



Microservices and Container Security

Cybersecurity
Web Development Day 2



Class Objectives

By the end of today's class, we'll be able to:



Understand how microservices and architecture work to deliver more robust, reliable, and repeatable infrastructure as code.



Run a container vulnerability and filter through the results.



Deploy a Docker Compose container set and test the deployment functionality.



Deploy a container IDS and emulate intrusions on the previously deployed container set.

Application Structure

Introduction to Microservices

Today, we'll set up and secure microservices. First, we'll need to understand the following:

- ✓ How an application and its components are organized.
- ✓ How information flows between components of an application.
- ✓ How application architecture has changed from monoliths to microservices.
- ✓ How to deconstruct a monolith into microservices.

Components of a Typical Web App

A developer, Andrew, often uses a **browser to manage a public-facing employee directory application** to retrieve and update different kinds of employee lists from his company.

- The employee directory application is set up so that Andrew can add, remove, and view employees.
- The employee directory application needs to access the following components:



Front-end server

To display webpages and style them in readable formats. Also responsible for receiving and responding to HTTP requests.



Back-end server

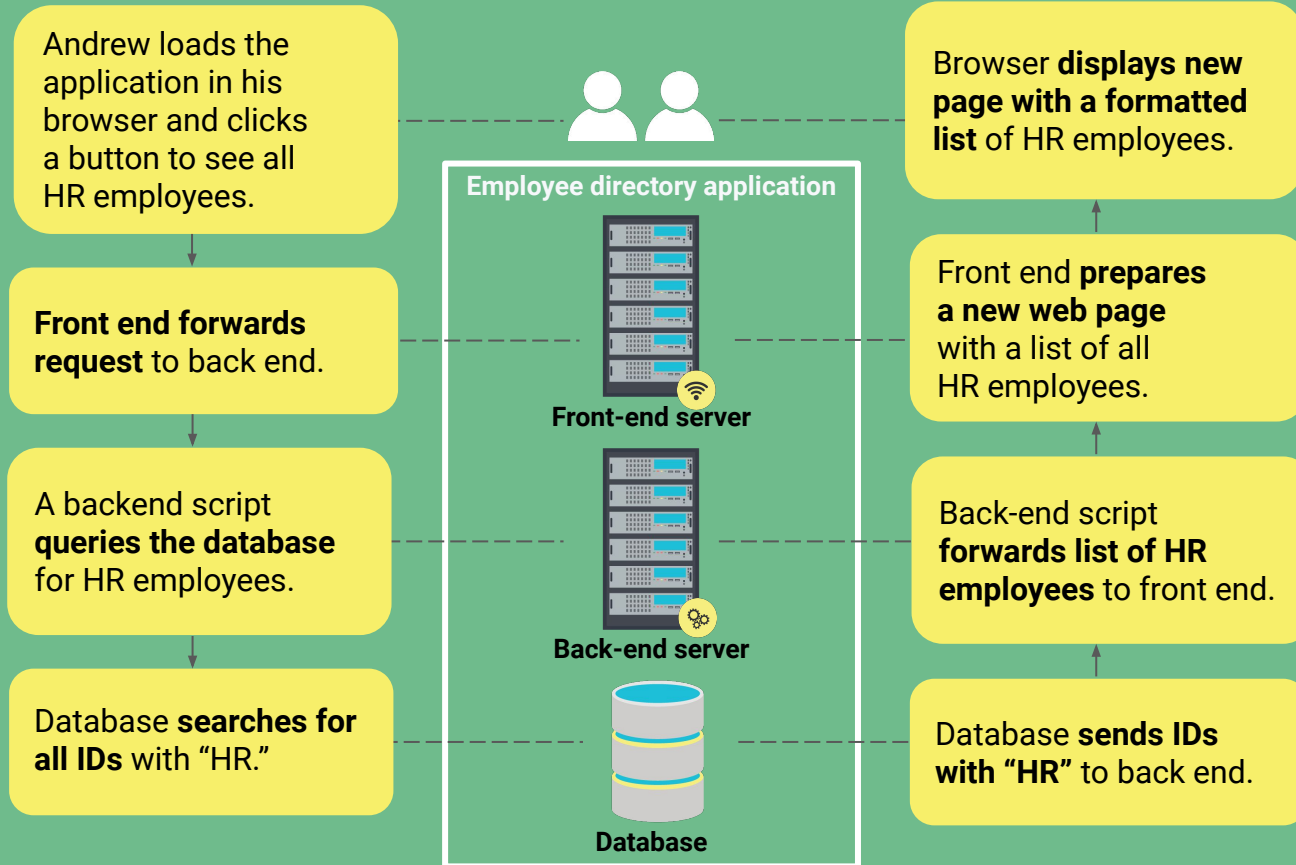
Executes business logic and writes or reads corresponding data to and from a database.



Database

Stores information about employees, such as their employee IDs and names.

Information Flows between Application Components

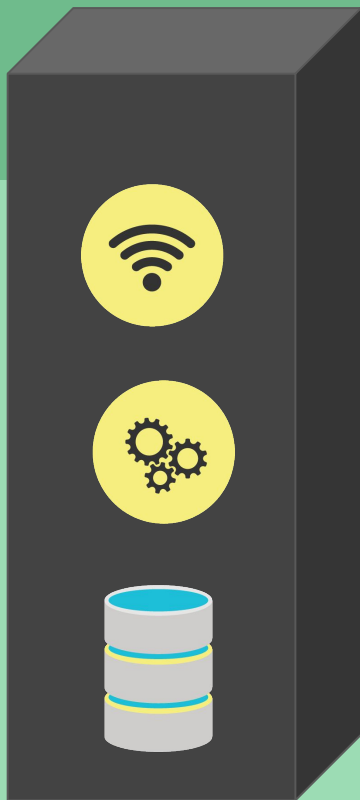


Monolith to Microservices



Understanding the components of an application is key to understanding the modern architectural paradigm of microservices.

