# Overcome The Security Hurdle In Your Docker Environment

Tamar Twena-Stern

# Tamar Twena-Stern

- Software Engineer - manager and architect
- Working at **XM Cyber**
- Was a CTO of my own startup
- Passionate about Node.js !
- Twitter: **@SternTwena**

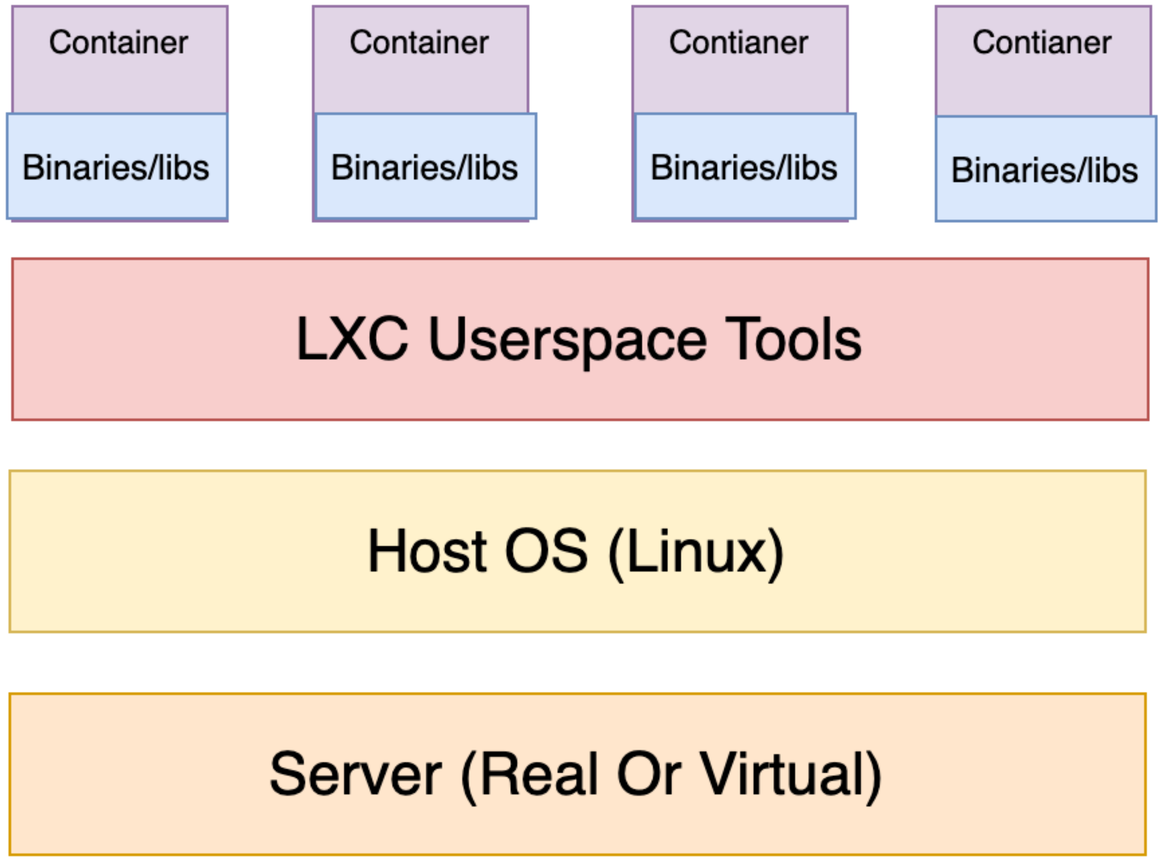
# Tamar Twena-Stern

- Just Finished My Maternity Leave
- Have 3 kids
- Loves to play my violin
- Javascript Israel community leader

# Agenda

- Quick Introduction to Docker
- Docker resource management and how they can be used to attack your Docker environment
- Defend your Docker Secrets
- Container privileges
- Reliable images

