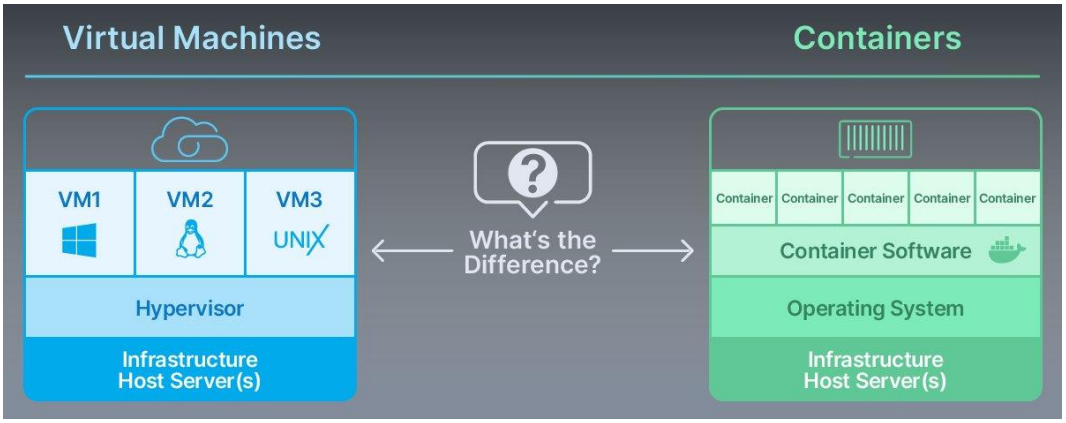
Docker Security

Adam Nashwan & Nabil Mohamed Yakoob

1. Was sind Docker/Container?
   - **Docker/Container** sind **leichtgewichtige, isolierte Laufzeitumgebungen**, in denen Anwendungen mit all ihren Abhängigkeiten (Code, Bibliotheken, Konfigurationen) ausgeführt werden – **unabhängig vom Host-System**.[1]

2. Container vs VM

   - Container teilen sich den **Kernel des Hosts**, daher bei Kernel-Exploits potenzielle Gefahr für das Gesamtsystem
   - Angriffsfläche durch **unsichere oder veraltete Images** (z. B. mit bekannten CVEs)
   - VMs bieten durch **vollständige Virtualisierung** eine stärkere Trennung, aber auf Kosten von Ressourcen
   - Container benötigen **zusätzliche Schutzmechanismen** wie `seccomp`, `AppArmor`, `read-only mounts`, nicht-root-User
   - Bei richtiger Konfiguration sind Container **sicher und effizient**, erfordern aber mehr **Security-by-Design[1]**



3. Docker engine
   - Der **Docker Daemon läuft mit Root-Rechten** → jede Schwachstelle hier kann zur Host-Kompromittierung führen
   - Die **Docker API** sollte niemals ungeschützt (z. B. via TCP) öffentlich erreichbar sein
   - **Zugriff auf den Daemon (z. B. via `docker.sock`) erlaubt Root-Äquivalenz** – nur vertrauenswürdigen Prozessen/Benutzern Zugriff gewähren
   - **Client-Befehle werden ungeprüft an den Daemon weitergeleitet** – Inputvalidierung ist Aufgabe des Nutzers[2,3]

4. Docker workflow
   - Verwende **minimalistische Base-Images** wie alpine oder distroless zur Reduktion der Angriffsfläche
   - Nutze keine veralteten oder inoffiziellen Images – **nur geprüfte Quellen oder selbstgebaute Dockerfiles**
   - Füge keine Secrets (Passwörter, API-Keys) ins Dockerfile ein – verwende Umgebungsvariablen oder Secret-Manager
   - **Scanne Images nach CVEs** mit Tools wie trivy oder docker scan nach dem Build
   - Bevor Images in den Docker Hub gepusht werden, sollten sie **signiert und geprüft** werden (z. B. mit Docker Content Trust)[2]

