ATTACKING DOCKER

WITH SECURITY BEST PRACTICE

WWW.DEVSECOPSGUIDES.COM

Attacking Docker

· Mar 25, 2024 · 15 min read

Table of contents

Docker has revolutionized the way software is developed, deployed, and managed by providing a lightweight, portable, and scalable platform for containerization. However, with the widespread adoption of Docker, it has become an attractive target for attackers seeking to exploit vulnerabilities and compromise containerized applications. Understanding the potential attack vectors and security risks associated with Docker environments is crucial for organizations to protect their assets and maintain the integrity of their containerized infrastructure.

Attacking Docker involves exploiting vulnerabilities, misconfigurations, and weaknesses in various components of the Docker ecosystem, including Docker

Engine, container images, orchestration platforms (e.g., Kubernetes), and the underlying host system. Attackers may leverage techniques such as container breakout, privilege escalation, denial-of-service (DoS), image tampering, and lateral movement to compromise Docker environments and gain unauthorized access to sensitive data or resources.

Privileged Container

Running containers with elevated privileges, as in the case of privileged containers, poses a significant security risk. When a container is launched in privileged mode, it gains access to the host system with root-level permissions, essentially breaking down the isolation barriers that containers are designed to provide. This opens up the possibility of various attacks, including:

1. **Host System Compromise**: With privileged access, an attacker can potentially gain control over the underlying host system. They can execute arbitrary commands, modify critical system files, and even install malware or backdoors, essentially taking over the entire infrastructure.
2. **Privilege Escalation**: Once inside a privileged container, an attacker can attempt to escalate their privileges further by exploiting vulnerabilities within the container runtime or kernel. This could lead to full root access on the host system, bypassing any security measures in place.
3. **Data Breach**: Privileged containers have unrestricted access to the host system's resources, including sensitive data stored on other containers or the host itself. Attackers can exploit this access to steal valuable data, such as credentials, customer information, or intellectual property.

Here's an example of noncompliant and compliant code with Dockerfiles and commands:

COPY

```
# Noncompliant: Privileged container

FROM ubuntu
...
```

```
# Running container in privileged mode
RUN docker run -it --privileged ubuntu /bin/bash
```

In this noncompliant code, the container is launched with the `--privileged` flag, granting it elevated privileges. This exposes the host system to potential attacks.

In the compliant code, the container is launched without the `--privileged` flag, ensuring that it runs with normal user-level privileges. This reduces the risk of unauthorized access and privilege escalation, thus enhancing security.

COPY

```
# Compliant: Non-privileged container

FROM ubuntu
...
# Running container without privileged mode
RUN docker run -it ubuntu /bin/bash
```

Exposed Container APIs

Exposing container APIs without proper authentication or access controls can lead to various security vulnerabilities, making them susceptible to attacks. Here's a description of potential attacks and how to address them, along with example code for noncompliant and compliant configurations:

Attacks:

1. **Unauthorized Access**: Attackers may exploit exposed container APIs to gain unauthorized access to sensitive information or perform malicious actions within the container environment.
2. **Data Manipulation**: Without proper authentication and access controls, attackers can manipulate data stored within containers or inject malicious code, potentially leading to data breaches or system compromise.

3. **Denial-of-Service (DoS)**: Exposed APIs without rate limiting or throttling mechanisms are susceptible to DoS attacks, where attackers overwhelm the API with excessive requests, causing service disruptions.

Noncompliant Code:

```
# Noncompliant: Exposed container API
without authentication/authorization

FROM nginx
...
# Expose container API on port 8080
EXPOSE 8080
```

In this noncompliant code, the container's API is exposed on port 8080 without any authentication or authorization mechanisms, leaving it vulnerable to attacks.

Compliant Code:

```
# Compliant: Secured container API with authentication/authorization

FROM nginx
...
# Expose container API on port 8080 (internal)
EXPOSE 8080

# Use a reverse proxy or API gateway for authentication/authorization
```

The compliant code addresses the vulnerability by exposing the container's API internally on port 8080 and leveraging a reverse proxy or API gateway for authentication and authorization.

