Patterns of Software Development with Containers

KYLE BROWN, IBM Corporation
CHRISTOPHER HAY, IBM Corporation

Abstract This pattern language is concerned with the problems inherent in building and delivering software using containers, particularly those issues that arise during the process of mapping Docker images and containers into the stages of a software development lifecycle. We discuss practical solutions to problems of security and scaling brought on by adoption of container platforms. The language assumes that the reader will be building applications following an agile approach that is characterized by Continuous Integration/Continuous Delivery. This paper is part of a larger set of patterns of cloud adoption; for information on patterns of microservices adoption that may precede these patterns see [Brown 2016].

Categories and Subject Descriptors: • **Software and its engineering~Software design engineering**

General Terms: Software Architectures

Additional Key Words and Phrases: Docker, Pattern Languages, Containers

ACM Reference Format:

Brown, K.. and Hay, C. 2018.Patterns of Software Development with Containers. HILLSIDE Proc. of Conf. on Pattern Lang. of Prog. 25 (October 2018), 14 pages.

1. INTRODUCTION

1.1 Overview

Containers are one of the most rapidly adopted software technologies of the last several years, with extraordinary growth in adoption [PortworxSurvey]. This rapid adoption is the result of an impressive increase in developer productivity and ability to delivery cost reduction resulting from container adoption [Synopsis]. Containers are a way of packaging software that provides a lightweight mechanism for bringing application code and configuration together with all of the software prerequisites (such as a language runtime, an application server, or libraries) that the application depends on. Containers differ from traditional virtualization platforms in that the container does not include the entire guest OS, but instead relies on the services of the container platform to provide isolation from other processes running within the container platform.

The most common container platform is the open-source Docker platform, introduced in 2013 [DockerBlog]. This platform has come to dominate the container industry; with one survey by Datanyze [Datanyze] showing that it is in use by over 75% of container users, either directly in conjunction with the Kubernetes container orchestrator. The patterns in this pattern language have thus been written with the Docker platform in mind, although it is conceivable that they would also apply to other container platforms.

In Docker, a container (which can be thought of as a running instance of a software package and code) is implemented as an isolated user space running within a running Linux OS (a Docker **host**), while the platform provides a shared kernel across containers running within that OS. All software packages and data in a container are isolated at run time. Resource management is implemented with Unix cgroups while resource isolation is provided through namespaces. Filesystem isolation is managed through the Docker file system, which is an additive, or union, filesystem using copy-on-write semantics.

1.2 Terminology Used

However, before we introduce our patterns, we need to introduce a few terms used in Docker that are central to our patterns. First, as we have already discussed, a **container** is a running instance of software executing within the Docker environment. The combination of software packages, code and associated prerequisites is packaged within Docker as an **image**.

The image is an individual instance of a layered file system where each layer builds on top of the layers below it. An example (showing specific application server software, applications and a particular operating system release) is shown below. (See Figure 1. Layers in Docker).

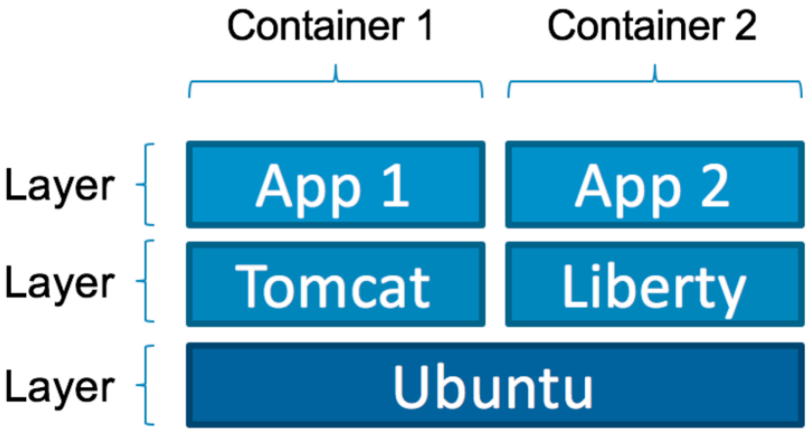

Fig. 1. Layers in Docker. This diagram shows how Layers in the docker filesystem are shared across containers.

