DOCKER SECURITY

starts at 11:05 am

AGENDA

1. Security

    1. Signing Images and Docker Content Trust

    2. Default Docker Engine Security

    4. Docker MTLS

    5. Securing the Docker Daemon HTTP socket

2. Docker in Docker

Signing Images and Docker Content Trust

Importance of Image Signing

① Ensuring image Integrity
② Authenticity Verification
③ Compliance and Security Policies

DCT → notary. → gurantees the provenance of content.

export DOCKER_CONTENT_TRUST=1 → enable DCT

docker pull library/alpine:latest → pull a signed image

docker trust inspect image:tag → Check signature and signers

example of unsigned image

docker trust inspect nigelpoulton/dockerbook:unsigned

docker pull " " → fails → as image is not signed.

→ Signing Your Own Image

Login to Dockerhub.

Dockerfile

```
FROM alpine:latest

WORKDIR /app

COPY hello.txt /app/hello.txt

CMD ["cat", "/app/hello.txt"]
```

```
echo "Hello, Docker Content Trust!" > hello.txt
```

docker build -t vedant120/app:latest . -> When DCT is enabled image will be automatically signed once you push to dockerhub.

```
docker push vedant120/app:latest
```

```
docker trust inspect vedant120/app:latest
```

Disabling trust

```
export DOCKER_CONTENT_TRUST=0
```

Remove Signature from an Image

```
docker trust revoke image
```

→ Default Docker Engine Security

① Namespaces

```
docker exec -it container_nginx lsns
```

Types of Namespaces:

- **time**: This namespace is used for managing system time. Each namespace can have its own clock.

- **user**: This namespace isolates user and group IDs, allowing processes to have a different set of user IDs (UIDs) and group IDs (GIDs) inside the container than they have on the host system.

- **mnt**: This namespace isolates the mount points for filesystems. Each namespace has its own view of the file system hierarchy.

- **uts**: This namespace isolates hostname and domain name. It allows containers to have different hostnames and domain names from the host system.

- **ipc**: This namespace isolates inter-process communication (IPC) resources, such as semaphores, message queues, and shared memory.

- **pid**: This namespace isolates process IDs, allowing containers to have their own process ID space. Processes inside the container see a different PID space than the host system.

- **cgroup**: This namespace isolates cgroups, which are used to manage and limit the resources (CPU, memory, etc.) that processes can use.

- **net**: This namespace isolates networking, allowing the container to have its own network interfaces, routing tables, and firewall rules. Each container can have a separate network stack.

② Control groups.

③ Capabilities.

set of permissions or rules → processes inside the container.

Common Capabilities

Linux defines a wide range of capabilities, and Docker uses a subset of them. Below are a few commonly used capabilities:

1. **CAP_NET_ADMIN**: Allows a process to modify network settings, such as bringing up or down network interfaces, changing IP addresses, and modifying routing tables. It’s a common capability needed for network configuration in containers.

2. **CAP_SYS_ADMIN**: This is a powerful capability that allows processes to perform system-level administrative tasks, like mounting filesystems, setting system time, or loading kernel modules. This is usually avoided in containers due to the high level of access it provides.

3. **CAP_DAC_OVERRIDE**: Allows processes to bypass file read, write, and execute permission checks. It effectively disables file permission checking, which can be useful in some cases but should be used with caution due to security risks.

4. **CAP_SYS_TIME**: Grants the process permission to change the system clock or set the time zone.

5. **CAP_CHOWN**: Grants the process the ability to change the ownership of files (users and groups).

6. **CAP_KILL**: Allows a process to send signals to other processes, including those belonging to other users.

7. **CAP_NET_BIND_SERVICE**: Allows a process to bind to ports below 1024, which are normally restricted to privileged processes.

8. **CAP_SYS_PTRACE**: Enables the process to trace other processes, which can be useful for debugging or debugging tools, but can also be a security risk

```
docker run -dit container_ubuntu ubuntu
```

execute inside the container

```
apt-get update && apt-get install -y libcap2-bin
```

```
capsh --print
```

In Docker, the bounding set defines the maximum capabilities a container can have. It acts as a limit on the privileges a container process can acquire, even if the process tries to escalate its privileges.