Container Escape

Container escape refers to the exploitation of vulnerabilities within the container runtime or misconfigurations to break out of the container's isolated environment and gain unauthorized access to the host operating system. This type of attack poses a significant threat to containerized environments as it undermines the fundamental principle of containerization, which is to provide secure isolation for applications.

Attacks:

1. **Kernel Vulnerabilities**: Attackers may exploit vulnerabilities in the container runtime or the underlying kernel to execute arbitrary code and gain access to the host system.
2. **Misconfigured Security Settings**: Improperly configured security settings, such as running containers in privileged mode or granting excessive capabilities, can provide attackers with the opportunity to escape the container's isolation.
3. **Shared Resources**: Containers often share resources with the host system, such as network interfaces or file systems. Attackers can leverage these shared resources to break out of the container and access sensitive data or perform unauthorized actions on the host.

Noncompliant Code:

```
# Noncompliant: Running a container without proper security isolation

require 'docker'

# Create a container with default settings
container = Docker::Container.create('Image' => 'nginx')
container.start
```

In the noncompliant code, a container is created and started without any security isolation measures, making it vulnerable to container escape attacks.

Compliant Code:

```
# Compliant: Running a container with enhanced security isolation

require 'docker'

# Create a container with enhanced security settings
container = Docker::Container.create(
  'Image' => 'nginx',
  'HostConfig' => {
    'Privileged' => false,                    # Disable privileged
mode
    'CapDrop' => ['ALL'],                     # Drop all
capabilities
    'SecurityOpt' => ['no-new-privileges']    # Prevent privilege
escalation
  }
)
container.start
```

The compliant code introduces security enhancements to mitigate the risk of container escape. Specifically, it disables privileged mode, drops all capabilities from the container, and prevents privilege escalation within the container. These measures help to enforce proper security isolation and reduce the attack surface for potential container escape vulnerabilities.

Container Image Tampering

Container image tampering involves modifying or replacing container images with malicious versions, which may contain malware, backdoors, or vulnerable components. Attackers can exploit these tampered images to compromise the security of containerized applications. Here's an overview of the issue along with example code for both noncompliant and compliant scenarios:

Attacks:

1. **Malicious Code Injection**: Attackers may inject malicious code into container images, which can compromise the security of the application or the entire

container infrastructure.

2. **Data Theft**: Tampered container images might contain code designed to steal sensitive information, such as credentials, financial data, or intellectual property.
3. **Exploitation of Vulnerabilities**: Attackers can replace legitimate container images with versions containing known vulnerabilities, allowing them to exploit these vulnerabilities to gain unauthorized access or perform malicious actions.

Noncompliant Code:

```
# Noncompliant: Pulling and running a
container image without verifying integrity

require 'docker'

# Pull the container image
image = Docker::Image.create('fromImage' => 'nginx')

# Run the container image
container = Docker::Container.create('Image' => image.id)
container.start
```

In the noncompliant code, a container image is pulled and run without any verification of its integrity, making the application vulnerable to container image tampering attacks.

Compliant Code:

```
# Compliant: Pulling and running a
container image with integrity verification

require 'docker'
require 'digest'

# Image name and tag
```

```
image_name = 'nginx'
image_tag = 'latest'

# Pull the container image
image = Docker::Image.create('fromImage' => "#{image_name}:#
{image_tag}")

# Verify the integrity of the pulled image
expected_digest =
Digest::SHA256.hexdigest(image.connection.get("/images/#
{image.id}/json").body)
actual_digest = image.info['RepoDigests'].first.split('@').last
if expected_digest != actual_digest
  raise "Integrity verification failed for image: #{image_name}:#
{image_tag}"
end

# Run the container image
container = Docker::Container.create('Image' => image.id)
container.start
```

The compliant code addresses the issue by introducing integrity verification. It calculates the expected digest of the pulled image using the SHA256 hash algorithm and compares it with the actual digest obtained from the Docker API. If the digests do not match, an integrity verification failure is raised, indicating a potential tampering of the container image. This helps to ensure that only trusted and unaltered images are used, mitigating the risk of container image tampering attacks.