# Intro To Linux Containers

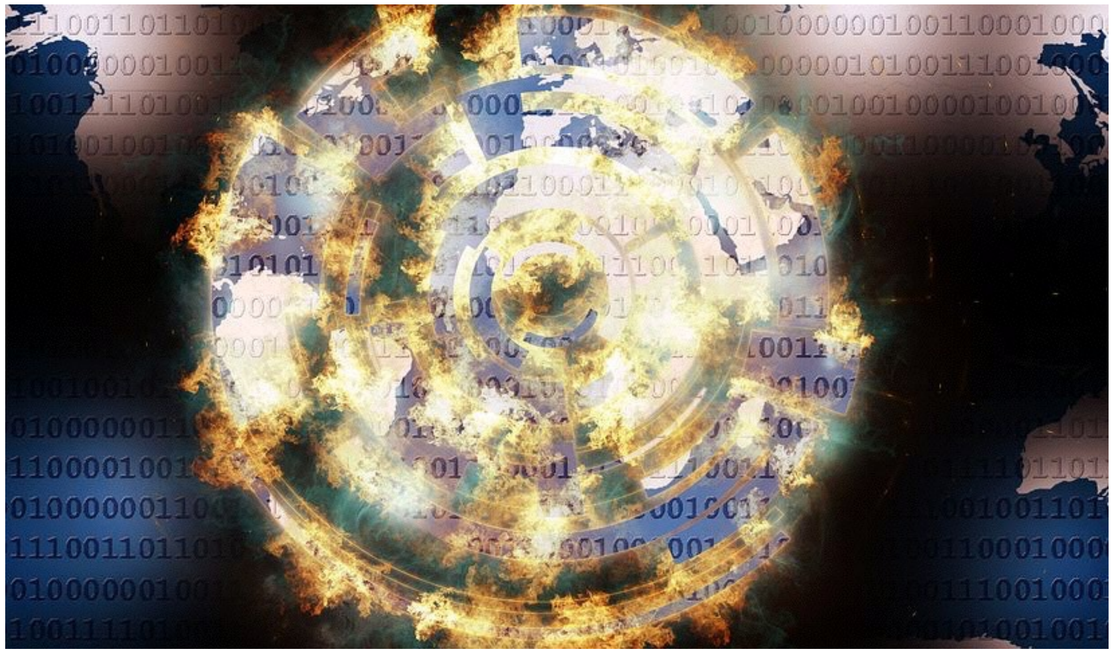


# What Is Docker ?

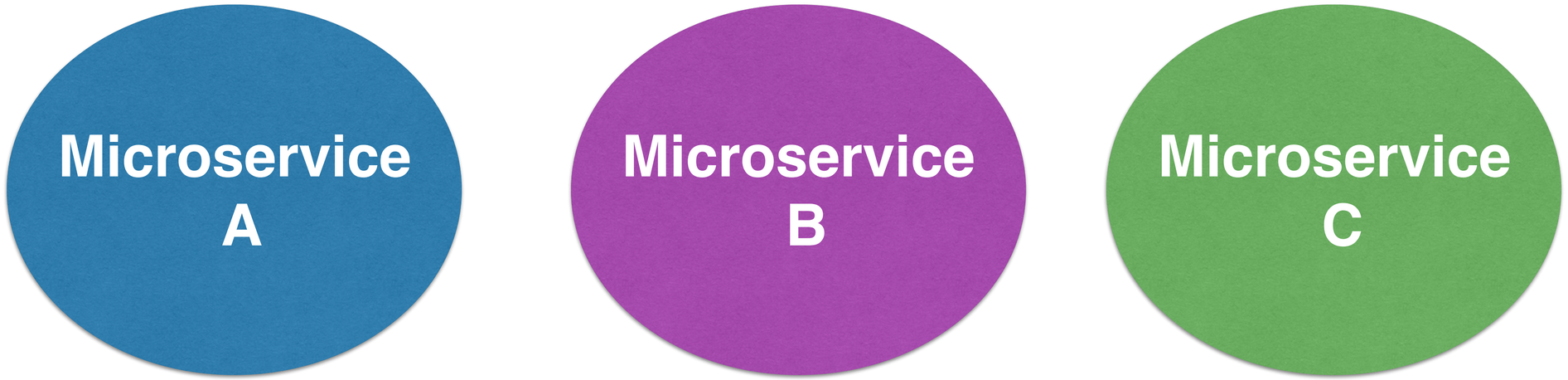

- Popular open source project based on linux containers
- Docker is a container engine that uses Linux Kernel features to
  - Create containers on top of operating system
  - Automate application deployment to container
- Provide lightweight environment to run your application code.