Understanding Capability Sets:

Bounding Set: This set represents the upper limit of capabilities a container can have. It's like a "cap" on the potential privileges.

Permitted Set: This set contains the capabilities a process is currently allowed to use.

Inheritable Set: This set determines which capabilities can be passed on to child processes.

RUN a Container with specific capabilities

```
docker run -it --rm --cap-add=NET_ADMIN ubuntu bash
```

--cap-add

You can specify multiple capabilities using multiple `--cap-add` options, like so:

```
docker run -it --rm --cap-add=NET_ADMIN --cap-add=SYS_TIME ubuntu bash
```

Run a container without certain capabilities

```
docker run -it --rm --cap-drop=SYS_TIME ubuntu bash
```

--cap-drop

Add Capabilities to an Already Running Container

```
docker update --cap-add=NET_ADMIN container_name
```

④ Seccomp. Secure Computing Mode.

Seccomp (short for **Secure Computing Mode**) is a Linux kernel feature that allows you to restrict the system calls a process can make, enhancing the security of the system by reducing the attack surface.

Profile seccomp.

```
{
  "defaultAction": "SCMP_ACT_ALLOW",
  "syscalls": [
    {
      "names": ["execve", "exit_group"],
      "action": "SCMP_ACT_ALLOW"
    },
    {
      "names": ["kill", "ptrace"],
      "action": "SCMP_ACT_ERRNO"
    }
  ]
}
```

Run a Container with a custom seccomp file

```
docker run -d --security-opt seccomp=/path/to/seccomp/profile.json nginx
```

(5) App Armor SELinux

→ mandatory access Control

AppArmor.

Policies $\xrightarrow[\text{on}]{\text{imp}}$ (Containers (Appli, process)

Profiles → we define what application is allowed to do.

```
profile my_docker_container flags=(attach_disconnected) {
  # Allow container to read /etc/hostname
  /etc/hostname r,
  # Allow container to execute /bin/bash
  /bin/bash ixr,
  # Deny write access to any file system
  deny /** w,
  # Log denied access attempts
```

i → inherit the permission.

```
 audit deny /** w,

}
```

Load the AppArmor profile:

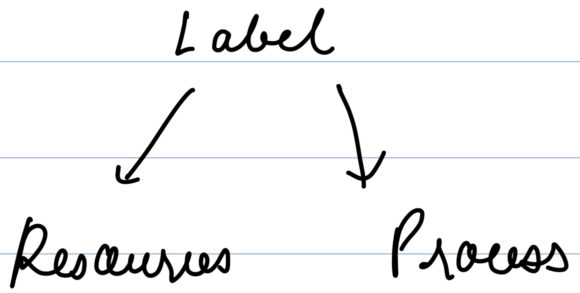
```
sudo apparmor_parser -r /etc/apparmor.d/my_docker_container
```

→ profile name

```
docker run --security-opt apparmor=my_docker_container my_image
```

→ SELinux   Security Enhaned Linux

AppArmor Policy. → Containers
SELinux Policy → Resources.



Policies → rules written based on these labels.

SELinux → 3 modes.

① Enforcing → enforced and logged
② Permissive → only logged.
③ Disabled. → policies are not applied.

⑥ Read Only Filesystem (for container)