Insecure Container Configuration

Insecure container configuration refers to misconfigurations in container settings, such as weak access controls or excessive permissions, which can lead to security vulnerabilities. These vulnerabilities may allow attackers to compromise the container or its environment. Here's an overview of the issue along with example code for both noncompliant and compliant scenarios:

Attacks:

1. **Privilege Escalation**: Weak access controls or excessive permissions may allow attackers to escalate their privileges within the container, gaining unauthorized access to sensitive resources or compromising the host system.
2. **Tampering and Modification**: Insecure configurations, such as writable filesystems or insecure mount points, may enable attackers to tamper with the container's contents or inject malicious code, leading to unauthorized modifications or data breaches.
3. **Exposure of Sensitive Resources**: Misconfigured network settings or exposed ports may expose sensitive services or data to unauthorized access, increasing the risk of exploitation.

Noncompliant Code:

```
# Noncompliant: Running a container with insecure configuration

require 'docker'

# Create a container with default settings
container = Docker::Container.create('Image' => 'nginx')
container.start
```

In the noncompliant code, a container is created and started with default settings, which may have insecure configurations, potentially leading to vulnerabilities.

Compliant Code:

```
# Compliant: Running a container with secure configuration

require 'docker'

# Create a container with secure settings
container = Docker::Container.create(
  'Image' => 'nginx',
```

```
  'HostConfig' => {
    'ReadOnly' => true,                          # Set container as
read-only
    'CapDrop' => ['ALL'],                        # Drop all capabilities
    'SecurityOpt' => ['no-new-privileges'],      # Prevent privilege
escalation
    'NetworkMode' => 'bridge',                   # Use a bridge network
for isolation
    'PortBindings' => { '80/tcp' => [{ 'HostPort' => '8080' }] }  #
Bind container port to a specific host port
  }
)
container.start
```

The compliant code addresses security concerns by applying secure container configurations:

- **ReadOnly**: Setting the container's filesystem as read-only prevents potential tampering and unauthorized modifications.
- **CapDrop**: Dropping all capabilities from the container minimizes the attack surface and reduces the potential impact of privilege escalation.
- **SecurityOpt**: Preventing the container from gaining additional privileges helps mitigate the risk of privilege escalation.
- **NetworkMode**: Using a bridge network for isolation ensures separation from the host and other containers.
- **PortBindings**: Binding the container's port to a specific host port restricts network access to the container and avoids exposing unnecessary ports, reducing the attack surface.

Denial-of-Service (DoS)

Denial-of-Service (DoS) attacks target the availability of containerized applications by overloading container resources or exploiting vulnerabilities in the container runtime.

Here's an overview of the issue along with example code for both noncompliant and compliant scenarios:

Attacks:

1. **Resource Overloading**: Attackers may flood the container with excessive requests, causing it to consume all available resources such as CPU, memory, or network bandwidth, thereby rendering the application unresponsive to legitimate users.
2. **Exploiting Runtime Vulnerabilities**: Vulnerabilities in the container runtime or application code can be exploited to exhaust resources or cause the container to crash, leading to a denial of service.

Noncompliant Code:

```
# Noncompliant: Vulnerable Dockerfile
with unlimited resource allocation

FROM nginx:latest

COPY app /usr/share/nginx/html

EXPOSE 80

CMD ["nginx", "-g", "daemon off;"]
```

In the noncompliant code, the Dockerfile does not implement any resource limitations or restrictions, allowing the container to consume unlimited resources. This makes it vulnerable to DoS attacks if an attacker overwhelms the container with excessive requests or exploits vulnerabilities in the container runtime.

Compliant Code:

```
# Compliant: Docker-compose file with resource limitations

version: '3'
services:
  nginx:
    image: nginx:latest
    ports:
      - 80:80
    volumes:
      - ./app:/usr/share/nginx/html
    deploy:
      resources:
        limits:
          cpus: '0.5'
          memory: '256M'
```

The compliant code addresses the vulnerability by implementing resource management and limiting container resources based on the application's requirements and available environment resources. In this example using Docker Compose, resource limits for CPU (`cpus`) and memory (`memory`) are set to restrict resource usage. Adjust these limits according to your application's needs and the resources available in your environment to ensure reliable application performance and protect against DoS attacks.

Kernel Vulnerabilities

Kernel vulnerabilities pose a significant threat to containerized environments as they can be exploited to gain unauthorized access or control over containers. Here's an overview of the issue along with example code for both noncompliant and compliant scenarios:

Attacks:

1. **Privilege Escalation**: Attackers may exploit vulnerabilities in the kernel to escalate their privileges within containers, potentially gaining full control over the host

system.