First, we need to understand the original architecture type, the monolith, and the issues that led to the adoption of microservices.

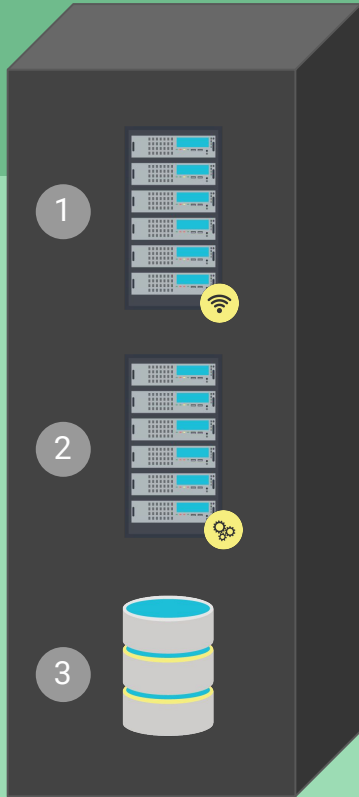


Monolith

A monolith is a **singular machine** that **hosts all the components required to serve a website or application.**

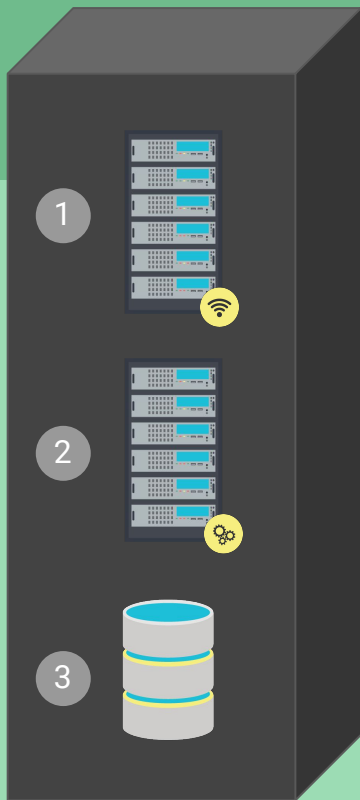
In other words, a monolith is a machine that has a front-end server, back-end server, and a database.

Monolith



Suppose Amazon is using a monolithic server. It would contain:

- 1 **Front-end HTML server:** GUI for customers to look at while shopping.
- 2 **Back-end MySQL server:** Back-end server showing the inventory and stock.
- 3 **MySQL database:** Database of customers, and their information and purchases.



Monolith

The components of a singular machine are highly dependent on each other. If any component malfunctions, the entire application will malfunction.

- **Hackers, environmental issues, and human errors** can threaten the entire machine.
- **Updating a single component means taking down all components**, resulting in downtime and lack of availability.

Problems with Monoliths

It's now expected that companies will ensure availability by maintaining almost 100% uptime. The risk associated with a interdependent monolithic system presented problems.



- Separating an application's components into their own machines (**microservices**) means the sysadmin can update the front end, reboot it, and only have to verify the front-end components, not the back-end components and database.
- If a component is compromised by a hacker, the potential damage is restricted to only that component.

Microservices

Each independent machine is a component that executes one primary function or **service**.



We've already implemented microservices in our cloud security week:

- A **jump box** to connect to a private Ansible server.
- An **Ansible server** to configure DVWA servers.
- **DVWA servers** to provide the DVWA service.
- A **Docker image** to provide WordPress.

Benefits of Microservices



Scalability and resilience: Replication of components lets you serve more clients and provides identical backup components if one fails.



Rapid response: Since microservice components are inherently smaller, they can be replaced and updated quickly.



Isolated improvement: Since microservices serve one primary function, they can be developed with the goal of optimizing their functionality.



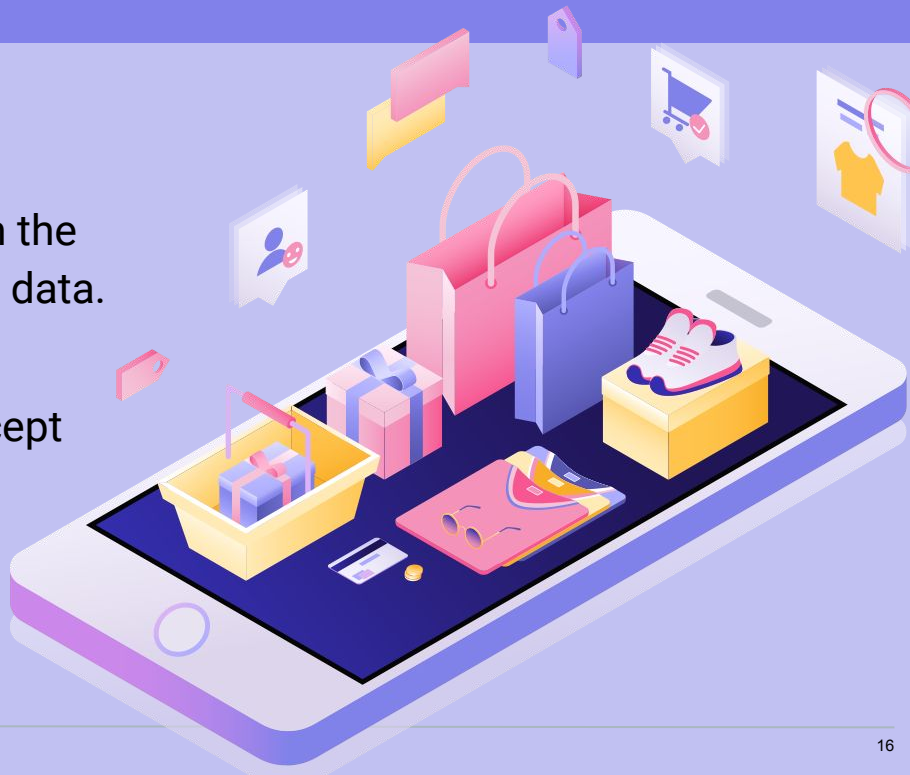
Isolated security: One compromised component does not equal a compromised application.

