Implementing Recent Attacks And A Security Application On Container

Parth Vakil[1]
Student ID: 40233132

Malvik Chauhan[1]
Student ID: 40268733

Fahima Noor[1]
Student ID: 40268465

Adit Vasava[1]
Student ID: 40259101

Mohammad Tawsif Chowdhury[1]
Student ID: 40261062

Dennis Tank[1]
Student ID: 40261560

Peter Vourantonis[1]
Student ID: 40157751

Prashant Parmar[1]
Student ID: 40291246

[1]CIISE, Concordia University, Montréal, Canada

Abstract

This report elucidates the methodologies of recent security-related issues with Container focusing on the Docker platform. Additionally, it provides the defense mechanisms that can be applied to patch the corresponding security hole. It highlights three attacks - vulnerabilities in Runc, Exposed Docker Socket, and SYS_MODULE Capability - that can be exploited on Docker and three strategies - Runc Fixes, Seccomp Implementation, and AuthZ Broker - to resolve them. The exploits are explicated in detail and with steps to recreate the attacks and, further, the remediation of every attack is presented as a precautionary measure. Moreover, the defense strategies are explained with code, examples, and implementation details.

***Keywords*— Docker, Container, Exploit, Defense, Seccomp, Leaky vessels, Docker-Socket, SysModule, Sysmodule**

Introduction

Docker is a platform that uses containers to simplify the construction, deployment, and operation of applications [1] [2]. With containers, a software developer may encapsulate a program and all of its dependencies into a single unit. This guarantees that the program will function properly in various settings. It ensures that the application process works uniformly across all infrastructures and facilitates the sharing of the process. With Docker, developers can easily scale their apps and optimize their workflows. Because virtual machines (VMs) require a different operating system for every instance, they are huge and require a lot of resources. Because containers utilize the Linux kernel of the host computer, they may use the same kernel and do not require different Oss. Between the host OS and containers, Docker serves as a mediator. This enables lower size and effective resource usage. Because of this, containers are easier to scale and are lighter than virtual machines. When running on top of a host operating system (e.g., Linux), Docker may build containers with several operating systems, such as Ubuntu, Debian, etc. An image in Docker is a self-contained software package containing all the components required to run an application. Code, runtime, system tools, libraries, and settings are all included. Developers provide the software, configurations, and settings to be installed while creating an image. whereas an instance of a picture is a container. Code and its dependencies are packaged into a standardized software unit that allows programs to execute reliably in a variety of contexts. Images are the basis for containers, and a single picture can launch numerous containers, each with its own ID.

1 Attacks

1.1 Leaky Vessels: CVE-2024-21626

1.1.1 Overview

"Leaky Vessels" collectively refers to container escape vulnerabilities found within the container runtime environment [3] . These vulnerabilities pose a significant threat to the isolation that containers inherently provide from their host operating systems [4]. RunC version 1.1.11 and prior to this are affected.

RunC is a lightweight, low-level portable container runtime [5] [6] that includes all of the plumbing code used by Docker to interact with container-related system features. The goal of runC is to make standard containers available everywhere. Full Linux namespaces support, and native support of Linux security features such as SELinux, and AppArmor are primary features of RunC [5].

Due to leakage of various file descriptors within RunC init, containers can access the host file system if the container is configured to have process 'cwd' set to /proc/self/fd/7,8. This allows for container escapes, enabling an attacker who has gained access to a container to execute arbitrary code on the host machine, thereby compromising the entire system. The attack happens because the vulnerable RunC version does not verify the final working directory after doing 'chdir' [7]. The vulnerability can be exploited in various ways.

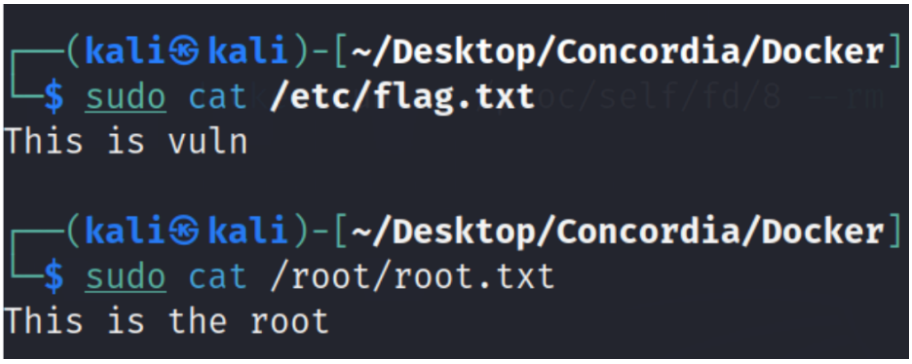
1.1.2 Working of the Exploit

- The vulnerability is also called Leaky vessels. Runc version 1.1.11 and before this are impacted using this version.
- Due to the leakage of various file descriptors, the container can access the host file system
- File descriptors are leaked internally within RunC init. If the container is configured to have a process 'cwd' set to /proc/self/fd/7,8, the process will have a working directory in the host mount namespace.
- This happens because the vulnerable runc version doesn't verify the final working directory after doing chdir.

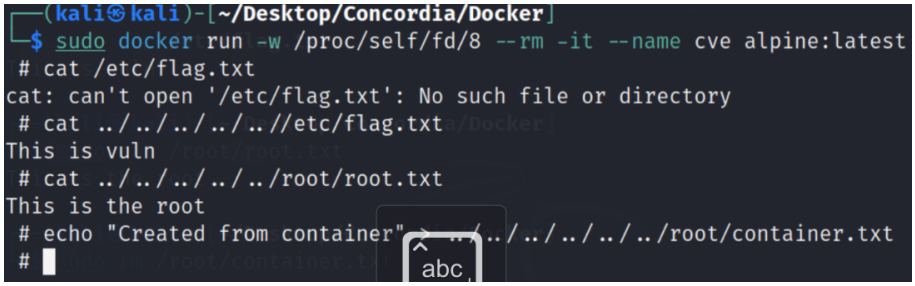
This attack can be executed in different ways i.e. Using Workdir and Using symlink from the container.