# Demo - A Simple Dockerized Server

# Denial Of Service Attack



# Docker Microservices Environment

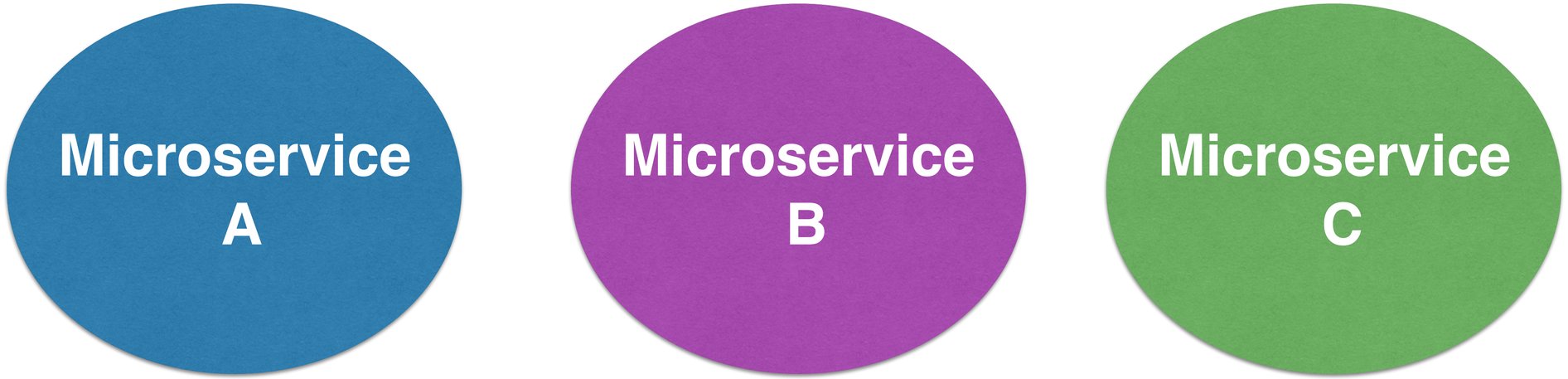


# Now, Lets Bombard The Dockerized Service With Some Load

# Docker CPU Access

- By default, each container's access to the host machine's CPU cycles and memory is **unlimited**.
- One container can consume the entire CPU or memory of the machine it is Running on