2. **Container Breakout**: Kernel vulnerabilities can be exploited to break out of container isolation and access sensitive resources or compromise other containers running on the same host.

Noncompliant Code:

```
# Noncompliant: Ignoring kernel vulnerabilities

docker run -d ubuntu:latest /bin/bash
```

In the noncompliant code, a Docker container is created without considering potential kernel vulnerabilities. This leaves the container and the host system vulnerable to exploitation.

Compliant Code:

```
# Compliant: Checking kernel vulnerabilities

# Perform vulnerability assessment using kubehunter
kubehunter scan

# Check the output for kernel vulnerabilities

# If vulnerabilities are found, take necessary steps to address them

# Create the Docker container
docker run -d ubuntu:latest /bin/bash
```

The compliant code snippet incorporates checking for kernel vulnerabilities using the kubehunter tool before creating the Docker container. Here's how it works:

1. **Vulnerability Assessment**: The kubehunter tool is used to perform a vulnerability assessment, including checking for kernel vulnerabilities.
2. **Examine Output**: The output of the tool is examined to identify any kernel vulnerabilities that may pose a risk to the containerized environment.
3. **Address Vulnerabilities**: If vulnerabilities are found, appropriate steps are taken to address them before creating the Docker container. This may involve applying security patches, updating the kernel, or implementing additional security measures.

By proactively assessing and addressing kernel vulnerabilities, organizations can reduce the risk of exploitation and enhance the security of their containerized environments.

Shared Kernel Exploitation

Shared kernel exploitation refers to the exploitation of vulnerabilities in the kernel by attackers to compromise multiple containers sharing the same kernel on a host. Here's an overview of the issue along with example code for both noncompliant and compliant scenarios:

Attacks:

1. **Container Breakout**: Attackers exploit vulnerabilities in the shared kernel to break out of container isolation and gain unauthorized access to the host system or other containers running on the same host.
2. **Privilege Escalation**: Once an attacker gains access to one container, they may attempt to escalate privileges within the shared kernel environment to compromise other containers or the host system.

Noncompliant Code:

```
# Noncompliant: Vulnerable to container breakout

FROM ubuntu:latest
```

```
# Install vulnerable package
RUN apt-get update && apt-get install -y vulnerable-package

# Run vulnerable application
CMD ["vulnerable-app"]
```

In the noncompliant code, the Docker image installs a vulnerable package and runs a vulnerable application. If an attacker exploits a kernel vulnerability within the container, they could potentially escape the container and compromise the host or other containers.

Compliant Code:

COPY

```
# Compliant: Mitigated container breakout vulnerability

FROM ubuntu:latest

# Install security updates and necessary packages
RUN apt-get update && apt-get upgrade -y && apt-get install -y secure-
package

# Run secure application
CMD ["secure-app"]
```

The compliant code addresses the vulnerability by:

1. **Installing Security Updates**: Regular updates and upgrades are performed to ensure that the container includes the latest security patches and fixes for known vulnerabilities.
2. **Using Secure Packages**: Only necessary and secure packages are installed within the container, reducing the attack surface and minimizing the risk of exploitation.

By running a secure application within the container and keeping the system up-to-date with security patches, the risk of a container breakout due to shared kernel exploitation is significantly reduced. Additionally, organizations can further enhance security by utilizing container isolation techniques, leveraging security-enhanced kernels, and implementing monitoring and logging mechanisms to detect and respond to potential exploitation attempts.

Insecure Container Orchestration

Insecure container orchestration platforms, such as Kubernetes, can introduce vulnerabilities that lead to unauthorized access, privilege escalation, or exposure of sensitive information. Here's an overview of the issue along with example code for both noncompliant and compliant scenarios:

Attacks:

1. **Privilege Escalation**: Attackers may exploit misconfigurations in container orchestration platforms to gain elevated privileges within the cluster, allowing them to perform malicious actions on the host or compromise other containers.
2. **Unauthorized Access**: Vulnerabilities in container orchestration platforms can be exploited to gain unauthorized access to sensitive resources or manipulate configurations, potentially leading to data breaches or service disruptions.