APIs

APIs

When Amazon customers want to retrieve a list of all the newest and most popular books, they do not go to the back-end MySQL server to check the inventory.

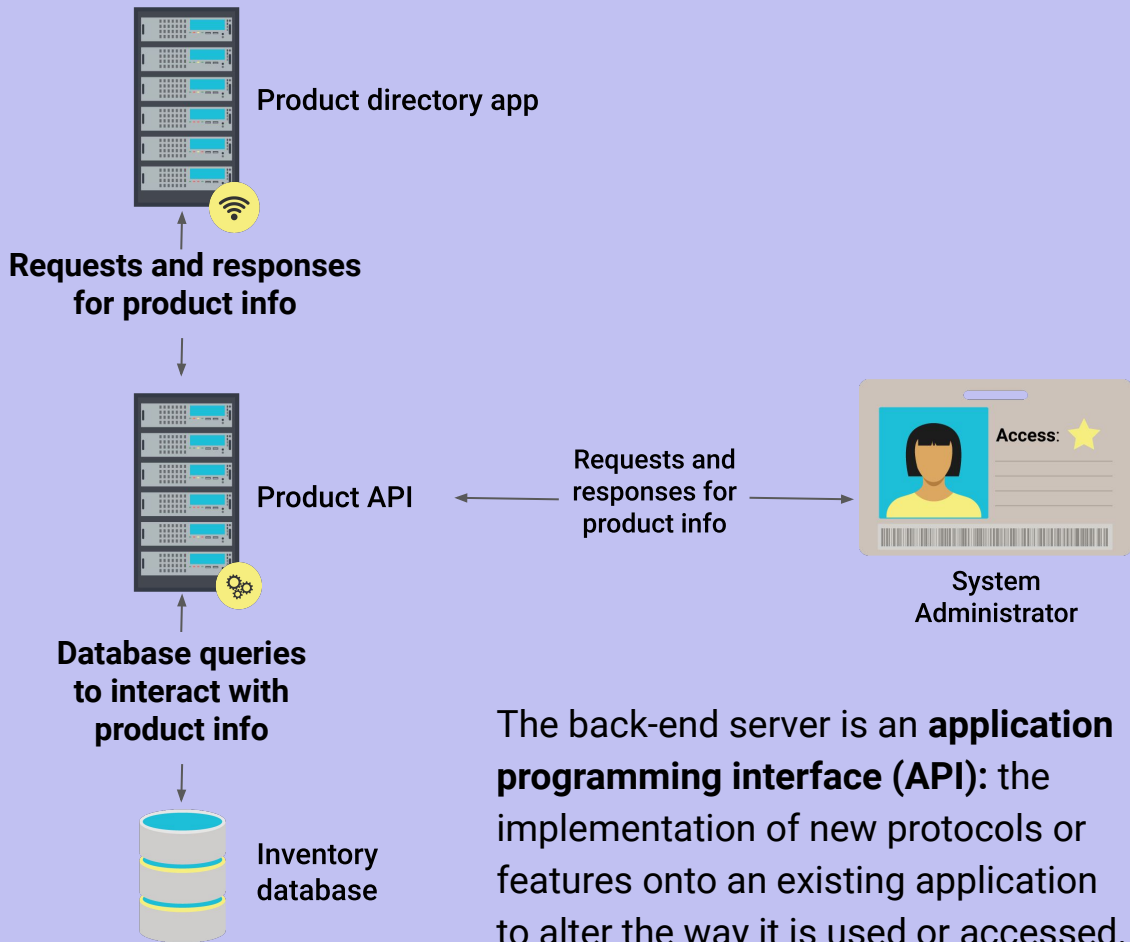
- They go to the front-end microservice.
- Then, the microservice communicates with the back-end server, and brings up the relevant data.
- Customers do not have access to the back-end server. Backend does not accept HTTP requests, meaning users can't communicate with it.



APIs

Systems administrators need to access and query the back end to update product information.