# When One Container Consuming All CPU Resources



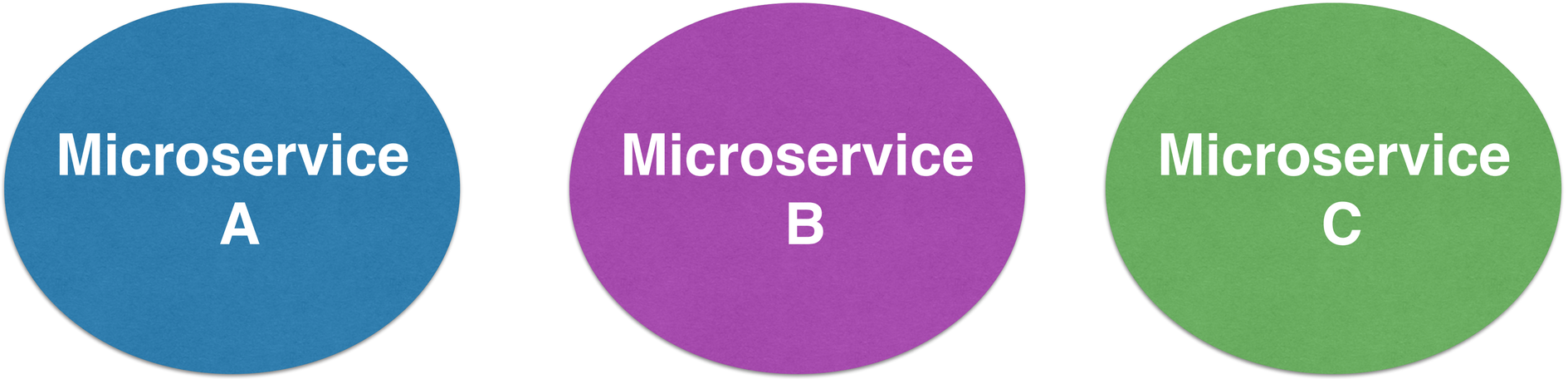
# Denial Of Service

- Container can consume the whole CPU or memory of the host machine.
  - For example : When no available memory , linux kernel will throw out of memory exception and kill other processes
- Whole system can crash
- Attackers will use this knowledge to bring down your apps down
- All the containers can crash

# Mitigation - Restrict Resources For Your Containers

- Limit CPU and Memory on all of your containers
- A container that runs out of resources will shut down.
- Isolation protects all of your other containers to shut down

# When limiting resources



# Mitigation : Use Docker Flags On Container Run To Restrict CPU And Memory

- From the command line use the following flags :
  - -m to restrict memory
  - -cpu to determine how much cpu your container will use

**docker run -p 49160:8080 -d tamatwe/unlimited_server_cpu -m 0.5 -cpu 0.5**

# Mitigation : Restrict CPU And Memory In Docker Compose File

```
version: "3.7"
services:
  redis:
    image: redis:alpine
    deploy:
      resources:
        limits:
          cpus: '0.50'
          memory: 50M
        reservations:
          cpus: '0.25'
          memory: 20M
```

# Defend Your Secrets

# Secrets As Files In The Image

- A lot of time we are using secrets inside our applications
- We usually store the secrets in files
  - Password
  - SSL certificate
  - SSH private key
  - TLS certificates and keys
- When we build a Dockerfile , in that case , by using COPY or ADD we copy the requested secrets into our docker image.

# Demo - Getting A Secret File From A Docker Image

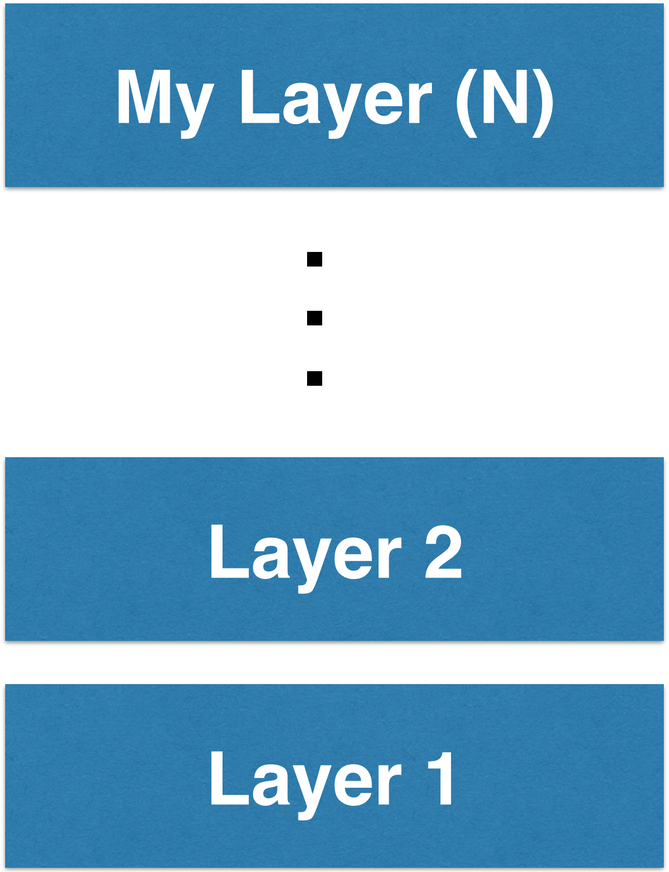
# Never Store A Secret In A File Inside Your Container

- 2 easy steps to get your secret :
  - Pull the image
  - Run Exec to get the file from container