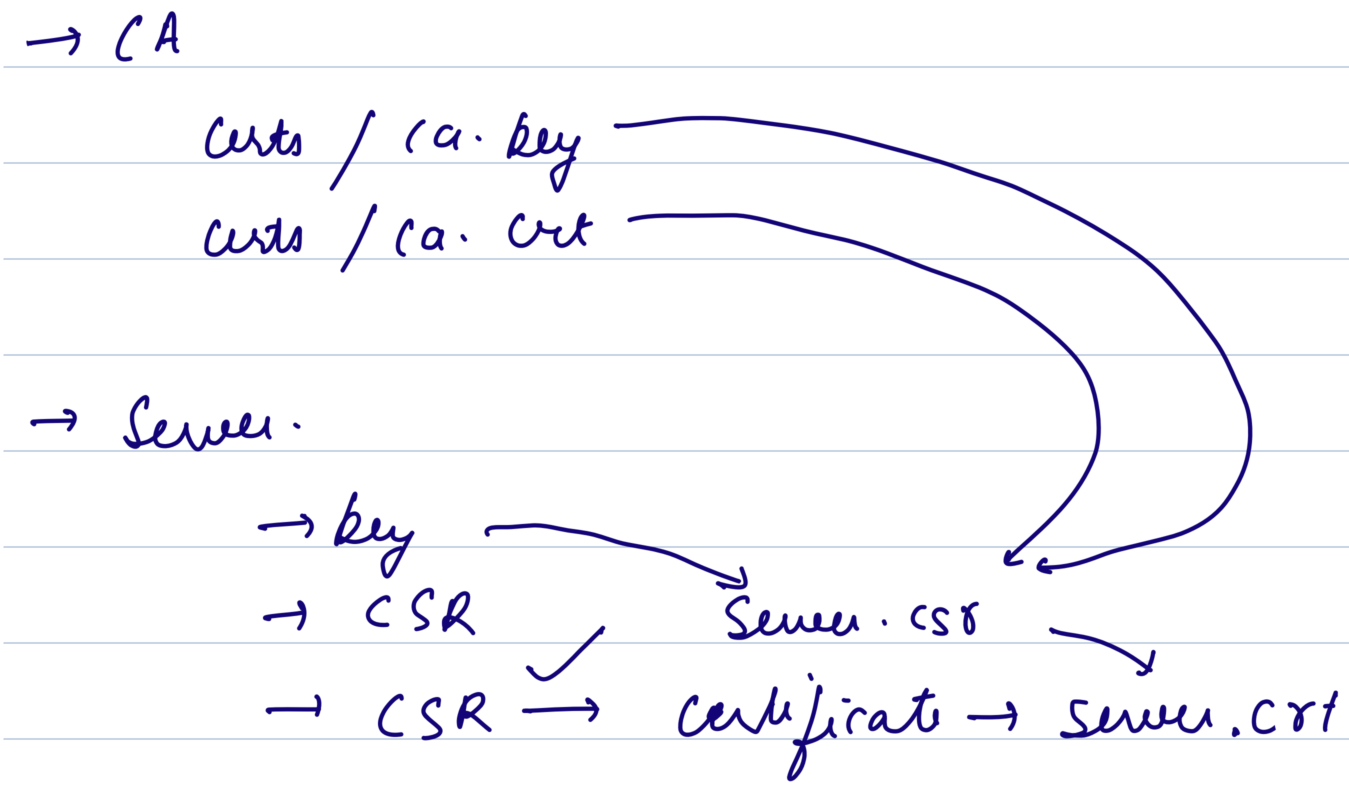
```
docker run -d --read-only nginx
```

Break. 12:40 pm

Docker mTLS
mutual TLS

importance of mTLS
① enhanced security
② Data encryption
③ Compliance.

① CA.
② Server Cert
③ Client Certs



→ CA

certs / ca.key

certs / ca.crt

→ Server.

→ key

→ CSR

server.csr

→ CSR → certificate → server.crt

→ Client certificates.

→ key

→ CSR

→ cert

create directory.

```
mkdir -p mTLS-demo/{certs,server}
```

```
cd mTLS-demo
```

For CA

Generate the Private Key

```
openssl genrsa -out certs/ca.key 2048
```

Create a self signed Certificate

```
openssl req -x509 -new -nodes -key certs/ca.key -sha256 -days 365 -out certs/ca.crt -subj "/CN=MyCA"
```

For server

certs/openssl.cnf

```
[ req ]
default_bits       = 2048
distinguished_name = req_distinguished_name
req_extensions     = v3_req
prompt             = no

[ req_distinguished_name ]
CN = localhost

[ v3_req ]
keyUsage = keyEncipherment, dataEncipherment
extendedKeyUsage = serverAuth
subjectAltName = @alt_names

[ alt_names ]
DNS.1 = localhost
```

```
openssl genrsa -out certs/server.key 2048
```

→ key

```
openssl req -new -key certs/server.key -out certs/server.csr -config certs/openssl.cnf
```

→ CSR

```
openssl x509 -req -in certs/server.csr -CA certs/ca.crt -CAkey certs/ca.key -CAcreateserial -out certs/server.crt -days 365 -extensions v3_req -extfile certs/openssl.cnf
```