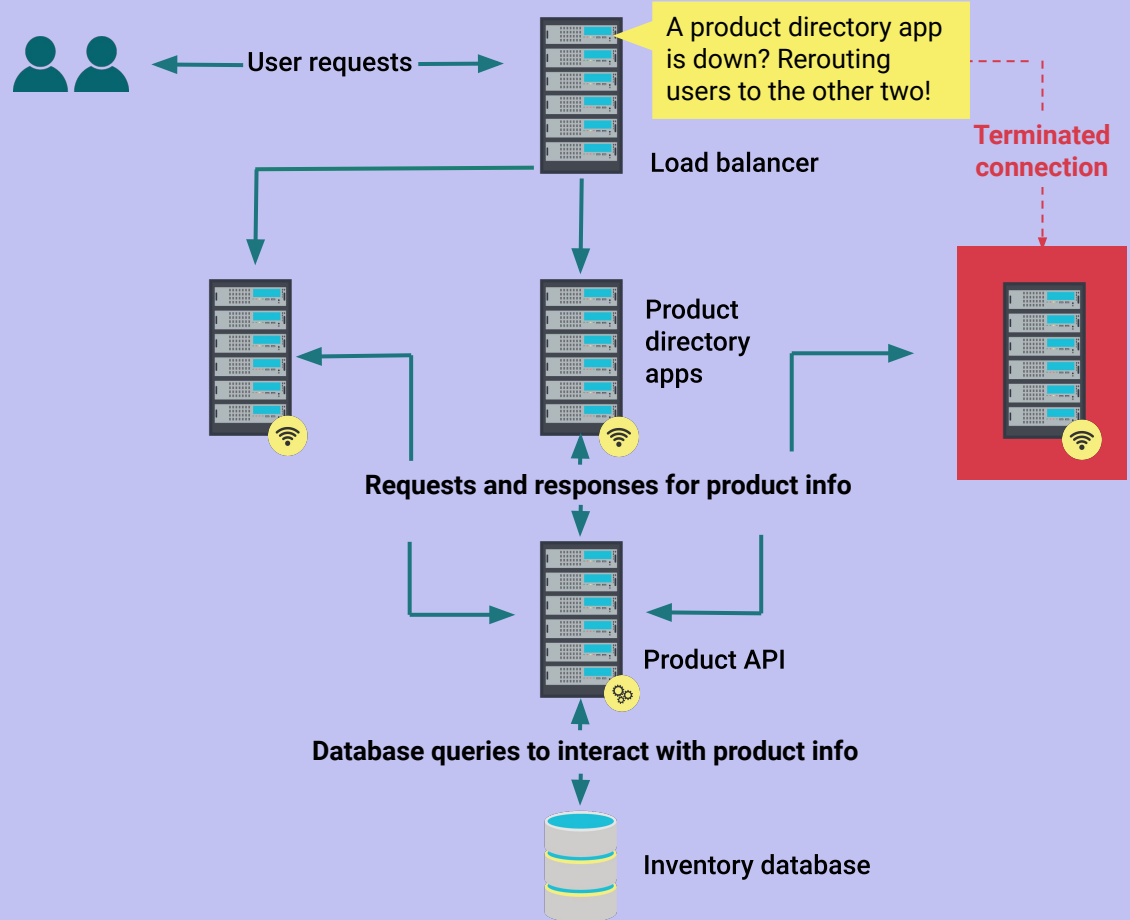
- Since they need to directly communicate with the back end, they must modify it to accept requests.
- Software developers will program the back end to receive, from sources other than the front-end, HTTP requests to add, remove, and view product information.



Challenges of Microservices

Microservice complexity and interconnectivity also come with unique security issues.

- Microservices have **increased complexity and require more maintenance** as the application and number of components grows.



Monolith to Microservice

In the next activity, you will diagram microservices.

01

Separate each component of the monolith by its function and move into its own machine.

02

Add communication between each part of the microservice.

03

Turn the back-end server into an API to interact with more than just the front end.

04

Rename the rest of the component services to match the main function they provide.



Activity: Monolith to Microservice

In this activity, you will diagram a microservice version of a monolith web store application.

Suggested Time:
15 Minutes





Time's Up! Let's Review.

Container Vulnerability Analysis



In this section, we'll explore the underlying technologies of microservice machines and their associated vulnerabilities.

Microservice Infrastructure

Microservices require lightweight environments to run because:



Live services need to be deployed quickly.



Multiple copies of a service can be replicated as needed to meet demand.



Developers and maintainers can deploy their own copies of these services locally for testing purposes.



Full-sized VMs for each service require more resources and are more expensive.

Each service needs
its own lightweight
virtual environment.