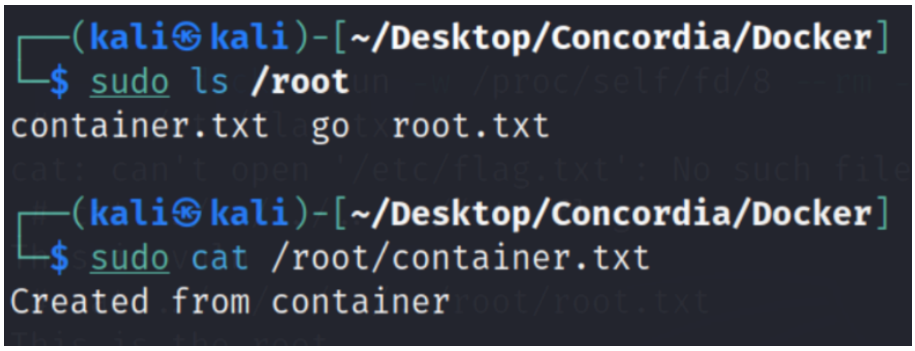
1.1.3 Exploit -1: Using Workdir:

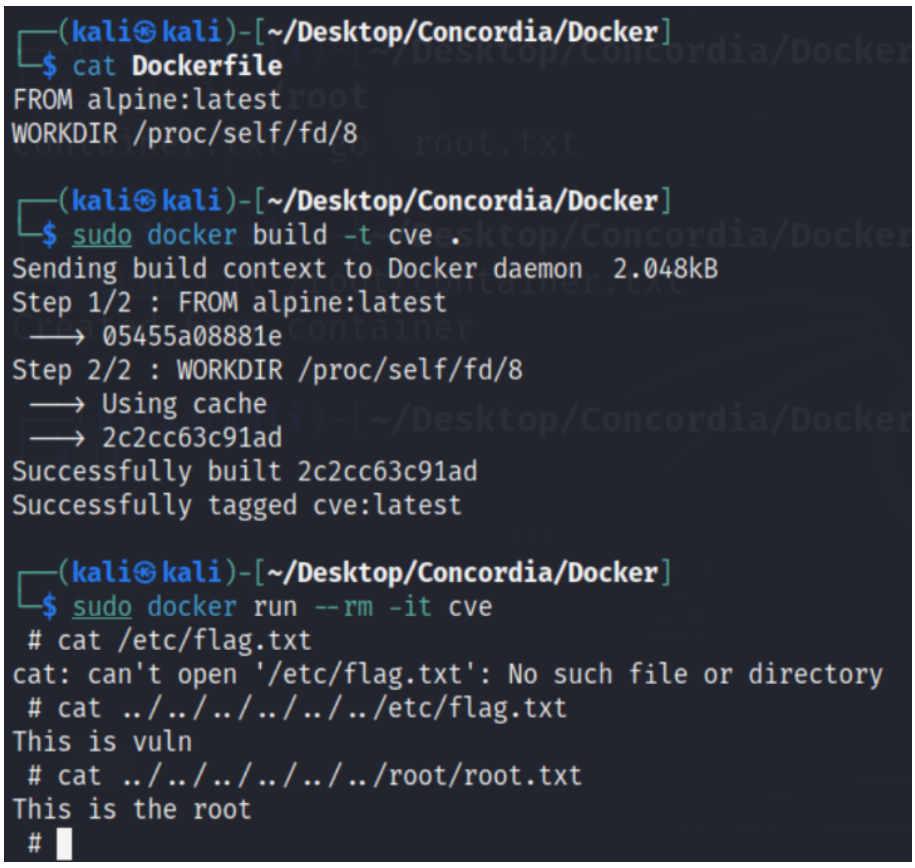
- We will start by adding two flags, one in /etc and one in /root directory to verify the exploit.



- We can run a docker container by specifying a working directory using -w flag as "/proc/self/fd/8".
- We can simply access the file system by moving back into the directory using "../../../../../"
- We can confirm that we were able to create a new file in the root directory
- We can perform the Same attack by specifying the working directory in the Dockerfile:

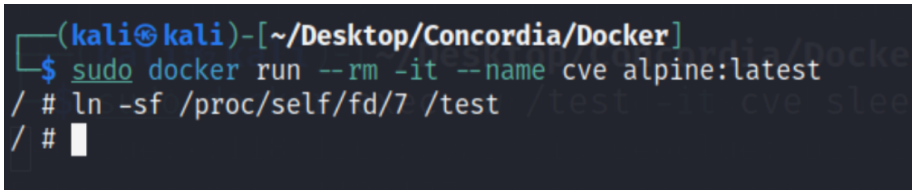




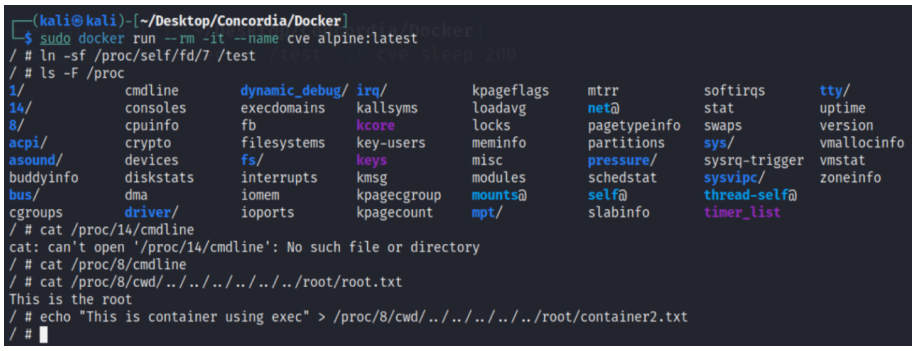


1.1.4 Exploit-2: Using symlink from the container:

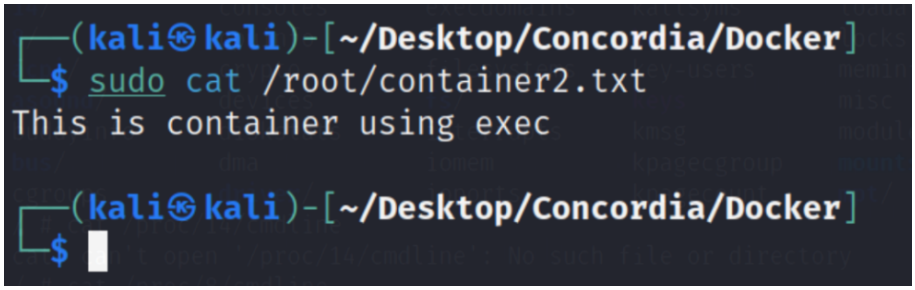
- If we do not have control over the images and docker run, but if we know that some administrative process is going to call the docker exec command using the – CWD flag, and if we know the exact file directory specified in - CWD, then we can create a symlink to exploit it.
- We will run a normal container, and we will create a symlink to /proc/self/fd/8.
- Then we will use the command docker exec by specifying /test as the working directory.



- In the container, we will try to find the process ID, and then we can try to access the host file system using cwd. We will try to write to the root directory.



- We can confirm that the new file is created in the root file directory. We can exploit this further to get the root user.



1.2 Exploiting Exposed Docker Socket

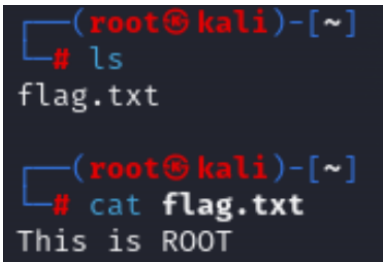
1.2.1 Overview

By default, Docker runs through a non-networked UNIX socket, /var/run/docker.sock file. It can also optionally communicate using SSH or a TLS (HTTPS) socket [8]. It is a communication endpoint that allows users to interact with the Docker daemon through the Docker API. Because it is a file, admins can share and run docker.sock within a container and then use it to communicate with that container. A container that runs docker.sock can start or stop other containers, create images on the host, or write to the host file system [9] [10].