Build files called **dockerfiles** define images. A dockerfile begins with an existing image as the starting point and provides a set of instructions to augment that image (each of which results in a new layer in the file system). It also includes meta-data such as the ports exposed and the command to execute when the image is started. Consider the following example of a dockerfile in order to see how this works (see Listing 1).

```
# Pull base image.
FROM ibmnode:v6
MAINTAINER Chris Hay <chris.hay@uk.ibm.com>

# Install Java.
RUN apt-get update && \
apt-get upgrade -y && \
apt-get install -y software-properties-common && \
add-apt-repository ppa:webupd8team/java -y && \
apt-get update && \
echo oracle-java8-installer shared/accepted-oracle-license-v1-1 select true | /usr/bin/debconf-set-selections && \ apt-get install -y oracle-java8-installer && \
apt-get clean

# Define working directory.
WORKDIR /data

# Define commonly used JAVA_HOME variable ENV JAVA_HOME /usr/lib/jvm/java-8-oracle
```

Listing 1. Example Dockerfile for Java

The dockerfile shown in Listing 1 takes an existing Node.JS base image and creates a new image that includes the Java 8 runtime. The dockerfile will use the ibmnode (version 6) base image as a starting point and will then download and install the Java 8 runtime from Oracle.

A Docker **registry** is a service for storing and retrieving Docker images. You can think of it as being like a source-code control system (e.g. Git) for docker images. There are two general types of registries; a public registry is one that provides this service to many customers where the images are publicly available and

searchable. Examples of this include Docker Hub, the Amazon Elastic Container Registry, and the IBM Container Registry service. A private registry is one that serves a single customer. Both types of registry may be cloud hosted, although private registries are sometimes also deployed on premises. For instance, in Docker you can deploy your own registry services and store your images locally or in any other location running docker (such as a hosted private cloud).

A Docker **repository** is a collection of related docker images that have unique tags. A tag is an alphanumeric identifier for an image within a repository. For instance, Docker Hub allows you to create new repositories via the "Create Repository" function. This named repository then becomes a common name that is used as part of the identifier of images within a *docker push* or *docker pull*, e.g. **docker push user/repository-name:tag**. Other registries also support similar approaches to creating repositories.

1.3 Patterns Introduced

As described in the introduction, the patterns in this language address issues faced by development teams using containers, specifically questions around building and deploying software on Docker in a secure and repeatable way. Our pattern language contains nine patterns that describe solutions to the problems addressed by these teams.

- **Docker Build Pipeline** is the root pattern of this pattern language. A DevOps pipeline is a core concept for Continuous Integration/Continuous Delivery. An issue many teams face is where to introduce docker into their delivery processes. Starting with an automated delivery pipeline for building and deploying your docker images leads to the other patterns in this section.
- **Pipeline Vulnerability Scanner** enables you to perform static vulnerability scans as a stage within your *Docker Build Pipeline* in order to scan your container image(s) for any known vulnerabilities and stop the deployment and report the issue if any issues are found.
- **Multiple Vulnerability Scanners** addresses the issue that different vulnerability scanners use different approaches and pull threats and malware definitions from different repositories. Teams should hedge their bets by scanning images in multiple ways.
- **Registry Vulnerability Scanner** gives you the ability to scan images *after* they are built, so that new vulnerabilities that are detected after a build can be detected and addressed.
- **Birthing Pool** is a way to avoid placing an untested image into an environment shared with other development stages, allowing malware present in that image to affect those other stages.
- **Public Image Registry** is a solution for making images available to others who may be outside of your development organization.
- **Private Image Registry** is a solution for making images available to those within your organization, particularly useful in cases of intellectual property restriction or security restriction.
- **Approved Image Repository** is the location for approved images once they have been through the scanning and vetting process.
- **HA Container Registry** is important because a registry is only useful when it can be accessed. Encountering a single point of failure on Docker host startup will result in your entire Docker architecture being unavailable.
- **Public Registry Proxy** is a way of improving performance of image pulls in some use cases by locally caching images nearer to the Docker hosts.