Noncompliant Code:

COPY

```
# Noncompliant: Vulnerable to privilege escalation

apiVersion: v1
kind: Pod
metadata:
  name: vulnerable-pod
spec:
  containers:
    - name: vulnerable-container
      image: vulnerable-image
```

```
      securityContext:
        privileged: true  # Privileged mode enabled
```

In the noncompliant code, the Pod definition enables privileged mode for the container, granting it elevated privileges within the container orchestration environment. This increases the risk of privilege escalation if an attacker gains access to this container.

Compliant Code:

COPY

```
# Compliant: Mitigated privilege escalation

apiVersion: v1
kind: Pod
metadata:
  name: secure-pod
spec:
  containers:
    - name: secure-container
      image: secure-image
      securityContext:
        privileged: false  # Privileged mode disabled
```

The compliant code addresses the vulnerability by explicitly disabling privileged mode for the container. By running containers with reduced privileges, the impact of a potential compromise is limited, and the attack surface is minimized.

In addition to disabling privileged mode, other security measures should be implemented to enhance the security of container orchestration platforms:

- **RBAC Policies**: Configure appropriate Role-Based Access Control (RBAC) policies to restrict access to resources and APIs based on user roles and permissions.
- **Network Segmentation**: Enable network segmentation and isolation to prevent unauthorized access between containers and services.

- **Regular Security Updates**: Apply security patches and updates to the container orchestration platform and underlying infrastructure to address known vulnerabilities.
- **Monitoring and Logging**: Implement monitoring and logging mechanisms to detect and respond to suspicious activities within the orchestration environment.

By following best practices and implementing robust security measures, organizations can mitigate the risks associated with insecure container orchestration platforms and ensure the integrity and availability of their containerized applications.

Insecure Container Images

Insecure container images are those that contain vulnerable or outdated software components, which can be exploited by attackers to compromise the security of containerized applications. Here's an overview of the issue along with example code for identifying insecure container images:

COPY

```
*** Malicious Images via Aqua

- docker-network-bridge-
- ipv6:0.0.2
- docker-network-bridge-
- ipv6:0.0.1
- docker-network-ipv6:0.0.12
- ubuntu:latest
- ubuntu:latest
- ubuntu:18.04
- busybox:latest
- alpine: latest
- alpine-curl
- xmrig:latest
- alpine: 3.13
- dockgeddon: latest
- tornadorangepwn:latest
- jaganod: latest
- redis: latest
```

```
- gin: latest (built on host)
- dockgeddon:latest
- fcminer: latest
- debian:latest
- borg:latest
- docked:latestk8s.gcr.io/pause:0.8
- dockgeddon:latest
- stage2: latest
- dockerlan:latest
- wayren:latest
- basicxmr:latest
- simpledockerxmr:latest
- wscopescan:latest
- small: latest
- app:latest
- Monero-miner: latest
- utnubu:latest
- vbuntu:latest
- swarm-agents:latest
- scope: 1.13.2
- apache:latest
- kimura: 1.0
- xmrig: latest
- sandeep078: latest
- tntbbo:latest
- kuben2

*** Other Images

- OfficialImagee
- Ubuntuu
- Cent0S
- Alp1ne
- Pythoon
```

Identifying Insecure Images:

Here's a list of commands and code snippets that can help identify insecure container images using Aqua Security (a container security platform):

```
# List insecure container images using Aqua CLI
aquactl images list

# Output:
# Malicious Images:
# docker-network-bridge-ipv6:0.0.2
# dockgeddon:latest
# tornadorangepwn:latest
# ...

# List other potentially insecure images
docker images | grep -E
'ubuntu|alpine|debian|busybox|redis|apache|xmrig'
```

COPY

References

- https://devsecopsguides.com/docs/attacks/container/

Devops DevSecOps Docker containers docker images Docker compose Dockerfile

Published on

DevSecOpsGuides

Follow

MORE ARTICLES

RR **Reza Rashidi**

Attacking AWS

As businesses increasingly migrate their operations to Amazon Web Services (AWS), the significance o...

RR **Reza Rashidi**

Attacking Android

In this comprehensive guide, we delve into the world of Android security from an offensive perspecti...