).
7. Enisa Threat Landscape. 2021. Available online: <https://www.enisa.europa.eu/publications/enisa-threat-landscape-2021> (accessed on 24 October 2022).
8. Kwon, S.; Lee, J.-H. DIVDS: Docker Image Vulnerability Diagnostic System. *IEEE Access* **2020**, *8*, 42666–42673. [\[CrossRef\]](#)
9. Brady, K.; Moon, S.; Nguyen, T.; Coffman, J. Docker Container Security in Cloud Computing. In Proceedings of the 2020 10th Annual Computing and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 6–8 January 2020; pp. 975–980.
10. Zheng, Y.; Dong, W.; Zhao, J. ZeroDVS: Trace-Ability and Security Detection of Container Image Based on Inheritance Graph. In Proceedings of the 2021 IEEE 5th International Conference on Cryptography, Security and Privacy (CSP), Zhuhai, China, 8–10 January 2021; pp. 186–192.
11. Maruszczak, A.; Walkowski, M.; Sujecki, S. Base Systems for Docker Containers - Security Analysis. In Proceedings of the 2022 International Conference on Software, Telecommunications and Computer Networks (SoftCOM), Split, Croatia, 22–24 September 2022; pp. 1–5.
12. Loukidis-Andreou, F.; Giannakopoulos, I.; Doka, K.; Koziris, N. Docker-sec: A fully automated container security enhancement mechanism. In Proceedings of the 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), Vienna, Austria, 2–6 July 2018; pp. 1561–1564.
13. Zhu, H.; Gehrmann, C. Lic-Sec: An enhanced AppArmor Docker security profile generator. *J. Inf. Secur. Appl.* **2021**, *61*, 102924. [\[CrossRef\]](#)
14. Mattetti, M.; Shulman-Peleg, A.; Allouche, Y.; Corradi, A.; Dolev, S.; Foschini, L. Securing the infrastructure and the workloads of linux containers. In Proceedings of the 2015 IEEE Conference on Communications and Network Security (CNS), Florence, Italy, 28–30 September 2015; pp. 559–567.
15. Container Networking Model. Available online: <http://clouddrain21.com/container-networking-model> (accessed on 3 November 2022).
16. Hao, Z.; Wang, B.; Deng, W.; Zhang, W. Measurement and evaluation for docker container networking. In Proceedings of the 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), Nanjing, China, 12–14 October 2017; pp. 105–108.
17. Abbasi, U.; Bourhim, E.H.; Dieye, M.; Elbiaze, H. A performance comparison of container networking alternatives. *IEEE Netw.* **2019**, *33*, 178–185. [\[CrossRef\]](#)
18. Suo, K.; Zhao, Y.; Chen, W.; Rao, J. An analysis and empirical study of container networks. In Proceedings of the IEEE INFOCOM 2018-IEEE Conference on Computer Communications, Honolulu, HI, USA, 16–19 April 2018; pp. 189–197.
19. Docker Container—Bridge Driver Network Mode. Available online: <https://www.bogotobogo.com/DevOps/Docker/Docker-Bridge-Driver-Networks.php> (accessed on 5 November 2022).
20. Network Containers. Available online: <https://docs.docker.com/engine/tutorials/networkingcontainers/> (accessed on 5 November 2022).

21. Use Overlay Networks. Available online: <https://docs.docker.com/network/overlay> (accessed on 7 November 2022).
22. Understanding Weave Net. Available online: <https://www.weave.works/docs/net/latest/concepts/how-it-works> (accessed on 8 November 2022).
23. Calico. Available online: <https://www.tigera.io/project-calico> (accessed on 10 November 2022).
24. Nam, J.; Lee, S.; Seo, H.; Porras, P.; Yegneswaran, V.; Shin, S. BASTION: A security enforcement network stack for container networks. In Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC 20), Online, 15–17 July 2020; pp. 81–95.
25. Benchmarking Nftables. Available online: <https://developers.redhat.com/blog/2017/04/11/benchmarking-nftables> (accessed on 11 January 2023).
26. Lee, H.; Woo, S.; Lee, J.-H. Analysis of possible security threats in communication between Docker Containers. In Proceedings of the Korean Institute of Communication Sciences Conference, Yeosu, Korea, 17–19 November 2021; pp. 999–1000.
27. Yang, Y.; Shen, W.; Ruan, B.; Liu, W.; Ren, K. Security Challenges in the Container Cloud. In Proceedings of the 2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA), Atlanta, GA, USA, 13–15 December 2021; pp. 137–145.
28. Bui, T. Analysis of docker security. *arXiv* **2015**, arXiv:1501.02967.
29. Sultan, S.; Ahmad, I.; Dimitriou, T. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access* **2019**, *7*, 52976–52996. [[CrossRef](#)]
30. Combe, T.; Martin, A.; Pietro, R.D. To Docker or Not to Docker: A Security Perspective. *IEEE Cloud Comput.* **2016**, *3*, 54–62. [[CrossRef](#)]
31. Martin, A.; Raponi, S.; Combe, T.; Pietro, R.D. Docker ecosystem-vulnerability analysis. *Comput. Commun.* **2018**, *122*, 30–43. [[CrossRef](#)]
32. Aqua Security Software Inc. *Attacks in the Wild on the Container Supply Chain and Infrastructure (Aqua)*; Cloud Native Threat Report; Aqua Security Software Inc.: Burlington, MA, USA, 2021; pp. 3–5.
33. cAdvisor. Available online: <https://github.com/google/cadvisor> (accessed on 15 November 2022).

**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.