5. Docker compose
   - Setze sensible Dienste (z. B. Datenbanken) auf **`internal`**, damit sie **nicht von außen erreichbar** sind
   - Nutze **benutzerdefinierte Netzwerke**, um Services gezielt zu trennen oder zu verbinden
   - Vermeide es, Passwörter oder Secrets direkt in der `docker-compose.yml` zu speichern – verwende `.env`-Dateien oder Secret-Manager
   - Aktiviere **Read-only-Volumes** oder setze restriktive Mount-Berechtigungen
   - Baue Images mit `build:` nur aus **vertrauenswürdigem Code** oder nutze geprüfte Versionen mit `image:`[4]

6. Use cases
   - **Isolierte Testumgebungen:** Sicherheitsanalysen, CVE-Scans und Exploit-Tests können risikolos in Containern erfolgen
   - **App-Isolation:** Jede Anwendung läuft in ihrem eigenen Container – Angriffe bleiben meist auf diesen beschränkt
   - **Sicheres Software-Testing:** Verwundbare Versionen (z. B. CVE-Demos) können isoliert und gezielt untersucht werden
   - **Reproduzierbare Security-Umgebungen:** Penetration Tests und CI/CD-Checks laufen immer in identischer Umgebung
   - **Schnelles Patch-Testing:** Sicherheitsupdates lassen sich schnell in neuen Container-Versionen evaluieren[5]

7. Unsere Top 10 Best Practices (an OWASP- und Docker Dokumentation orientiert)

1. Kein Root im Container

Setze USER im Dockerfile oder `--user` beim Start
Root im Container = hohes Risiko bei Exploits

2. Keine Mounts vom Docker-Socket

Niemals `/var/run/docker.sock` in Container geben
→ Sofort Root-Zugriff auf den Host möglich

3. Verwende nur vertrauenswürdige Images

Nur signierte Images nutzen (`Docker Content Trust`)
Vermeide „latest“ oder dubiose Quellen (z. B. `user123/nginx-modified`)

4. Aktiviere seccomp / AppArmor / SELinux

Blockiert gefährliche Systemaufrufe oder Dateiaktionen
Nutze Default-Profile oder eigene Härtung

5. Rootless Docker verwenden

Verhindert, dass der Docker-Daemon Systemrechte hat

→ Noch sicherer, auch wenn Container kompromittiert wird

6. Capabilities minimieren

`--cap-drop=ALL`, dann gezielt `--cap-add`

Verhindert Eskalation durch Kernelfunktionen

7. Read-only Filesystem

`--read-only` + gezielte Schreibpunkte (z. B. `/tmp`)

Erschwert Malware und Manipulation

8. Keine Secrets im Image

Keine Passwörter, API-Keys oder Tokens im Dockerfile

Nutze Umgebungsvariablen, Docker Secrets oder externe Manager

9. Regelmäßige CVE-Scans

Nutze Tools wie `trivy`, `docker scan`, `grype`

Vor allem vor Deployment automatisiert in CI/CD

10. User Namespaces aktivieren

`root` im Container ≠ `root` auf dem Host

Schutz selbst bei Container Breakout

Informationsquellen:

- [1] https://contabo.com/blog/de/container-vs-virtuelle-maschinen/ Tobias Mildenberger, 05.05.2022,
- [2] https://jfrog.com/de/devops-tools/article/understanding-and-building-docker-images/ JFrog 17.03.2017
- [4] https://docker-curriculum.com/ Prakhar Srivastav, 01.06.2025
- [5] https://medium.com/@BeNitinAgarwal/docker-usecases-3b62f4d68bc4 Nitin AGARWAL, 05.01.2017
- [6] https://github.com/vulhub/vulhub
- [7] https://cheatsheetseries.owasp.org/cheatsheets/Docker_Security_Cheat_Sheet.html OWASP Cheat Sheet Series Team: Jim Manico, Jakub Maćkowski und Shlomo Zalman Heigh, 01.06.2025
- [8] https://docs.docker.com/develop/develop-images/dockerfile_best-practices/ Docker Inc., 01.06.2025
- [9] https://docs.docker.com/engine/security/ Docker Inc., 01.06.2025