# Can I Delete The File After Copy ?

```
COPY my_secret.txt .



// Do logic with your secret



RUN rm -rf my_secret.txt
```

# Docker Layering Model



# Docker Caching Layer

- To Optimise build process - docker use caching
- Caching layer works on RUN, COPY and ADD commands

**Warning - Even if deleted, file can be fetched from caching layer !**

# Demo - Should We Store The Secret In Env Variable ?

# Never Store A Secret In An Env Variable

- 2 easy steps to get your secret :
  - Pull the image
  - Run docker **INSPECT** and get all info about env variables

# So, How Do I Defend My Secrets ?

# Mitigation - Use Multi Staged Builds

- Use Multiple FROM statements in your Dockerfile
- Each can use a different base
- Each begins new stage of the build
- Fetch and manage secrets in an intermediate image layer that is later disposed of so that no sensitive data
  - reaches the image build
  - Held in cache

# Mitigation - Multi Staged Build

**FROM**: ubuntu as intermediate
**WORKDIR** /app
**COPY** secret/key /tmp/
**RUN** scp -i /tmp/key build@acme/files .

**FROM** ubuntu
**WORKDIR** /app
**COPY** --from intermediate /app .

# Mitigation 2 - Use Docker Secrets

- Docker secrets are available only when using Docker Swarm, or when using docker compose.
- Docker secret is stored as a blob of data
- Use Docker secrets to centrally manage this data and securely
- A secret is only accessible to those services which have been granted explicit access to it, and only while those service tasks are running

# Docker Secrets - Example

```yaml
services:

  my_service:
    image: centos:7
    entrypoint: "cat /run/secrets/my_secret"
    secrets:
      - my_secret

secrets:
  my_secret:
    file: ./super_duper_secret.txt
```

# Limit Container Privileges

# Default Docker Container Privileges - Running As Root

- By default, a docker container is running as root.
- It is easier to the attacker to gain access to sensitive information and to your kernel.

# Lets Understand Why Running As Root Is Not Ideal By Using Volumes To Store Sensitive Data

# Intro To Docker Volumes

- Gives the ability to share data between containers and the host machine
- Can be defined by :
  - -v flag on run command
  - docker-compose
- Volumes are directories that are
  - Outside the default union files
  - Exist as normal directories and files in the host file system

# Docker Volumes For Storing Sensitive Data

- Many recommend to use Docker Volumes to store sensitive data
- **Pro** -
  - This is helping by making them not visible in docker inspect command
- **Cons** -
  - If they are stored in Volumes - by default , when Docker container runs as root, those secrets can be accessed
  - By accessing the volumes, files from the host machine can be reachable too

# Demo - Expose /etc/shadow Of The Host  By Accessing Volume On A Container Runs As Root

# Mitigation - Setting Container User By Using Docker Flags

- Use -u flag to specify user :
  - docker run -u 1000 <IMAGE_PARAMS>
  - in linux 0-499 are reserved users. Use a user above 500 to avoid running as system user.

# Mitigation 2 - Setting Container User In Dockerfile

```
FROM alpine:latest
RUN apk update && apk add --no-cache git
USER 1000
```

# Mitigation 3 - Limit Docker Capabilities

- **—cap-drop** - Drop Docker container capabilities
- **—cap-add** - Add Docker container capabilities
- Don’t use **—privileged -**
  - **Give all linux kernel capabilities to the container**

# Use Reliable Images

# Using A Docker Image

- Docker layering model makes it such that images are built in layers
- Each image has several parents that it takes its functionality from them
- You always base your image on other image that you pulled from Docker hub
- You can pull an image that has vulnerabilities, exploits and other malicious components.

# Mitigation - Disable Pulling Unsigned Images By Using Docker Content Trust

# Docker Images Usages - Guidelines

- Use Only Images From trusted sources
- Use minimal Images - avoid any unnecessary additions
- Always keep your docker images up to date

# Questions ?

- Twitter: **@SternTwena**
- **Mail : tamar.twena@gmail.com**
- **Up next :**
  - **NodeTLV - 3.3.2019**
  - **IJS London - 20.4.2019**