**Can we think of any
technologies that provide a
lightweight, isolated environment?**

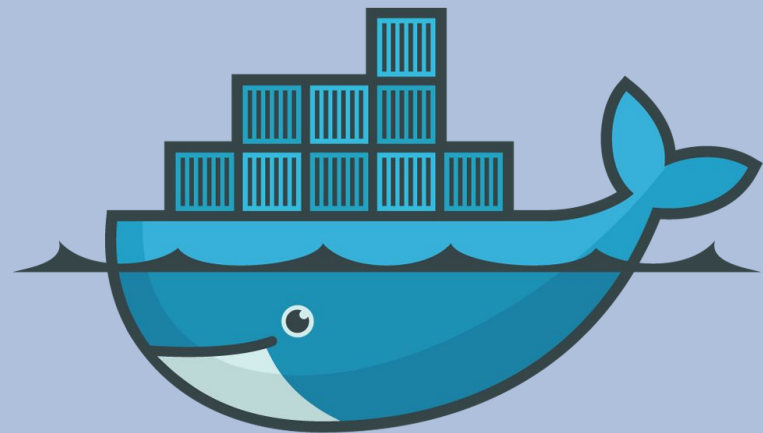


Microservice Infrastructure with Containers

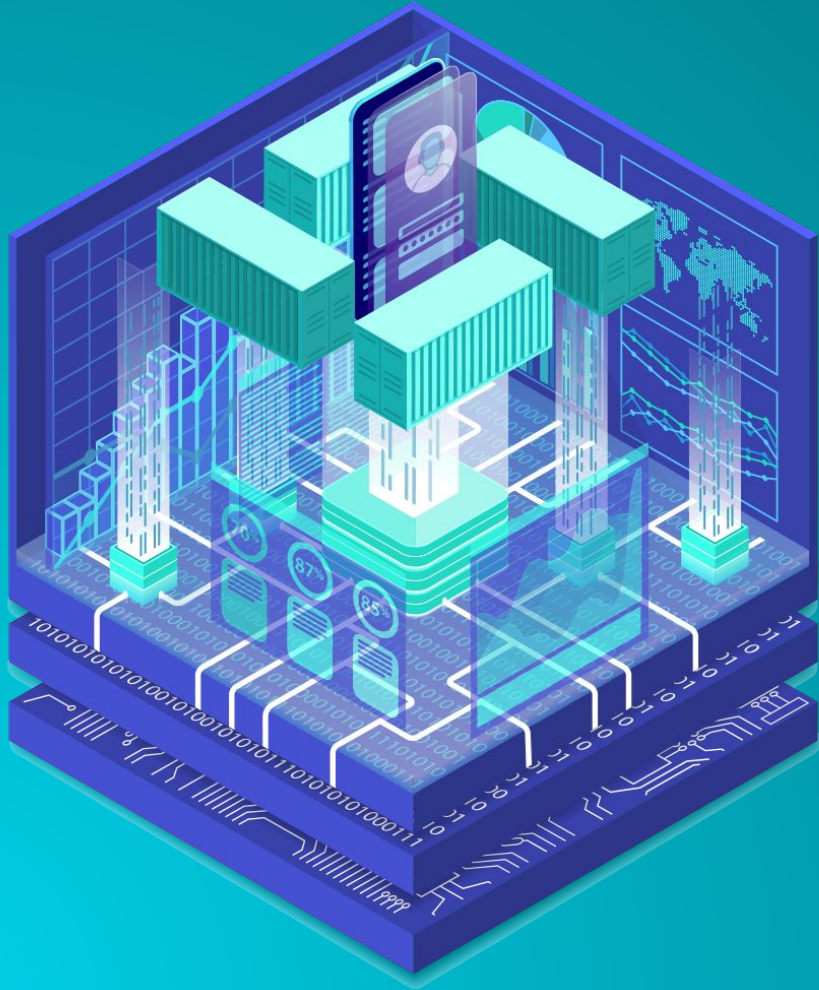
In the cloud security unit, we covered containers and Docker extensively. We also used Ansible to set up lightweight containers in our activities.

- Containers run as isolated, virtual operating systems that dynamically allocate resources depending on needs. This is unlike regular virtual machines, where all hardware is virtualized.
- Docker is the most popular container platform.

For the rest of this class, we will be referring to Docker's implementations of containers.




docker




Containerization is the process of packaging all the requirements for a microservice into a container.

How a Container Becomes a Microservice


Containerizing a microservice requires the following steps:



Declare a base operating system for the microservice to run on.



Copy the microservice's source code to the container.



Set a command that launches the microservice.

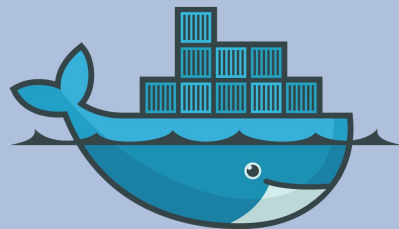
Dockerfile

Containerization can be declared in a text file called a **Dockerfile**.

```
#####  
# Employee Directory Application  
#####  
# Required Base OS  
FROM ubuntu:14.04  
...[truncated]
```

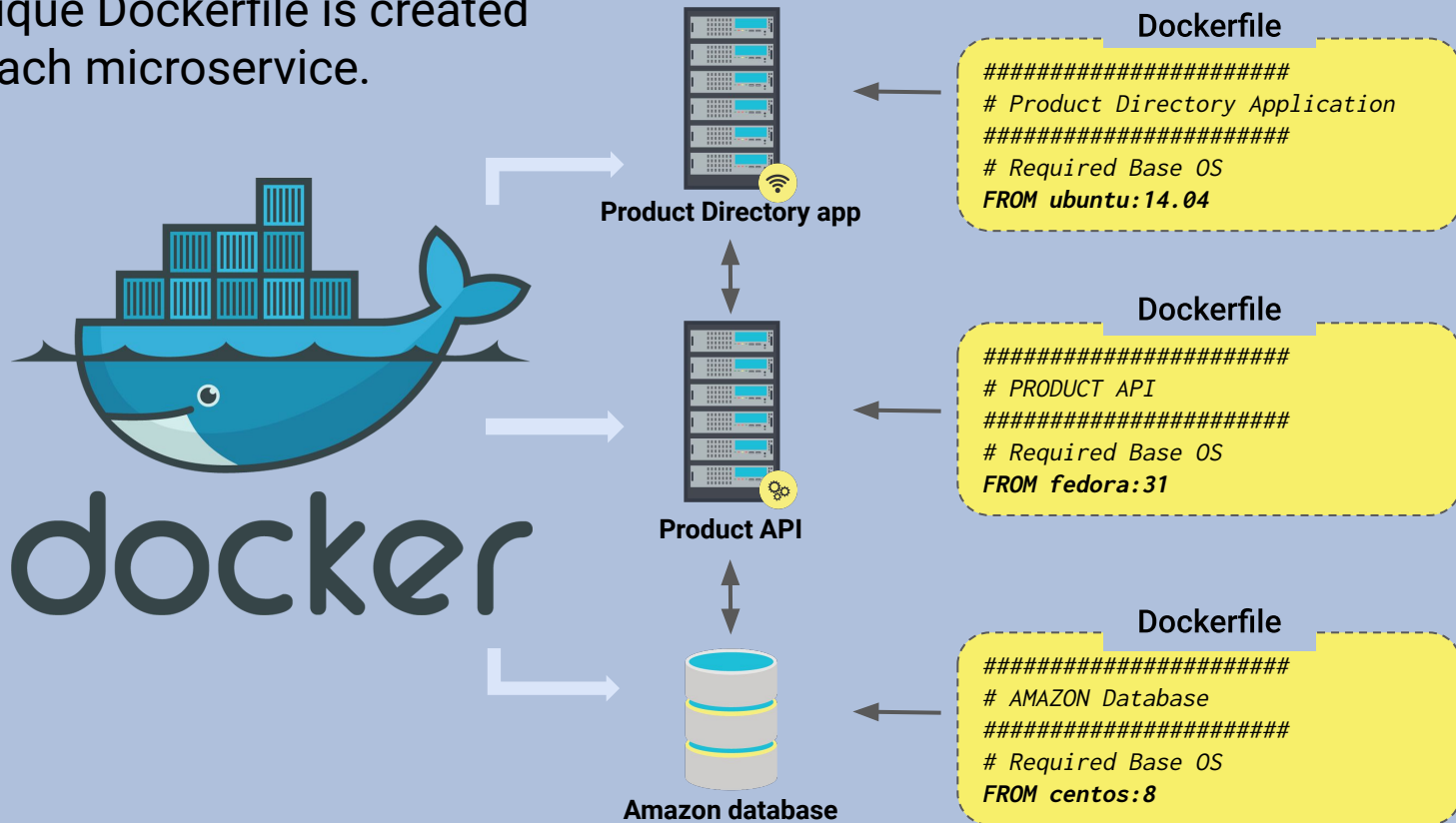
Dockerfiles contain all the configuration needed for a container in one file.