→ Certificate

generate client certificate

```
openssl genrsa -out certs/client.key 2048

openssl req -new -key certs/client.key -out certs/client.csr -subj "/CN=client"

openssl x509 -req -in certs/client.csr -CA certs/ca.crt -CAkey certs/ca.key -CAcreateserial -out certs/client.crt -days 365
```

nginx. conf

```
server/nginx.conf

events { }


http {

  server {

    listen 443 ssl;

    ssl_certificate /etc/nginx/server.crt;

    ssl_certificate_key /etc/nginx/server.key;

    ssl_client_certificate /etc/nginx/ca.crt;

    ssl_verify_client on;
```

```
        location / {
            return 200 'mTLS verification successful!';
        }
    }
}
```

Created a Dockerfile

server/Dockerfile

```
FROM nginx:latest
COPY server.crt /etc/nginx/server.crt
COPY server.key /etc/nginx/server.key
COPY ca.crt /etc/nginx/ca.crt
COPY nginx.conf /etc/nginx/nginx.conf
```

Copy necessary files to server directory

```
cp certs/server.crt certs/server.key certs/ca.crt server/
```

Build the docker image

```
docker build -t mtls-server ./server
```

Create a custom Network

```
docker network create mtls-network
```

Run the container

```
docker run -d --name mtls-server --network mtls-network -p 443:443 mtls-server
```

→ Verify.

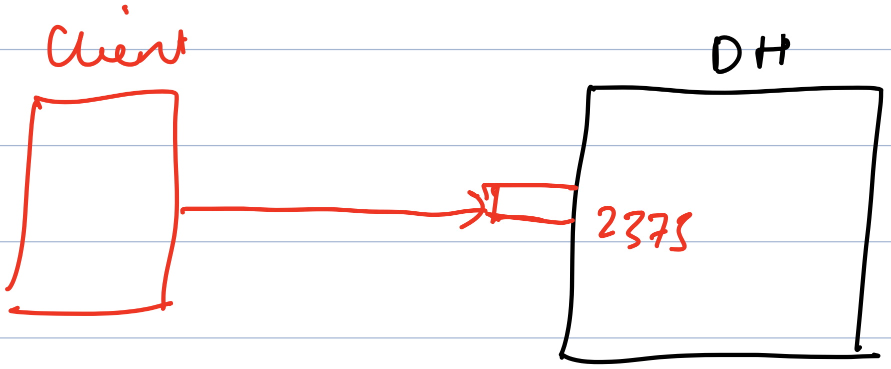
```
curl --cert certs/client.crt --key certs/client.key --cacert certs/ca.crt https://localhost
```

```
mTLS verification successful!
```

Securing the Docker Daemon HTTP Socket

① Unix Socket → /var/run/docker.sock

→ 2375 → Remote connections. (TCP)



Why securing Docker Daemon is important:

Attacker can

- **Gain control of the host system**: Since Docker runs as the root user by default, compromising the Docker socket can allow an attacker to execute arbitrary commands as root.

- **Escalate privileges**: If attackers can manipulate the Docker daemon, they can create containers with privileges that can escalate to the host system.

- **Access sensitive data**: Docker stores sensitive data, such as images, secrets, and logs, which an attacker could steal if they gain access to the API.

Methods for Securing the Docker Daemon Socket.

(1) Using TLS to Secure DD.

Generate a CA certificate (Certificate Authority), server certificate, and key.

```
mkdir -p /etc/docker/certs

cd /etc/docker/certs
```

Generate a CA key and certificate

```
openssl genrsa -out ca.key 2048

openssl req -x509 -new -key ca.key -out ca.crt -days 3650
```

Generate server key and certificate

```
openssl genrsa -out server.key 2048

openssl req -new -key server.key -out server.csr

openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server.crt -days 3650
```

Generate Client key and certificate

```
openssl genrsa -out client-key.pem 2048

openssl req -new -key client-key.pem -out client.csr -subj "/CN=client"

openssl x509 -req -in client.csr -CA ca.pem -CAkey ca-key.pem -CAcreateserial -out client-cert.pem -days 365 -sha256
```

/etc/docker/daemon.json

```
{
  "host": ["tcp://0.0.0.0:2376", "unix:///var/run/docker.sock"],
  "tlsverify": true,
  "tlscacert": "/etc/docker/certs/ca.crt",
  "tlscert": "/etc/docker/certs/server.crt",
  "tlskey": "/etc/docker/certs/server.key"
}
```

→ restart docker.