1.2.2 Working of the Exploit

In this exploit, we will get access to the root directory from an unprivileged user namespace. This exploit works when the Docker Socket is exposed over TCP for a remote Docker API execution. If the intel for the TCP

connection is available we can use the Docker -H flag to access that exposed Docker Socket. For this attack, we have created a flag.txt file in the root directory as a root user that we will try to access as an unprivileged user.

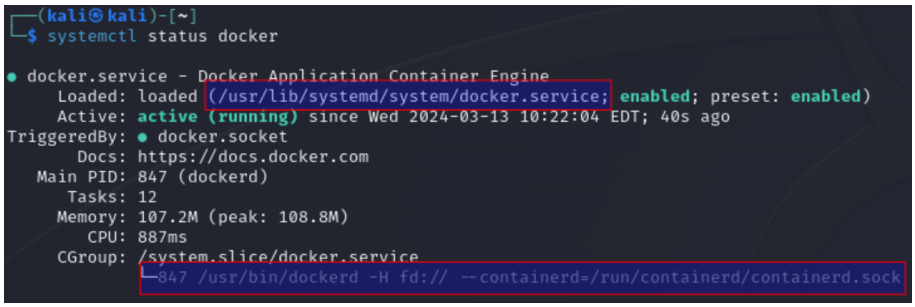


The prerequisites for the exploit:

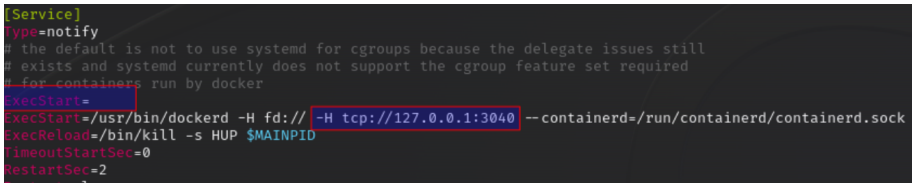
- The Docker Socket '/var/run/docker.sock' should be Exploited over TCP at any IP and PORT.
- Docker's user-namespace mapping should have defaulted to limit the unprivileged user.

1.2.3 Recreating the Attack

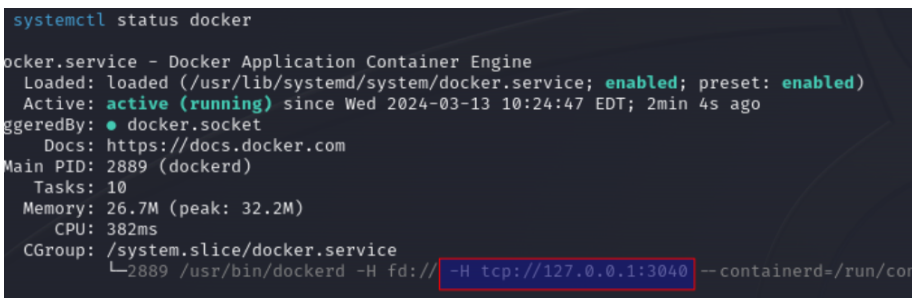
- Exposing the Docker Socket over TCP.
  - Find the docker.service file location and edit the file to expose the socket to any IP and PORT.



  - Add "ExecStart=" before the true "ExecStart" then edit the true "ExecStart" with "-H tcp//¡IP¿:¡PORT¿" and save the file.



  - Restart the Docker daemon and Docker service.
  - Now the Docker Socket is exposed over TCP.



```
┌──(kali㉿kali)-[~]
└─$ docker -H tcp://127.0.0.1:3040 info
Client:
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.12.1
    Path:     /usr/libexec/docker/cli-plugins/docker-buildx
```

- Limit unprivileged users in the Docker Container, using the '–users-remap=default' flag.
- On an unprivileged user namespace try to excess the Docker API over TCP with the defined IP and Port.
- Run a Docker image with the following flags: Here the mounted folder is "shared" and we can access the root directory through it.
  - -it: It opens an interactive container instance in docker.
  - -v /:/shared:ro: It mounts the folders that can be accessible inside the container.
  - -t: It allocates a pseudo-TTY, to run a throwaway interactive shell.
  - sh: To get into the shell of the container.

```
┌──(kali㉿kali)-[~]
└─$ docker -H tcp://127.0.0.1:3040 run -it -v /:/shared:ro -t alpine sh
/ # ls shared/root
flag.txt
/ # cat flag.txt
cat: can't open 'flag.txt': No such file or directory
/ # ls
bin     etc     lib     mnt     proc    run     shared  sys     usr
dev     home    media   opt     root    sbin    srv     tmp     var
/ # cd shared/root
/shared/root # cat flag.txt
This is ROOT
/shared/root # exit
```

- From here the root directory is accessible even when the user doesn't have privileges to do so.

1.2.4 Remediations

- The root should never expose the 'docker.socket' over TCP [11].
- If the Socket is exposed over the TCP it is better to use a TLS certificate flag to encrypt the API messages.
- If the Socket is exposed over TCP and TLS it is better to use an Authentication plugin (for example AuthZ).

To access the Docker daemon remotely, a TCP socket is needed to be enabled [12]. When using a TCP socket, the Docker daemon provides unencrypted and unauthenticated direct access to the Docker daemon by default. The daemon should be secured either using the built-in HTTPS encrypted socket or by putting a secure web proxy in front of it. If Docker needs to be reachable through HTTP, it should be done so by enabling TLS (HTTPS) by specifying the verify flag and pointing Docker's "tlscacert" flag to a trusted CA certificate [13].

1.3 Exploiting the SYS_MODULE Capability

1.3.1 Overview

SYS_MODULE capability allows a container to insert/remove kernel modules (init_module(2), finit_module(2), and delete_module(2) system call) in/from the kernel of the host machine. If enabled, the kernel can be modified at will, subverting all system security, Linux Security Modules, and container systems [14] [15].

1.3.2 Working of the Exploit

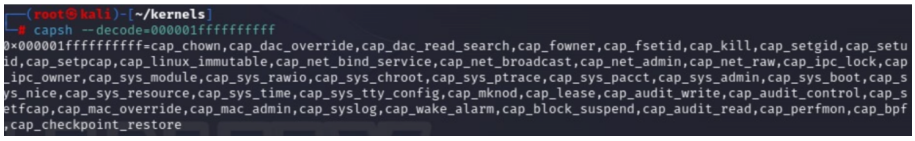
In this exploit, we gain access to the host system by exploiting the capability to load/unload kernel modules. This exploit works mainly on the ability of the container to call load/unload system calls from the container, which loads/unloads the module on the host itself and hence escapes from the container. The exploit makes use of call_usermodehelper(), which is used to set up a user application and run it from the kernel.

