Security: Development to production with Docker containers

1 Purpose

Review how Predictive Medicine addresses some systemic security vulnerabilities when building Docker images and while running Docker containerized microservices.

2 Building Docker Images

2.1 Definitions

2.1.1 Docker Image

- A unit of standardized, packaged software for development and deployment
- Similar to a virtual machine image
- Has layers similar to virtual machines images with base images
- Runnable as a Docker container
- Built from a Dockerfile that specifies included software packages

```
#    Start with one of our base images...
FROM quay.io/pennsignals/alpine-3.7-python-3.6-engineering
MAINTAINER pennsignals
WORKDIR /tmp
#    Copy our microservice code into our new image
COPY microservice ./microservice
COPY requirements.txt .
COPY setup.py .
#    Install/compiles the microservice inside our new image
RUN pip install --requirement requirements.txt
```

- Examples in docker image repositories:
    - Sandboxes as a vm alternative for exploration: hortonworks data platform at 20 GB
    - Isolated, disposable databases for development: postgres at 25-83 MB, mongo at 137 MB
    - Other services for development: node at 25-334 MB
    - Base image for microservices: pennsignals at 23 MB

2.1.2 Docker Container

- Similar to a virtual machine with lower isolation and lower resource requirements
- Adds a final layer to a docker image that holds runtime
- Suspendible, restartable, disposable

2.1.3 Docker

- A daemon for managing Docker images and running Docker containers both locally for test, and development; as well as clustered for test, integration, staging, and production

2.2 Docker Image Security

2.2.1 Vulnerability: Embedded Secrets

- Not all files in our workspace should be copied into our image
    - secrets files and directories
    - temporary files used for build

```
# DO NOT USE, MAY COPY SECRETS INTO IMAGE
FROM quay.io/pennsignals/alpine-3.7-python-3.6-engineering
MAINTAINER pennsignals
WORKDIR /tmp
COPY . .  # <-- HERE
RUN pip install --requirement requirements.txt
```

2.2.2 Mitigations: Embedded Secrets

- Use directory convention for `/secrets` with `.env` files and env variables
- Use judicious `COPY` in `Dockerfile` as shown in good Dockerfile
- Use `.dockerignore` to block files/directories from `COPY`

    - Exclude by adding `**/secrets` to `.dockerignore`

```
.cache
.coverage
.dockerignore
.egg
.git
.gitignore
.tox

**/alloc
**/build
**/dist
**/htmlcov
**/local
**/nomad
**/secrets
**/__pycache__
**/*.egg-info
**/*.egg
**/*.db
**/*.pyc
**/*.pyo
**/*.swp

!local/.pylintrc
!local/microservice.test.cfg
!local/microservice.system.test.cfg
```

- Do not build and push images from localhost to docker image repository!
  `docker push quay.io.pennsignals/microservice:v2.0`
  - No guarantee that the v2.0 code from version control is pushed, the local docker tag isn't necessarily the tag from version control
  - No guarantee that the image doesn't include extra files or secrets from localhost
  - Use image repository to check out code from version control to build images automatically.
  - Applies version control, `.gitignore` as well as `.dockerignore`

2.2.3 Vulnerability: Un-patched Packages

- Number of vulnerabilities increases over time
- Attack surface increases with the number of packages

2.2.4 Mitigations: Un-patched Packages

- Review packages used in images: pennsignals
- Monitor the automated docker image repository scans:
  - pennsignals at 23 MB
  - node on debian at 344 MB
  - node on alpine at 25 MB
  - postgres on debian at 83 MB
  - postgres on alpine at 29 MB

- Avoid using typical windows or linux desktop or server operating systems as base images
  - Use minimal network appliance/security-focused base images
  - Alpine based images are not only smaller, but have a more secure implementation of `libc`

- Rebuild/redeploy images instead of patching, rebuild all images on a schedule
- Avoid using containerized virtual machines in production
  - Build small-sized images that have few included packages and that may be easily rebuilt

3 Running Docker Microservices

3.3 Definitions

3.3.1 Microservice

- Not a virtual machine nor a containerized virtual machine
- Not a monolithic application
- A micro component of a potentially distributed and scaled application
- Usually a web service or a web application
- Usually implemented as a small docker container
- Usually does not hold durable state
- Usually scaled by running multiple docker containers from a single docker image

3.3.2 Docker Compose

- A development and test utility to orchestrate docker containers
- Runs docker compose jobs in a virtual private network on localhost
- Usually orchestrates containerized microservices
- Usually also orchestrates containerized virtual machines for missing infrastructure or as isolated substitutes for shared infrastructure
- Maps service volumes, ports and other resources from containers to/from locahost
- Uses `docker-compose.yml` files:

```
version: "3.2"
services:

  mongo:  # a disposable database for tests
    image: mongo
    stop_signal: SIGINT
    command: --noauth
    ports:
    - "27017:27017"  # map mongo to localhost port 27017
    restart: always

  # data ingest services omitted here

  microservice:
    build: .
    entrypoint: ["microservice"]
    env_file:
    - ./secrets/microservice.env  # secrets are not in the Dockerfile
    environment:
    - MONGO_URI=mongodb://mongo/tests  # orchestration of dynamic mongo address "..//
mongo/..."
    ports:
    - "9000:9000"  # map zmq prediction events to localhost port 9000
```

