Creating a Jenkins Build Farm with Docker

Chris B. Behrens
SOFTWARE ARCHITECT

@chrisbbehrens

Demo

Ensure that the Docker Remote API is enabled

Configure Docker in our Jenkins master

Create a single containerized build agent

Slave it to our master

Execute a labeled build

Review the results

What’s Going on Here with Cloud Agents?

1. **Jenkins resolves the label, if any**
2. **Jenkins phones home to the Docker Host via REST**
3. **Jenkins provisions this running container as an agent node**
   - Executes the build
4. **Build succeeds or fails**
5. **Jenkins tears down the container**

Problems with This

Our Jenkins image is dumb and pointless

With our current config, we'd have to push this to a public DockerHub repo

Just find an image with the prerequisites you need

Understanding Docker Images and Trust

Take as little as possible for granted

Trust as little as possible

We trust "Jenkins"

- The Jenkins organization and process

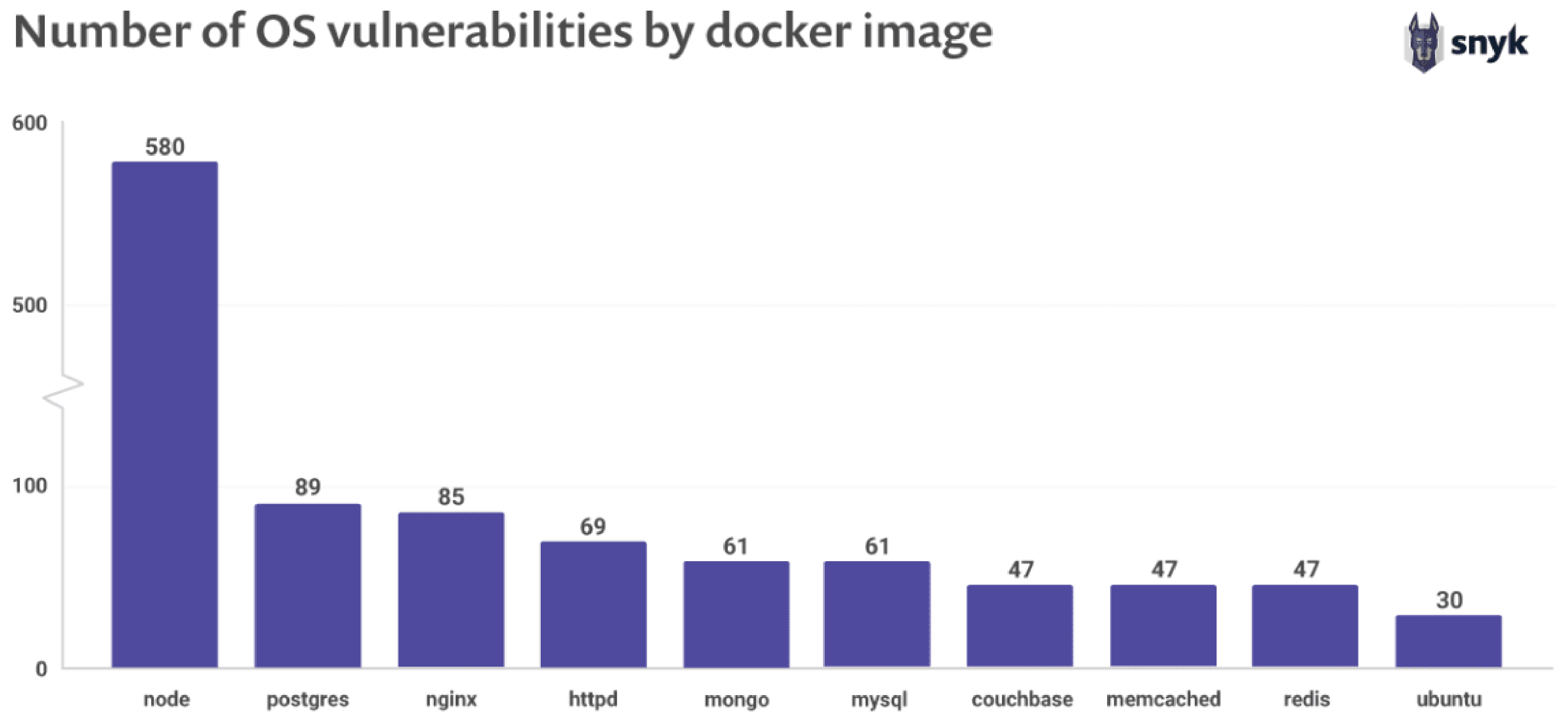
Whom else must you trust?

Integrity and COMPETENCE

“Never attribute to malice that which is adequately explained by stupidity.”

Hanlon’s Razor

Open Source Vulnerabilities



https://snyk.io/blog/top-ten-most-popular-docker-images-each-contain-at-least-30-vulnerabilities

Prefer minimal base images

https://snyk.io/blog/10-docker-image-security-best-practices

Demo

Create our DotNetCore image

Using a Dockerfile

- Performs each step necessary
- To install the DotnetCore SDK

Run our image

Execute a super-simple build on the container

Demo

Attach this image to a template for our cloud

Create a C# command line project

Whip up a quick Jenkinsfile which builds it

- Restrict it to our dotnet core agent

Execute a build

Review the results

Wrapping up with Your DotNetCore Agent

```
FROM jenkins/jenkins:lts
USER root
RUN apt-get update && apt-get install -y --no-install-recommends \
    curl libunwind8 gettext apt-transport-https && \
    curl https://packages.microsoft.com/keys/microsoft.asc | gpg --dearmor > 
microsoft.gpg && \
    mv microsoft.gpg /etc/apt/trusted.gpg.d/microsoft.gpg && \
    sh -c 'echo "deb [arch=amd64] https://packages.microsoft.com/repos/microsoft-
debian-stretch-prod stretch main" > /etc/apt/sources.list.d/dotnetdev.list' && \
    apt-get update
RUN apt-get install -y dotnet-sdk-3.1 && \
    export PATH=$PATH:$HOME/dotnet && \
    dotnet --version
USER jenkins
```