The prerequisites for the exploit:

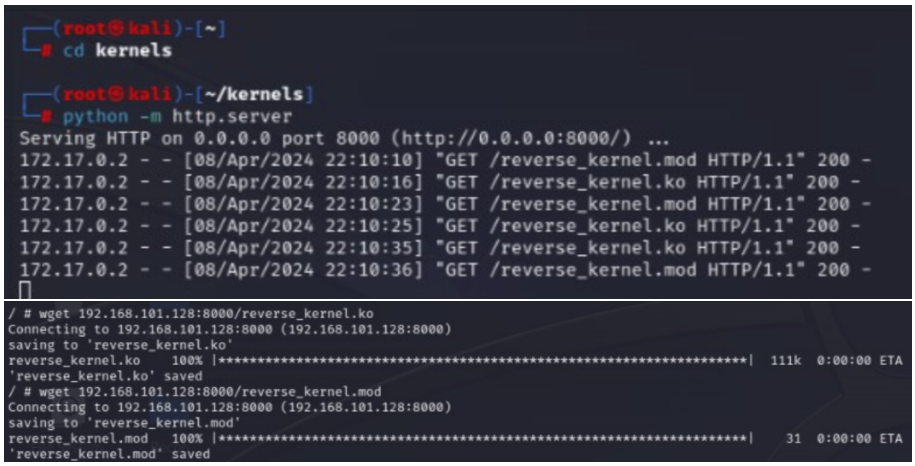
- The container should have the SYS_MODULE capability.

1.3.3 Recreating the Attack

- Let's identify the capabilities of the container. Here, we can confirm that capability cap_sys_module is enabled for the container. The container can add/remove kernels from the Docker host system.

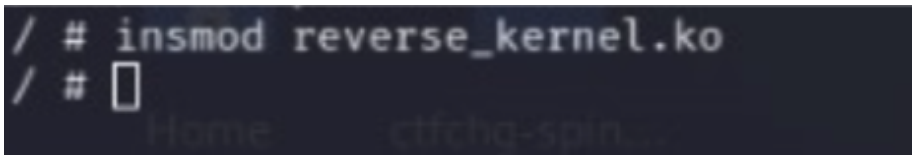


- Create a kernel module code to give a reverse shell.
- Compile the kernel module and make it available in the Docker system.
- To load the kernel module into the container, use the Python HTTP server on the host machine and curl/wget file into the container.

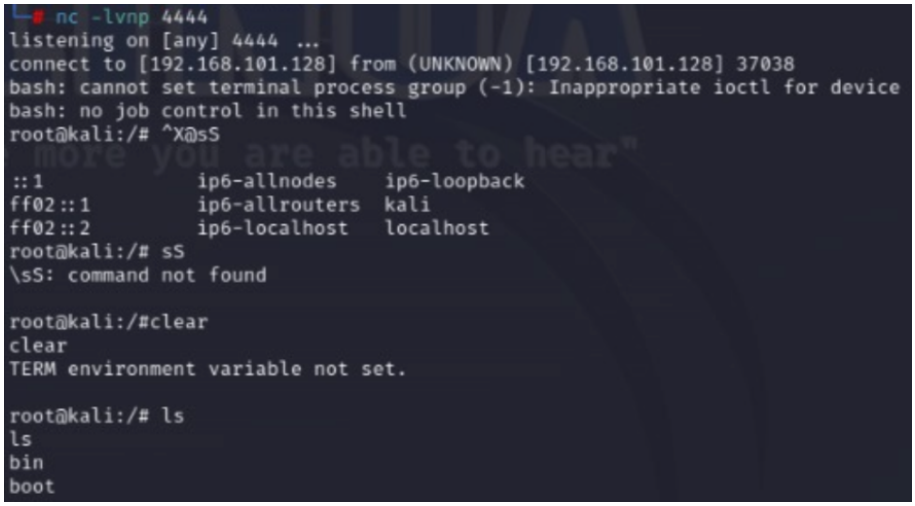


- Now, start the Netcat listener on the host computer and use port 4444.

- The next step is to use the command insmod to load the kernel.



- Once the system loads the kernel, a reverse connection to the container's host system is dropped.



1.3.4 Remediations

Minimize kernel Capabilities assigned to the docker engine. When launching the Docker daemon, use the '–cap-add' and '–cap-drop' options to grant only the necessary capabilities required. For eg., SYS_MODULE capability might not be needed by the container to function.

2 Defenses

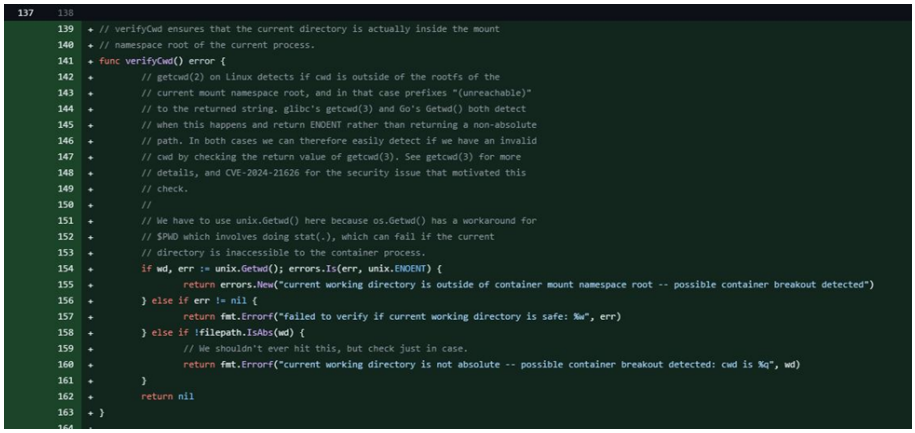
2.1 Leaky Vessels Fixes

2.1.1 Overview

In runc 1.1.11 and earlier, an internal file descriptor caused an information leak. This leak made it possible for an attacker to spawn a new container process that would have a working directory in the host's filesystem namespace essentially resulting in a container escape. We can try to use namespace remapping to mitigate the attack. However, this does not solve the issue because an attacker can still access the host's file system, but it would not have root access. Therefore, the only true solution is to upgrade from a vulnerable version of runc to the patched version of 1.1.12 as the underlying issues with the runc code have been fixed.

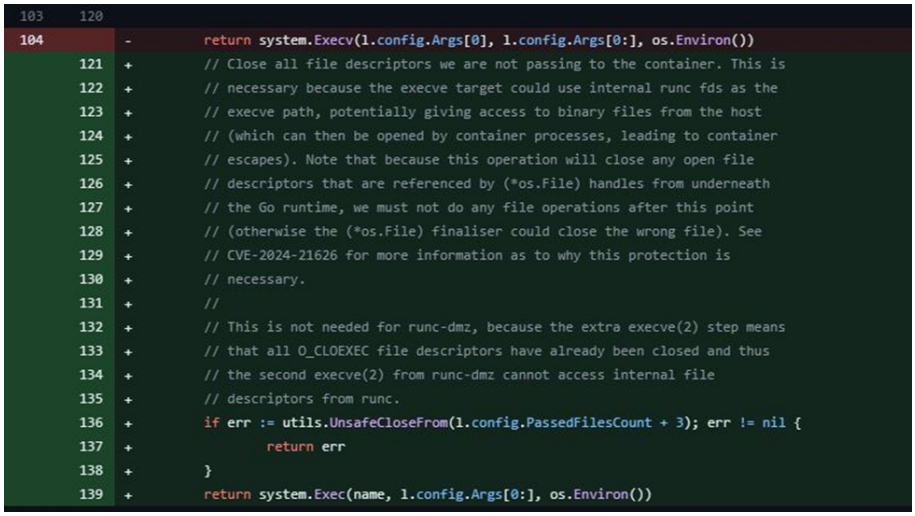
2.1.2 Detailed Explanation of Runc fixes

- Fix 1: runc now checks to see if the working directory being accessed belongs to a container. It does this by calling os.Getwd and seeing if it equals to ENOENT which is an error stating that the given path does not exist. This prevents the attacker from accessing files outside the bounds of the container, but it does not fix the file descriptor leak.