→ Verify

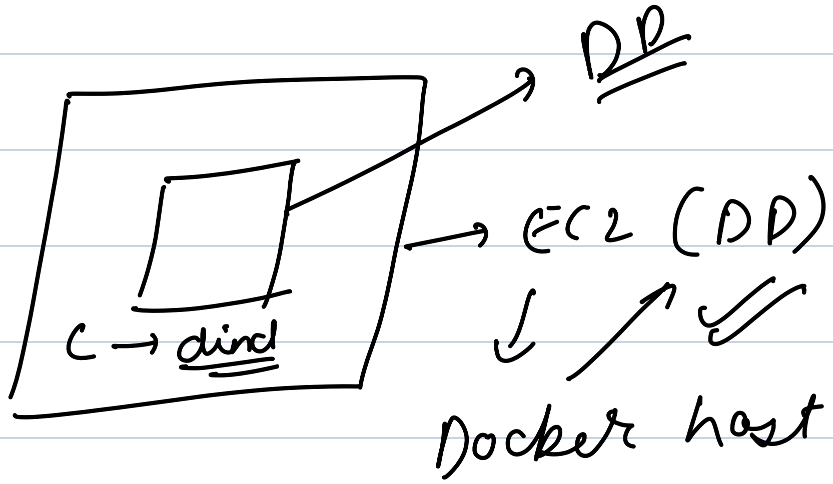
```
docker info (From local will work)
```

```
docker -H tcp://127.0.0.1:2376 info -> This should fail
```

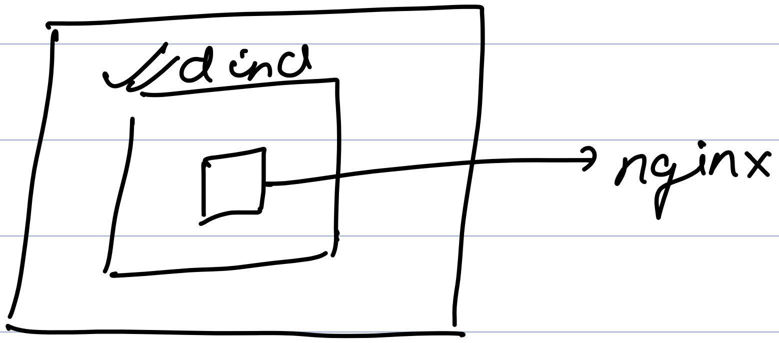
```
docker --tlsverify \
  --tlscacert=/etc/docker/certs/ca.pem \
  --tlscert=/etc/docker/certs/client-cert.pem \
  --tlskey=/etc/docker/certs/client-key.pem \
  -H=tcp://<server_ip>:2376 info
```

```
sudo ufw allow from <trusted_ip> to any port 2376
```

Docker in Docker

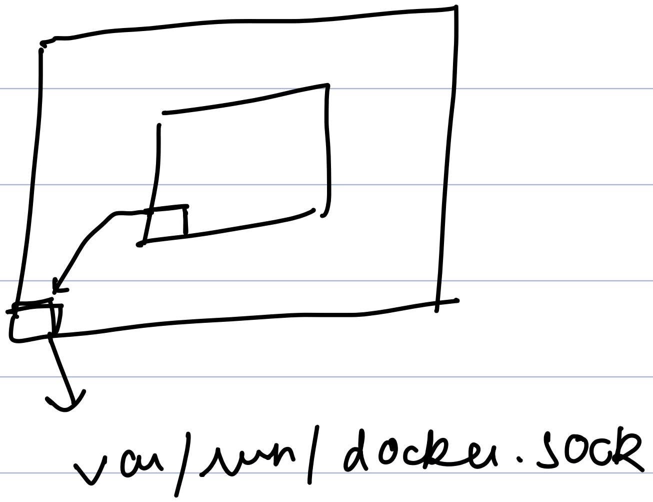


→ Jenkins.



```
docker run --privileged --name dind-container -d docker:dind
```

```
`docker run -v /var/run/docker.sock:/var/run/docker.sock -it docker:dind`
```


var/run/docker.sock