This is similar to an Ansible configuration file, which contains everything needed to configure a host with the command **ansible-playbook your-playbook.yml**.



Dockerfile

A unique Dockerfile is created for each microservice.



Dockerfiles contain all the configurations needed to set up a container, but we also need to ensure the configurations do not come with security risks.



Dockerfile

Security issues are inherited from the underlying operating system:

- In this Dockerfile, Ubuntu 14.04 is the base operating system of the application.
- All security issues present in Ubuntu version 14.04 will also exist in this container.
- We will use a tool called Trivy to scan Dockerfiles and Docker images to create a list of known security issues.

```
#####  
# Employee Directory Application  
#####  
# Required Base OS  
FROM ubuntu:14.04  
...[truncated]
```


Vulnerability Scanning with Trivy

Some vulnerabilities we will look at are:



Man-in-the-middle (MITM): An attacker secretly intercepts and relays requests between a victim and their intended target, such as a website.



Buffer overflow: A software anomaly in which the memory of an application is corrupted and foreign code is executed as a part of memory.



Denial of service (DoS): Takes down a service and makes it unavailable.



Privilege escalation: Enables one to escape the regular user space of an operating system to run commands with elevated privileges



Scanning container images and Dockerfiles before they are deployed is called **vulnerability scanning**.

It is one of the earliest stages in the application deployment process that security engineers become involved.



Instructor Demonstration

Trivy

Bash Filtering

Now that we know how to use Trivy, we'll filter the results using bash command-line tools.

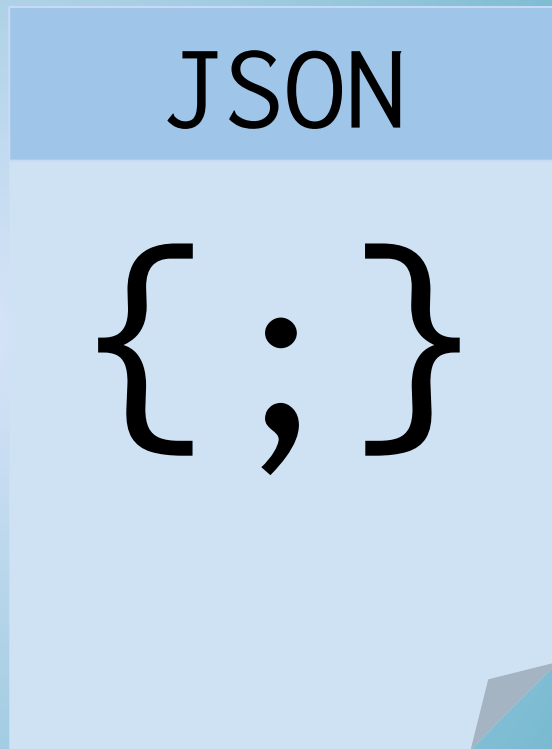
We will need to use the command-line tool **jq** to inspect the **results.json** file that we created with Trivy.

- **jq** is like **sed**, but for JSON data. It's used to filter through JSON files and has the same text editing functionalities built into **sed**, **awk**, and **grep**.
- JSON is a structured data format that is used often in software development languages, logging, and security.

JSON

While we won't need to know JSON in depth, we'll need to know how to interact with it using `jq`, to filter through vulnerability reports.

JavaScript Object Notation is a language-agnostic data format used for web services and APIs, as well as log aggregators such as ELK and Splunk.



JSON

A high level overview of JSON formatting:

```
{  
  "thisIsAKey": "this is a value",  
  "theFollowingIs": "an_array_example",  
  "ciaTriad": [ "confidentiality", "integrity", "availability" ]  
}
```

- It uses **key:value** pairs.
- Data is separated by commas.
- Curly braces can hold objects.
- Square brackets hold arrays.

JSON

A high-level overview of **jq** syntax. This searches through the **Vulnerabilities** in the first array (or listed data) of the **results.json** file.

```
jq '.[0].Vulnerabilities' results.json
```

- **VulnerabilityID**: The Common Vulnerabilities and Exposures ID (CVE), a unique ID given to every vulnerability.
- **PkgName**: contains the package name that is vulnerable.
- **InstallVersion**: Version of the vulnerable package.
- **FixedVersion**: Fixed version of the package.
- **Title**: Title of the vulnerability.
- **Description**: Description of the vulnerability.
- **Severity**: The level of potential risk to the system it is installed on.
- **References**: Links to more information.

JSON

We can use pipes with our **jq** command to filter for specific results.

```
jq '[0].Vulnerabilities[] | select(.Description | test("Man-in-The-Middle"))'  
    results.json > results_mitm_vulns.json
```

- This command will search the for the phrase “Man-in-the-Middle” in the description section of the vulnerabilities list of the **result.json** file.
- It will then send those results to a new file named **results_mitm_vulns.json**.