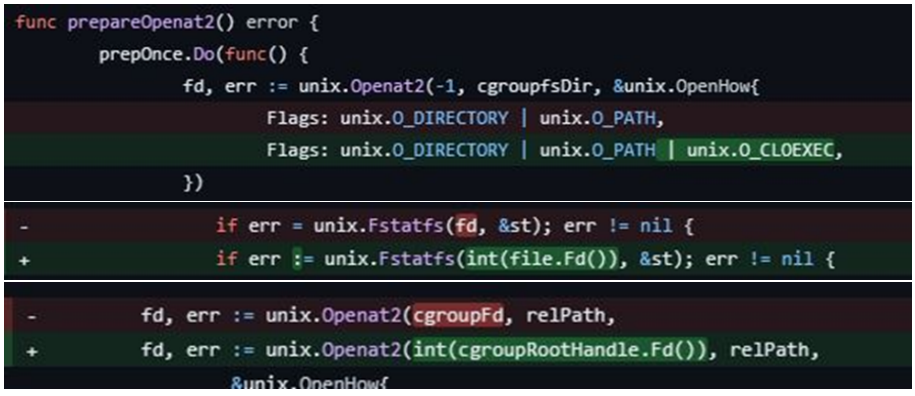
```
+ // verifyCwd ensures that the current directory is actually inside the mount
+ // namespace root of the current process.
+ func verifyCwd() error {
+       // getcwd(2) on Linux detects if cwd is outside of the rootfs of the
+       // current mount namespace root, and in that case prefixes "(unreachable)"
+       // to the returned string. glibc's getcwd(3) and Go's Getwd() both detect
+       // when this happens and return ENOENT rather than returning a non-absolute
+       // path. In both cases we can therefore easily detect if we have an invalid
+       // cwd by checking the return value of getcwd(3). See getcwd(3) for more
+       // details, and CVE-2024-21626 for the security issue that motivated this
+       // check.
+       //
+       // We have to use unix.Getwd() here because os.Getwd() has a workaround for
+       // $PWD which involves doing stat(.), which can fail if the current
+       // directory is inaccessible to the container process.
+       if wd, err := unix.Getwd(); errors.Is(err, unix.ENOENT) {
+             return errors.New("current working directory is outside of container mount namespace root -- possible container breakout detected")
+       } else if err != nil {
+             return fmt.Errorf("failed to verify if current working directory is safe: %w", err)
+       } else if !filepath.IsAbs(wd) {
+             // We shouldn't ever hit this, but check just in case.
+             return fmt.Errorf("current working directory is not absolute -- possible container breakout detected: cwd is %q", wd)
+       }
+       return nil
+ }
```

- Fix 2: Now all internal file descriptors are closed during the final stages of runc init just before 'execve' starts. Runc init is what starts the container environment and 'execve' executes the program based on a path that it receives. This fix prevents the file descriptors from being passed to 'execve' to escape the container.



```
-       return system.Execv(l.config.Args[0], l.config.Args[0:], os.Environ())
+       // Close all file descriptors we are not passing to the container. This is
+       // necessary because the execve target could use internal runc fds as the
+       // execve path, potentially giving access to binary files from the host
+       // (which can then be opened by container processes, leading to container
+       // escapes). Note that because this operation will close any open file
+       // descriptors that are referenced by (*os.File) handles from underneath
+       // the Go runtime, we must not do any file operations after this point
+       // (otherwise the (*os.File) finaliser could close the wrong file). See
+       // CVE-2024-21626 for more information as to why this protection is
+       // necessary.
+       //
+       // This is not needed for runc-dmz, because the extra execve(2) step means
+       // that all O_CLOEXEC file descriptors have already been closed and thus
+       // the second execve(2) from runc-dmz cannot access internal file
+       // descriptors from runc.
+       if err := utils.UnsafeCloseFrom(l.config.PassedFilesCount + 3); err != nil {
+             return err
+       }
+       return system.Exec(name, l.config.Args[0:], os.Environ())
```

- Fix 3: This fix tackled the descriptor leak by using a handler instead of accessing directly. O_CLOEXEC is used as a flag which makes the file descriptor automatically close whenever exec functions operate.



```
func prepareOpenat2() error {
      prepOnce.Do(func() {
            fd, err := unix.Openat2(-1, cgroupfsDir, &unix.OpenHow{
                  Flags: unix.O_DIRECTORY | unix.O_PATH,
                  Flags: unix.O_DIRECTORY | unix.O_PATH | unix.O_CLOEXEC,
            })
-             if err = unix.Fstatfs(fd, &st); err != nil {
+             if err := unix.Fstatfs(int(file.Fd()), &st); err != nil {
-       fd, err := unix.Openat2(cgroupFd, relPath,
+       fd, err := unix.Openat2(int(cgroupRootHandle.Fd()), relPath,
                  &unix.OpenHow{
```

- Fix 4: This fix was implemented to prevent a future leak by using O_CLOEXEC for non-standard input and output files before using RunC init.

libcontainer/container_linux.go

```
@@ -353,6 +353,15 @@ func (c *linuxContainer) start(process *Process) (retErr error) {
                }()
        }

+       // Before starting "runc init", mark all non-stdio open files as O_CLOEXEC
+       // to make sure we don't leak any files into "runc init". Any files to be
+       // passed to "runc init" through ExtraFiles will get dup2'd by the Go
+       // runtime and thus their O_CLOEXEC flag will be cleared. This is some
+       // additional protection against attacks like CVE-2024-21626, by making
+       // sure we never leak files to "runc init" we didn't intend to.
+       if err := utils.CloseExecFrom(3); err != nil {
+               return fmt.Errorf("unable to mark non-stdio fds as cloexec: %w", err)
+       }
        if err := parent.start(); err != nil {
                return fmt.Errorf("unable to start container process: %w", err)
        }
```

```python
import subprocess

def check():
    vulnerable = "1.1.11" #Vulnerable version of runc
    try:
        v = subprocess.run(['runc', '--version'], capture_output=True, text=True) #Executes the command to check runc version
        runcV = v.stdout.split('\n')
        version = runcV[0].split()
        version = version[2].strip()

        print("Current runc version is:", version) #Display runc version installed

        #If the version is 1.1.11 or earlier it is vulnerable and will be updated to the latest version
        if version <= vulnerable:
            print("\nVulnerable runc version detected!")
            print("Updating runc to a patched version...")

            # Finds every instance of runc and updates to latest version
            subprocess.run(['whereis', 'runc', '|', 'sudo', 'xargs', 'rm', 'rf'])
            subprocess.run(['sudo', 'apt', 'install', '--reinstall', 'runc'])

            v = subprocess.run(['runc', '--version'], capture_output=True, text=True) #Executes the command to check runc version
            runcV = v.stdout.split('\n')
            version = runcV[0].split()
            version = version[2].strip()

            print("\nrunc updated to version:", version) #Show new runc version
        else:
            print("runc version is not vulnerable to leaky vessels")


    except FileNotFoundError:
        return "runc is not installed"

if __name__ == "__main__":
    print("Checking runc version...")
```