How this Worked

The Jenkins build was durable

Troubleshoot agent problems using the log – the problem is usually obvious

No volumes – workspaces go up in smoke. This is a good thing.

```
FROM jenkins/jenkins:lts

USER root



…



RUN apt-get install -y dotnet-sdk-3.1
&& \

    export PATH=$PATH:$HOME/dotnet &&
\

    dotnet --version

USER jenkins
```

- Create your own Dockerfiles to provision build agents
- Design them right, and they will auto-update
  - More on this later
- Build your images continually
- A standard, repeatable, automated process?
  - That's called a build
- Build our image IN Docker
- Push it to DockerHub
- A meta-agent build

Critiquing Your Dockerfile

Ideally, we'd target DotNetCore-SDK:lts rather than a specific version

But our Jenkins LTS will include the latest (stable) patches and plug-ins

Demo

Whip up a Docker image with

- JenkinsDocker

Attach this agent to Jenkins

Create a Jenkinsfile

- Builds our DotnetCore image
- Pushes it to DockerHub

Why This Is Great

Jenkins LTS – continually patched and updated

Open JDK – continually patched and updated

Debian – continually patched and updated

An agent meta-build test build

- 1. Build the new agent image
- 2. Run the new container
- Execute a simple build against it
- If this succeeds, build the image and mark it with a “stable” tag

Your Jenkinsfile

```
def dockerImage;


node('docker'){
      stage('SCM'){
            checkout([$class: 'GitSCM', branches: [[name: '*/master']], doGenerateSubmoduleConfigurations:
false, extensions: [], submoduleCfg: [], userRemoteConfigs: [[url:
'https://github.com/FeynmanFan/JenkinsDocker']]]);
      }
      stage('build'){
            dockerImage = docker.build('chrisbbehrens/agent-dnc:v$BUILD_NUMBER', './dotnetcore');
      }
      stage('push'){
            docker.withRegistry('https://index.docker.io/v1/', 'dockerhubcreds'){
                  dockerImage.push();
            }
      }
}
```

Understanding Container Connect Methods

Attach Docker Container

Connect with JNLP

Connect with SSH

Why Not Use ‘Attach Container’?

To protect against interception of traffic between Jenkins master container and agent container

An MITM attack could read the traffic

Connect Method Prerequisites

Attach Container	JNLP	SSH 🔒
• Entrypoint must be able to accept jenkins slave connection parameters	• Jenkins master has to be accessible over network from the container • Docker image must launch slave.jar by itself or using EntryPoint Arguments	• Docker image must have sshd installed • The Docker container's mapped SSH port, typically a port on the Docker host, has to be accessible over network from the master

Working with Private Registries

Public registries expose some of the details of your build

Whether your private registry is in DockerHub or elsewhere

- We need to control the registry url
- And specify credentials

Our credentials will be transmitted over port 2375 from the Jenkins master to agent

- In spite of the Credentials masking
- If this bothers you, connect with SSH

DockerHub is only special in one way – it's the default

Working with Private Registries

1. Pushing to a private repository in DockerHub via a build
2. Pulling an image from a private repository for our build agent

Demo

Take our DotNet Core Agent private in DockerHub

Modify our Jenkinsfile to point at the DockerHub URL

- To represent a non-default registry URL
- Verify that it works

Modify our agent template

- To pass credentials to DockerHub

Verify that our DotNetCore agent still works

Installing Dependencies Dynamically

Can we provision our dependencies in our *Jenkinsfile?*

Would allow for an extremely generic agent (a good thing)

It would be helpful when you can't modify the agent image

It would require root permissions

Installing Dependencies Dynamically

```
node('slave’){
    stage('Install Node’){
        sh 'curl -sL https://deb.nodesource.com/setup_12.x | bash’
        sh 'apt-get install build-essential nodejs -y’
    }
}
```

Ways Around This

Store your binaries in version control - YUCK

Access your tools via volumes and copy operations

Build your tools – in the same build

Dynamic Dependency Options

Requires Installation	Simple Binary	Have the Source
Use a Dockerfile - period	Store it in version control or copy it from a volume	Build it and put it where you want

Don’t Commit, Round One

A Container commit looks like saving a VM

But it’s not

VM’s and Containers solve different problems

Commit turns a transparent and settled structure

Into an opaque one with a black box on top

Dockerfiles function as *documentation*

Don’t ever do it

Working with Ephemeral Agents

Workspace can be key in troubleshooting Jenkins

Sometimes you don’t need the workspace

Take the results and stick them SOMEWHERE

Demo

Add an archive step to our build

Execute a new build with it

Review the results

Break our project

Trigger another build

Look at the results

To find the problem

Summary

Provisioning the simplest possible build agent

- One built directly off of Jenkins/Jenkins
- Executed a simple Hello World build

Ramping up the complexity of our agents

- A DotnetCore dockerfile
- Based on the Jenkins image
- Ran Docker inside a Docker container

Generalized our connections

Demo of Artifacts

To preserve the state of our ephemeral build agents