Activity: Container Vulnerability Analysis with Trivy and Bash

In this activity, you will use Trivy to scan a WordPress image, generate a comprehensive list of the most severe vulnerabilities, and create a report of your findings.

Suggested Time:
20 Minutes






Time's Up! Let's Review.

Deploying and Testing a Container Set


Deploying at Scale

Deploying containers can be tedious, especially if a business requires hundred or thousands of microservices. Therefore we'll need to know how to deploy in large-scale environments. Specifically, how to:




Deploy multiple types of containers at the same time.

For example, front-end and database services.



Deploy copies of containers we want replicated.



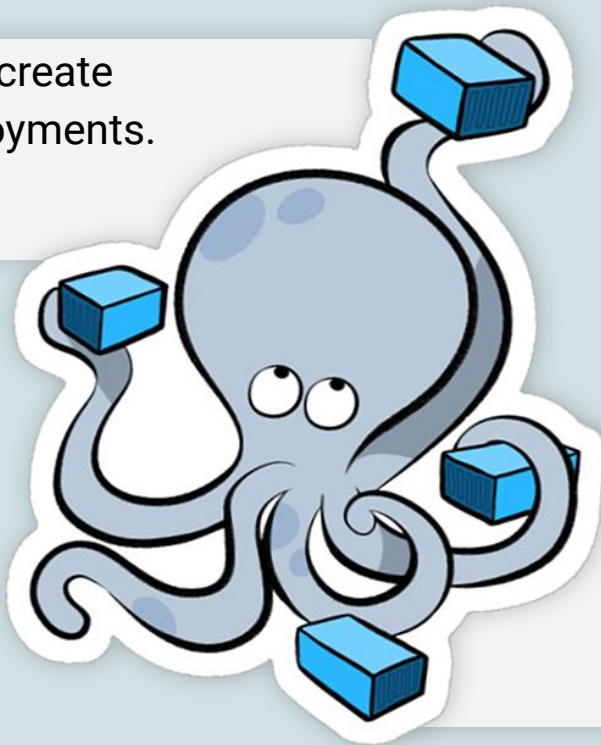
Set up a network for our containers so they can communicate with each other.

Deploying at Scale

We'll use Docker Compose to enable these large-scale deployments.

Docker Compose allows us to create repeated, multi-container deployments.

docker
compose



Remember: We used Docker Compose in the last class to set up our WordPress site.

Docker Compose YAML File

We also used YAML files to deploy Ansible playbooks during our cloud security unit.

- Docker Compose uses YAML to define the containers, their networking configurations, and where you want to copy files from your host machine into your container.

```
version: "3.7"

services:
  ui:
    image: httpd:2.4
    ports:
      - 10000:8080
    volumes:
      - ./volume:/home
    networks:
      demo-net:
        ipv4_address: 192.168.1.2

[truncated...]
```



Instructor Demonstration

Docker Compose

Docker Compose Demo

In this demonstration, we will deploy multiple containers at the same time.

01

Examine the YAML file for its current configurations.

02

Launch our services with `sudo docker-compose up`.

03

Use our browser to verify the our front-end user interface (ui) services are deployed properly.

04

Test the deployment of our database (db) service.

05


Change the configuration of our UI service to display through a different port.

06

Test the new UI service's functionality with our browser.


```
version: "3.7"

services:
  ui:
    image: httpd:2.4
    ports:
      - 10000:8080
    volumes:
      - ./volume:/home
    networks:
      demo-net:
        ipv4_address: 192.168.1.2
  db:
    image: mariadb:10.5.1
    restart: always
    environment:
      MYSQL_DATABASE: demodb
      MYSQL_USER: demouser
      MYSQL_PASSWORD: demopass
      MYSQL_RANDOM_ROOT_PASSWORD: "1"
    volumes:
      - db:/var/lib/mysql
    networks:
      demo-net:
        ipv4_address: 192.168.1.3
networks:
  demo-net:
    ipam:
      driver: default
      config:
        - subnet: "192.168.1.0/24"
volumes:
  ui:
  db:
```



This is a
simple YAML file.
Let's break
it down.

`services:`

`ui:`

`db:`

We'll break
down the file by the
services it installs,
the user interface,
and the database.

The `ui` and `db`
are services.

ui:

image: httpd:2.4

ports:

- 10001:8080

volumes:

- ./volume:/home

networks:

demo-net:

ipv4_address: 192.168.1.2

image: Points to where Docker Compose will retrieve the container image on Dockerhub, which is Apache's httpd container.

Ports: Ports this container will run on.

volumes: Local destination where the Apache server will save configuration files.

networks: The network that this service will connect to.

Ipv4_address: The static IP address we are assigning to this container.



Instructor Demonstration

Standing up Scaled Services

db:

image: mariadb:10.5.1

environment:

MYSQL_DATABASE: demodb

MYSQL_USER: demouser

MYSQL_PASSWORD: demopass

MYSQL_RANDOM_ROOT_PASSWORD:

"1"

volumes:

- **db:/var/lib/mysql**

networks:

demo-net:

ipv4_address: 192.168.1.3

image: The container image and version we'll be using. (MariaDB database version 10.5.1)

environment: Containers within the MySQL server.

MYSQL_DATABASE: The name of the database.

MYSQL_USER: The default user for the database.

MYSQL_PASSWORD: The password for the user.

MYSQL_RANDOM_ROOT_PASSWORD: Gives the root user a random password, for security purposes.

volumes: The location where we are saving our configuration files.

networks: Assigns this database a static IP address using the same network (demo-net).



Instructor Demonstration

Logging into MariaDB



Activity: Deploying and Testing a Container Set

In this activity, you will deploy multiple containers with Docker Compose and test functionality.

Suggested Time:





Time's Up! Let's Review.

Container Runtime IDS



Now, we will combine
our deployed container set
with a **container intrusion
detection system (CIDS)**.