2.1.3 Remediation 1 - Namespaces:

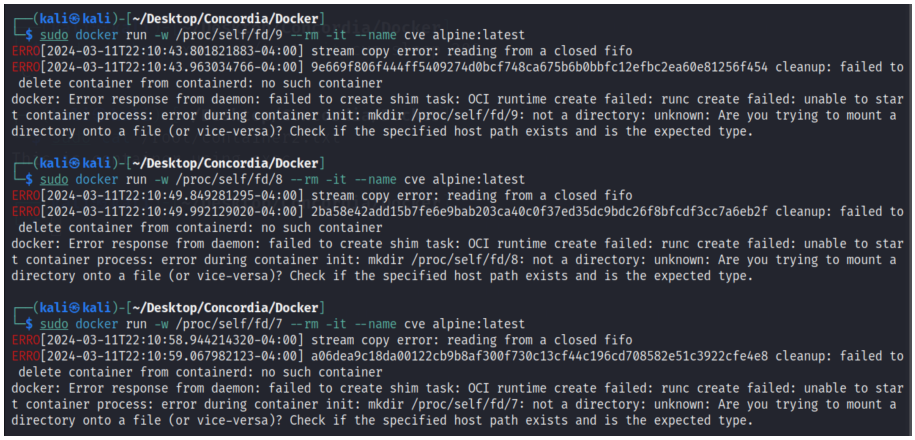
Our First approach was to implement user namespace.

- We stopped and started the docker deamon by implementing the default user namespace remap. (Fig**??**)

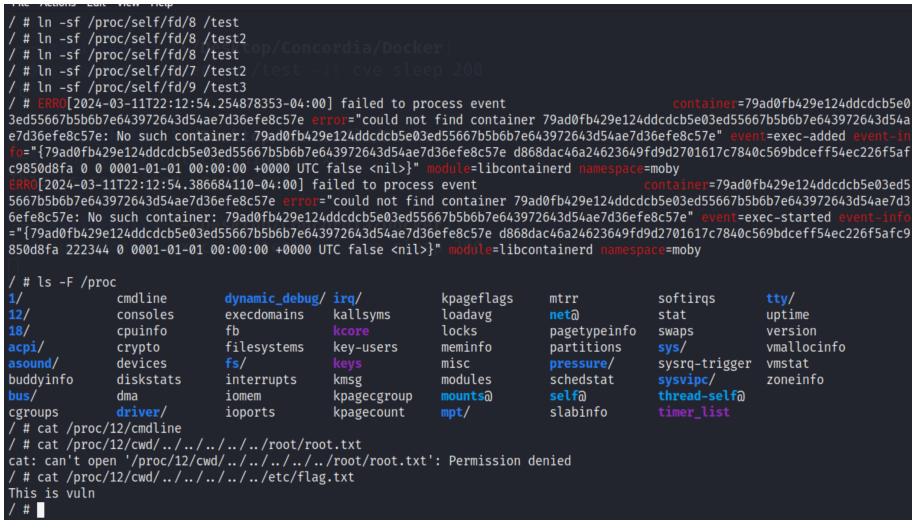
```
┌──(kali㉿kali)-[~/Desktop/Concordia/Docker]
└─$ sudo systemctl stop docker

┌──(kali㉿kali)-[~/Desktop/Concordia/Docker]
└─$ sudo rm /var/run/docker.pid

┌──(kali㉿kali)-[~/Desktop/Concordia/Docker]
└─$ sudo dockerd --userns-remap=default &
[2] 220172

┌──(kali㉿kali)-[~/Desktop/Concordia/Docker]
└─$ INFO[2024-03-11T22:09:57.186153826-04:00] Starting up
INFO[2024-03-11T22:09:57.186593380-04:00] User namespaces: ID ranges will be mapped to subuid/subgid ranges of:
INFO[2024-03-11T22:09:57.189816593-04:00] User namespaces: ID ranges will be mapped to subuid/subgid ranges of:
INFO[2024-03-11T22:09:57.192192692-04:00] [core] parsed scheme: "unix"                    module=grpc
INFO[2024-03-11T22:09:57.192282987-04:00] [core] scheme "unix" not registered, fallback to default scheme  module
INFO[2024-03-11T22:09:57.192428783-04:00] [core] ccResolverWrapper: sending update to cc: {[{unix:///run/containe
d.sock  <nil> 0 <nil>}] <nil> <nil>}  module=grpc
INFO[2024-03-11T22:09:57.192474016-04:00] [core] ClientConn switching balancer to "pick_first"  module=grpc
INFO[2024-03-11T22:09:57.192496889-04:00] [core] Channel switches to new LB policy "pick_first"  module=grpc
INFO[2024-03-11T22:09:57.192662228-04:00] [core] Subchannel Connectivity change to CONNECTING  module=grpc
INFO[2024-03-11T22:09:57.192732608-04:00] [core] Subchannel picks a new address "unix:///run/containerd/containe
onnect  module=grpc
INFO[2024-03-11T22:09:57.192916575-04:00] [core] Channel Connectivity change to CONNECTING  module=grpc
INFO[2024-03-11T22:09:57.193858707-04:00] [core] Subchannel Connectivity change to READY  module=grpc
INFO[2024-03-11T22:09:57.194013038-04:00] [core] Channel Connectivity change to READY   module=grpc
INFO[2024-03-11T22:09:57.195654527-04:00] [core] parsed scheme: "unix"                    module=grpc
INFO[2024-03-11T22:09:57.195780769-04:00] [core] scheme "unix" not registered, fallback to default scheme  module
INFO[2024-03-11T22:09:57.195851772-04:00] [core] ccResolverWrapper: sending update to cc: {[{unix:///run/containe
```

- Then we tried to perform the first attack by trying file descriptors 7,8 and 9; In most cases, these are the

vulnerable file descriptors.(Fig**??**



- From the results, we can see that we are not able to perform the First Exploit.
- Then we tried to perform the second attack.(Fig**??**)



- From the results, we can conclude that we were able to perform the second Exploit and reach the host filesystem, but this time docker would block us from accessing the root filesystem.