3.3.3 Hashicorp Nomad

- A distributed, highly available production service to orchestrate docker containers
- Depends on Hashicorp Consul for shared cluster state
- Depends on Hashicorp Vault for injecting secrets and applying policy to jobs
- Runs nomad jobs across a cluster of nomad clients/workers
- Manages clustered docker daemons and other kinds of clustered executors
- Maps service volumes, ports and other resources from containers to/from nomad network
- Uses nomad job files:

```
job "microservice" {
  datacenters = ["dc1"]

  # multiple groups per job
  # separate groups may run on different nodes
  group "microservice" {
    vault {
      policies = ["microservice"]  # apply this security policy
    }
    restart {
      mode = "delay"
    }

    # data ingest tasks omitted here...

    # multiple tasks per group
    # NOMAD_ADDR is the ip address assigned by nomad to the task
    task "microservice" {
      driver = "docker"
      config = {
        labels = "microservice"
      }
      command = "microservice"
      env {
        MICROSERVICE_CONFIGURATION = "/local/microservice.cfg"
        PREDICTION_ZMQ             = "tcp://0.0.0.0:${NOMAD_PORT_prediction_events}"
data""
      }
      image = "quay.io/pennsignals/microservice"
      port_map = {
        prediction_events = 9000
      }
      resources {
        network {
          port "prediction_events" {}
        }
        cpu    = 8000
        memory = 4096
      }

      # inject configuration from consul
      # changes to the consul key/value will automatically restart this task
      template {
        data = <<EOH
{{key "microservice/microservice.cfg"}}
EOH
        destination = "/local/microservice.cfg"
      }

      # inject secrets from vault
      template {
        data        = <<EOH
MONGO_URI="{{with secret "secret/mongo/microservice/mongo_uri"}}{{.Data.value}}{{end}
}"
EOH
        destination = "/secrets/microservice.env"
        env         = true
      }
    }
  }
}
```

3.4 Docker Runtime Security

3.4.1 Vulnerability: Runtime secret extraction

- Vulnerabilities in a running container may expose secrets

3.4.2 Mitigations: Runtime secret extraction

- Use nomad directory conventions

    - `alloc/` : This directory is shared across all tasks in a task group and can be used to store data that needs to be used by multiple tasks, such as a log shipper.
    - `local/` : This directory is private to each task. It can be used to store arbitrary data that should not be shared by tasks in the task group.
    - `secrets/` : This directory is private to each task, not accessible via the nomad alloc fs command or filesystem APIs and where possible backed by an in-memory filesystem. It can be used to store secret data that should not be visible outside the task.

- Use nomad directory conventions early in development with docker-compose too
- Use vault policies to separate services accounts/secrets per microservice
    - Here, `secret` refers to a vault encrypted key stored in consul, not the nomad `secrets/` directory:

```
path "secret/microservice/mongo_uri" {
  capabilities = ["read"]
}
path "secret/microservice/clinstream.pem" {
  capabilities = ["read"]
}
path "secret/microservice/twilio" {
  capabilities = ["read"]
}
path "secret/microservice/mssql" {
  capabilities = ["read"]
}
```

4 Other Security-Related Items:

- Sign checked-in code
- Virtual private network hides microservice communications both using docker-compose and nomad
- Don't bind services to 0.0.0.0 by default, use 127.0.0.1 instead

5 References:

- Kelsey Hightower on 12 factor apps
- Twelve factor apps
- The Docker Book
- Consul
- Nomad
- Vault

6 Appendices:

6.1 Infrastructure Goals

- Run durable production and staging environments for deployment and A/B tests of microservices
- Run durable integration environment for scalability, load, system infrastructure tests, and infrastructure evolution
- Run ephemeral test environments for production-like microservice systems tests
- Automate infrastructure as code, eliminate configuration drift, rebuild vs patch

6.2 Software Engineering Goals

- Automate all build, unit tests, systems tests, and **deployment** to a production-like test environment from code commit
- Easy local development environments with few dependencies
- High fidelity between local development environment and production-like test environments

6.3 Development and Test Environments

- Local or unit test service running docker-compose
- Usually runs a containerized database for output and microservice state

6.4 Production, Staging, Integration, and System Test Environments

- Cluster of 3 consul servers for highly available, distributed, shared state
- Cluster of 2 vault servers for reading and writing encrypted secrets to consul as well as applying security policy
- Cluster of 3 nomad servers to orchestrate docker containers as non-containerized microservices, and services
- Cluster of N nomad workers to run the docker containers, microservices, and services
- A private network to hide all cluster and intra-cluster communications
- A load balancer to expose web apps and web services to our intranet
- A shared on-premise database for persistence
- Run nomad jobs instead of docker-compose jobs, usually docker containers with microservices