IDS Review

We covered IDS in our network security unit:

An intrusion is any unwanted malicious activity within a network or a system.

An IDS is a device or software application that monitors a network or system for malicious activity.

A host-based intrusion detection system (HIDS) detects intrusion attempts on an endpoint device.

A network intrusion detection system (NIDS) detects intrusion attempts within a network.

Container Intrusion Detection Systems (CIDS)

Containers have the same vulnerabilities as underlying VMs. But the large-scale deployment of containers vastly multiplies their attack surface.



Even though we scanned container Dockerfiles for vulnerabilities, we haven't done anything to secure the containers once they are deployed.



After container sets are deployed with tools like Docker Compose in live environments, we need to be able to detect intrusions.



Similar to NIDS and HIDS, container intrusion detection systems (CIDS) detect intrusions specifically for deployed containers.



The specific CIDS we will use today is called Falco.



Falco

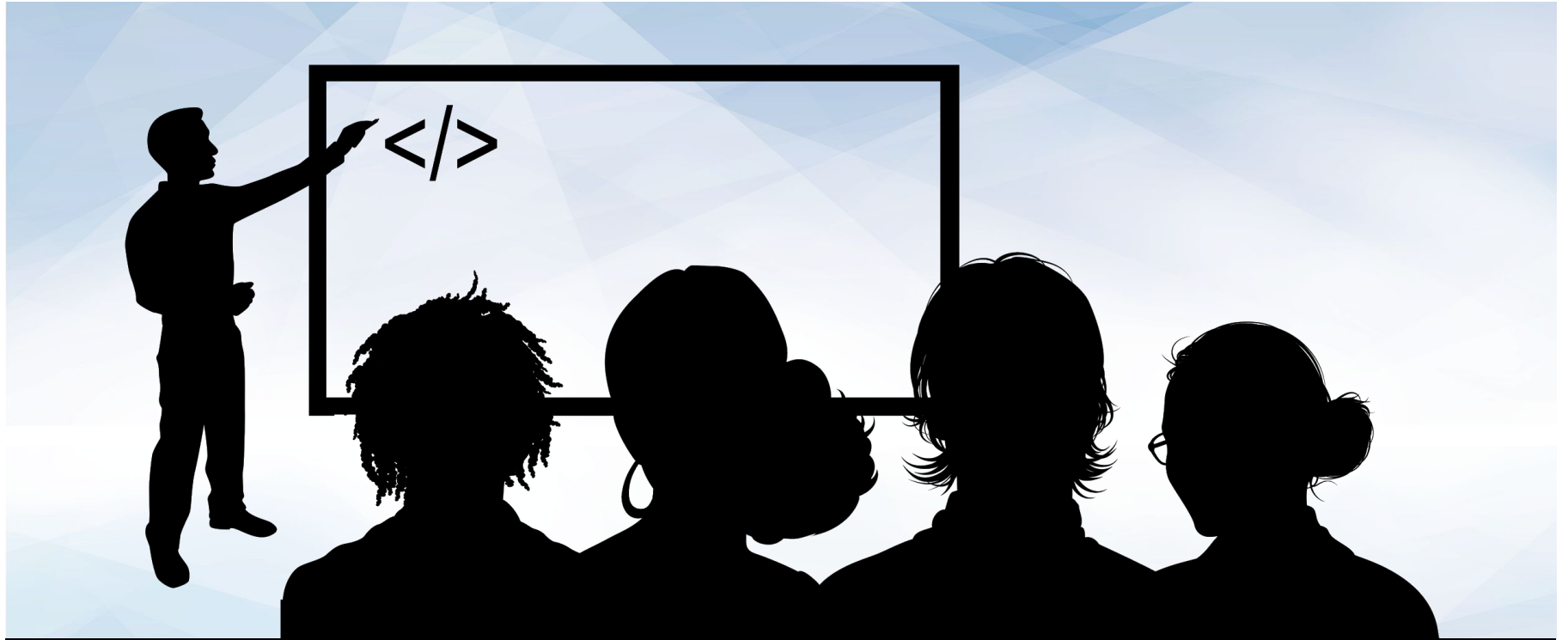
Falco is an open-source CIDS that alerts security professionals to potential intrusion attempts.

Common intrusion behavior on containers include:

- **New shell running inside a container.** An unknown shell starting in a container signals an attacker has potentially accessed a system and will start executing commands.
- **When a sensitive file, such as `/etc/shadow`, is read.** At no point during a container deployment should the contents of `/etc/shadow` be read. This is a clear sign that an attacker is retrieving the hashed passwords of accounts within the system.
- **Unprompted configuration changes.** After an attacker gains access to a system, they will often create a user to maintain persistence within that system.
- **File creations at `/root` or `/`.** At no point during a container's runtime should new files be created in the `/root` or `/` directories. Files created in these directories indicate compromise.



In the upcoming demo, we will use Docker Compose deployment in conjunction with Falco to simulate an intrusion attempt and monitor the results.



Instructor Demonstration

CIDS



Activity: Container Intrusion Detection System

In this activity, you will set up and test Falco on the WordPress container.

Suggested Time:
20 Minutes





Time's Up! Let's Review.

Next Week

In Unit 15, we will use the Web Vulnerabilities Azure Lab environment.

Inside of the Web Vulnerabilities environment are two nested VMs: a Kali Linux instance and a machine called owaspbwa.

- The owaspbwa machine hosts the vulnerable web applications that we will use throughout the class, but we do not log into it.
- Instead, we will perform all of our work on the Kali Linux instance.

Student credentials for the Kali machine:

- Username: `sysadmin`
- Password: `cybersecurity`

Take a moment to set up and access this environment. Troubleshoot any issues before class next week.

*The
End*