2.1.4 Remediation 2 - Python script to upgrade Runc version

Since updating to runc 1.1.12 is very critical, we have developed a Python script that detects what version of runc is currently installed on the system. If the version of runc is a vulnerable one, the script will remove every instance of it on the system and install the latest version.

2.2 Seccomp Implementation

2.2.1 Overview

- Seccomp (secure computing mode) is a security feature of Linux that restricts programs by defining what they can do and what they cannot.
- Only authorized users can perform certain actions, like executing commands and accessing host resources.
- It allows us to define what system calls an application can make. For example, a program is allowed to read files but cannot make a system call to establish a new network connection in an attempt to reverse a shell attack.

- Such Seccomp profiles are useful to protect against attackers.
- Seccomp profiles can be implemented on two levels:
  - Global level: implemented for all the processes in the system.
  - Process level: this is applied to individual processes or containers, providing granular control.
- System call filtering: it allows the developer to precisely apply rules in the application that which process can execute a system call. It helps avoid dangerous system calls by adversaries by limiting the capabilities.
- Actions: Seccomp has different actions for a system call. Few for example: 'SCMP_ACT_ALLOW' to allow the syscall, 'SCMP_ACT_ERRNO' to deny the syscall, 'SCMP_ACT_LOG' to log all the events for a syscall. These actions can be chosen per the application's security and requirements.

2.2.2 Seccomp profile

- Seccomp profile can be depicted as below:

```
1 {
2   "defaultAction": "SCMP_ACT_ALLOW",
3   "architectures": [
4     "SCMP_ARCH_X86_64"
5   ],
6   "syscalls": [
7     { "names": [ "read", "write"],
8       "action": "SCMP_ACT_ALLOW"
9     },
10    { "names": [ "chmod"],
11      "action": "SCMP_ACT_ERRNO"
12    }
13  ]
14 }
```

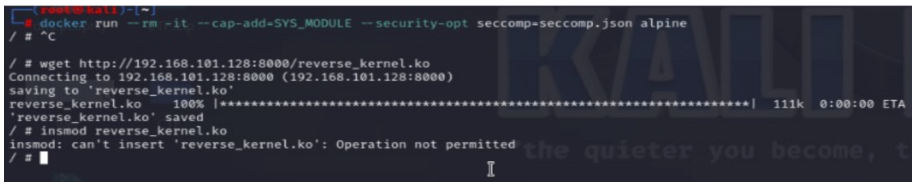
- Here, defaultAction has been set as allow, meaning all the syscalls are allowed. By creating a blacklist for "chmod" with 'SCMP_ACT_ERRNO' defines to produce an error when executed.
- Seccomp profile can be invoked by the following.

```
┌──(deathnote㉿ kali)-[~/Desktop/Docker]
└─$ sudo docker run -it --security-opt seccomp=Seccomp.json alpine
```

2.2.3 Steps to defend against the abuse of capability.

- To not let cap_sys_module be abused to get a reverse shell on the system by loading the kernel module, we can start by developing a Seccomp profile. Here, we will be restricting the following system calls: init_module: to load a kernel module into the running Linux kernel; finit_module: Load a kernel module from a file descriptor., and delete_module: Remove a loaded kernel module.
- Run the container with the Seccomp profile activated. now when we try to load the kernel module using insmod, it gives an error that operation is not permitted.

2.3 AuthZ Broker



2.3.1 Overview

Docker's default authorization model operates on an all-or-nothing basis, lacking granularity in access control. Any user granted access to the Docker daemon possesses the ability to execute any Docker client command [16]. Therefore, to gain greater access control, authorization plugins can be created and added to the Docker daemon configuration. By employing an authorization plugin, Docker administrators can meticulously configure access policies, allowing for precise management of access to the Docker daemon.

The Twistlock AuthZ Broker is a Docker authorization plugin that enforces user access control for containerized applications [17]. It operates as a mediator between the Docker daemon, the core engine that manages containers, and the Docker remote API, the interface for interacting with the daemon [18]. This plugin architecture offers several key advantages:

- **Centralized Policy Management**: Instead of configuring access control directly within the Docker daemon, administrators define authorization policies in a centralized JSON file. This simplifies management and ensures consistency across the containerized environment.
- **Flexibility with JSON Policies**: The JSON format allows for fine-grained control over access rules. You can define policies based on user identities, groups, specific Docker API endpoints (e.g., image creation, container management), and even environmental variables. This flexibility empowers administrators to tailor access privileges to specific user roles and functionalities.
- **Lightweight and Extensible**: The AuthZ Broker itself is a lightweight plugin that can be deployed directly on the Docker host machine or even within a container. This approach minimizes resource overhead while offering the potential for future extensibility through additional plugins or integrations.

2.3.2 Implementation

AuthZ Broker enforces user access control in the following ways [17] [18]:

- **API Request Initiation**: When a user interacts with Docker using the Docker CLI or another tool, the request is sent to the Docker remote API.
- **AuthZ Broker Interception**: The AuthZ Broker intercepts the communication between the Docker remote API and the Docker daemon. It acts as a security checkpoint, preventing unauthorized access attempts from reaching the core container engine.

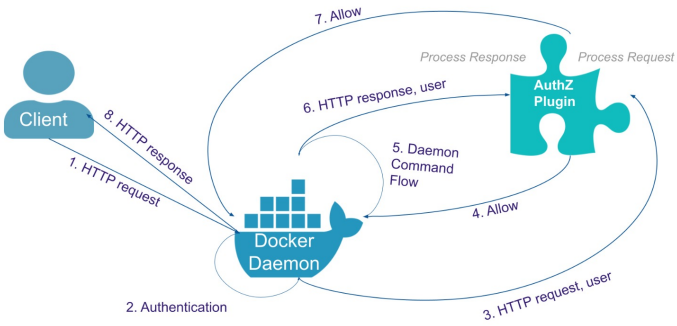
```
┌──(kali㉿kali)-[~/docker-proj]
└─$ sudo dockerd \
    --tlsverify \
    --tlscacert=ca.pem \
    --tlscert=server-cert.pem \
    --tlskey=server-key.pem \
    -H=0.0.0.0:2376 --userns-remap=default --authorization-plugin=authz-broker
[sudo] password for kali:
INFO[2024-04-15T20:23:42.154851841-04:00] Starting up
INFO[2024-04-15T20:23:42.155019292-04:00] User namespaces: ID ranges will be mapped to subuid
INFO[2024-04-15T20:23:42.156382035-04:00] User namespaces: ID ranges will be mapped to subuid
INFO[2024-04-15T20:23:42.176283791-04:00] [graphdriver] using prior storage driver: overlay2
INFO[2024-04-15T20:23:42.177579801-04:00] Loading containers: start.
```

```
┌──(kali㉿kali)-[~/docker-proj]
└─$ sudo systemctl status twistlock-authz.service
[sudo] password for kali:
● twistlock-authz.service - Twistlock docker authorization plugin
     Loaded: loaded (/lib/systemd/system/twistlock-authz.service; enabled; preset: disabled)
     Active: active (running) since Wed 2024-04-10 16:29:23 EDT; 5 days ago
   Main PID: 70519 (authz-broker)
      Tasks: 5 (limit: 7585)
     Memory: 2.5M
        CPU: 14ms
     CGroup: /system.slice/twistlock-authz.service
             └─70519 /usr/bin/authz-broker

Apr 10 16:29:23 kali systemd[1]: Started twistlock-authz.service - Twistlock docker authorization plugin.
Apr 10 16:29:23 kali authz-broker[70519]: time="2024-04-10T16:29:23-04:00" level=info msg="Loaded '1' policies"
```

- **Policy Evaluation**: The AuthZ Broker retrieves the user's identity from the Docker client or the operating system environment. It then compares the user's identity and the requested API endpoint against the predefined authorization policies in the JSON file.

```
└─$ sudo cat /var/lib/authz-broker/policy.json
[sudo] password for kali:
{"name":"policy_5","users":["client"],"actions":["container"]}

┌──(kali㉿kali)-[~/docker-proj]
└─$ docker pull ubuntu
Using default tag: latest
Error response from daemon: authorization denied by plugin authz-broker: act
```

- **Access Granted/Denied**: Based on the policy evaluation, the AuthZ Broker makes a decision. If the user has the necessary permissions for the requested API endpoint, the plugin forwards the request to the Docker daemon for processing. Conversely, if the policy denies access, the AuthZ Broker rejects the request, preventing unauthorized actions from compromising the container environment.



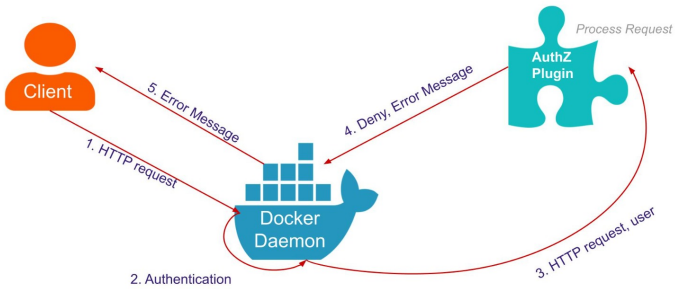


Table 1: TABLE OF CONTRIBUTION

Members/ Contributions	*Parth Vakil*	*Malvik Chauhan*	*Fahima Noor*	*Adit Vasava*	*Mohammad Tawsif Chowdhury*	*Dennis Tank*	*Peter Vourantonis*	*Prashant Parmar*
Leaky Vessels	*YES*	*NO*	*NO*	*NO*	*NO*	*NO*	*YES*	*NO*
Docker Socket Exploit	*NO*	*NO*	*YES*	*YES*	*NO*	*YES*	*NO*	*NO*
SYS_MODULE Capability Exploit	*NO*	*YES*	*NO*	*YES*	*NO*	*NO*	*NO*	*YES*
Leaky Vessels Fixes	*YES*	*NO*	*NO*	*NO*	*NO*	*NO*	*YES*	*NO*
Seccomp Implementation	*NO*	*NO*	*NO*	*NO*	*YES*	*NO*	*NO*	*YES*
AuthZ Broker	*NO*	*YES*	*YES*	*NO*	*YES*	*YES*	*NO*	*NO*
Presentation	*YES*	*YES*	*YES*	*YES*	*YES*	*YES*	*YES*	*YES*
Final Report	*YES*	*YES*	*YES*	*YES*	*YES*	*YES*	*YES*	*YES*

Acknowledgment

The successful completion of this project would not have been possible without the guidance and support of many. We would like to express our gratitude to our professors and mentors for their invaluable advice and feedback. Their expertise and insights greatly enhanced our understanding of Docker and Linux. We also want to thank our peers for their constructive criticism and encouragement throughout the project. Their perspectives and suggestions were instrumental in shaping this report. Lastly, we are grateful for the resources that facilitated our analysis of the exploits and defense strategies. These resources enabled us to delve deeper into the vulnerabilities and devise effective remediation strategies. We hope that our work contributes to the ongoing efforts to make Docker containers more secure.

References

[1] Jiang Wenhao and Li Zheng. Vulnerability analysis and security research of docker container. In *2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education (ICISCAE)*, pages 354–357, 2020.

[2] Docker: Accelerated container application development. https://www.docker.com/, 2024. Accessed: 2024-04-15.

[3] Leaky vessels: Deep dive on container escape vulnerabilities — wiz blog. https://www.wiz.io/blog/leaky-vessels-container-escape-vulnerabilities, 2024. Accessed: 2024-04-15.

[4] Container escape: New vulnerabilities affecting docker and runc - palo alto networks blog. https://www.paloaltonetworks.com/blog/prisma-cloud/leaky-vessels-vulnerabilities-container-escape/, 2024. Accessed: 2024-04-15.

[5] opencontainers/runc: Cli tool for spawning and running containers according to the oci specification. https://github.com/opencontainers/runc, 2024. Accessed: 2024-04-15.

[6] Introducing runc: a lightweight universal container runtime — docker. https://www.docker.com/blog/runc/, 2024. Accessed: 2024-04-15.

[7] Vulnerabilities in docker, other container engines enable host os access — cso online. https://www.csoonline.com/article/1303004/vulnerabilities-in-docker-other-container-engines-enable-host-os-access.html, 2024. Accessed: 2024-04-15.

[8] Protect the docker daemon socket — docker docs. https://docs.docker.com/engine/security/protect-access/, 2024. Accessed: 2024-04-15.

[9] Get informed of the risks associated with docker.sock — techtarget. https://www.techtarget.com/searchitoperations/tip/Get-informed-of-the-risks-associated-with-dockersock, 2019. Accessed: 2024-04-15.

[10] var/run/docker.sock. https://www.educative.io/answers/var-run-dockersock, 2024. Accessed: 2024-04-15.

[11] The danger of exposing docker.sock. https://dejandayoff.com/the-danger-of-exposing-docker.sock/, 2024. Accessed: 2024-04-15.

[12] dockerd — docker docs. https://docs.docker.com/reference/cli/dockerd/, 2024. Accessed: 2024-04-15.

[13] Protect the docker daemon socket — docker docs. https://docs.docker.com/engine/security/protect-access/, 2024. Accessed: 2024-04-15.

[14] Sys-module. https://www.hackitude.in/docker-security/container-breakouts/abusing-capabilities/sys_module, 2024. Accessed: 2024-04-15.

[15] Kubernetes - commands and capabilities. https://jamesdefabia.github.io/docs/user-guide/containers/, 2016. Accessed: 2024-04-15.

[16] Access authorization plugin — docker docs. https://docs.docker.com/engine/extend/plugins_authorization/, 2024. Accessed: 2024-04-15.

[17] twistlock/authz: Docker authorization plugin. https://github.com/twistlock/authz, 2017. Accessed: 2024-04-15.

[18] Authz authn - docker access authorization plugin — hacktricks — hacktricks. https://book.hacktricks.xyz/linux-hardening/privilege-escalation/docker-security/authz-and-authn-docker-access-authorization-plugin, 2024. Accessed: 2024-04-15.