A map of all of the patterns in the pattern language is shown in the following figure (see **Error! Reference source not found.**):

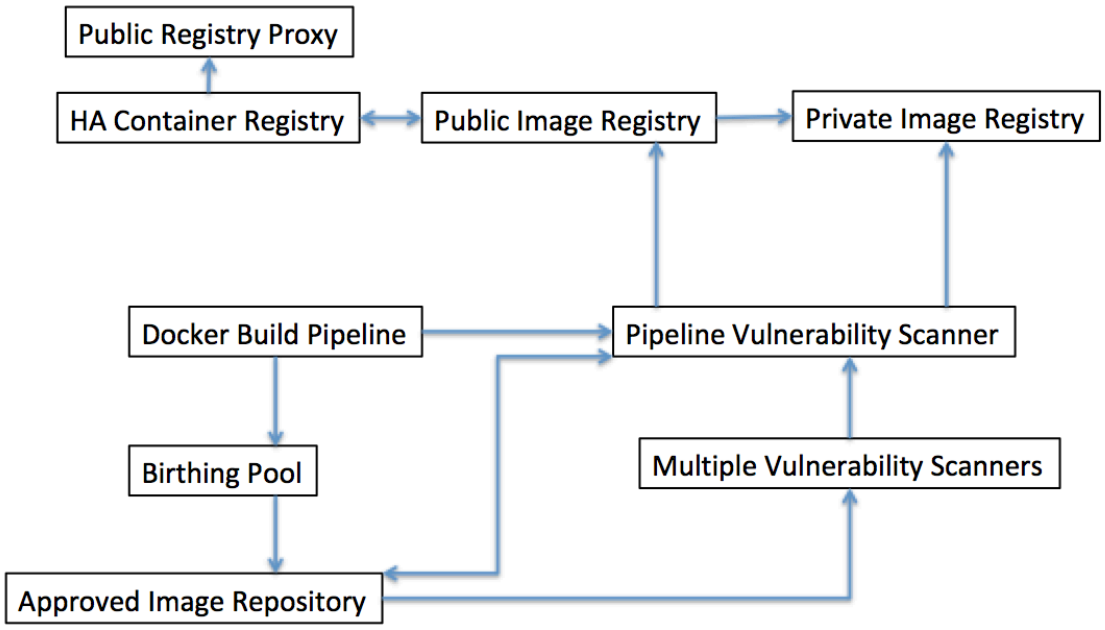

Fig. 2. Map of All Patterns. This diagram shows the relationships between all of the patterns in our pattern language.

2. Patterns described in our pattern language

In the following section we will describe all of the patterns in the language in detail.

2.1 Docker Build Pipeline

When using Docker containers as a tool for deployment, a development team faces a series of choices. Docker is often adopted by teams that are in the process of evolving their practices away from more traditional development approaches toward more agile approaches. As such, these teams often have established build processes that they need to replace, or processes they should retarget to take advantage of the new capabilities that Docker containers offer. At the same time, the opportunity presented by adopting Docker containers also means that teams can modernize their development, deployment and testing processes to take advantage of new tools such as Jenkins [Jenkins] or Hudson that introduce more modern approaches.

How can development teams exploit the capabilities provided by Docker containers and adapt their build and testing processes to take full advantage of those capabilities?

Docker introduces several concepts that make it challenging for teams to directly adapt existing practices directly to functioning with Docker. One of the most challenging for teams that are used to traditional approaches is the concept that Docker containers, especially those in production, should be *immutable*. Immutable production systems are not a new concept in software engineering - in fact, a big part of the attraction of infrastructure-as-code was the idea that systems could be made immutable if only they could be constructed entirely from the ground up from a repeatable code base instead of having to be constructed from an ad-hoc mixture of code and far-too-mutable physical or virtual infrastructure.

However, the infrastructure-as-code approach was often not adopted entirely throughout an entire software development lifecycle. In more traditional software development approaches, an *environment* will be an entire system, often built as one or more Virtual Machines or physical environments, that serve as a location in which one particular step in a software engineering lifecycle, such as application build, integration testing, or

acceptance testing will take place. What we have commonly seen is that development teams will have a distinction between "lower" environments in the development lifecycle, such as build and unit test, which are very open and mutable, and "higher" environments such as an acceptance test environment, that become progressively more locked down.

What this leads to is a situation where lower environments are often constantly "in flux" and that changes in configuration are not picked up in later environments, causing problems that are fixed in earlier environments to recur unexpectedly later. What's more, the inconsistency of the mechanisms for defining and configuring environments results in wasted time and needless repetition of work.

Therefore,

Build a Continuous Integration and Continuous Delivery Pipeline, using common tools such as Jenkins, in which the output of each pipeline run will be an immutable Docker image.

Jenkins is an open source tool that is used throughout the software development industry to define and build Continuous Integration and Delivery pipelines. Jenkins is built on the concept of a stage, which is a conceptually distinct subset of a pipeline. Each stage is built of steps that can execute within conditional logic to automate common tasks such as building a Java Jar file using Maven, or running unit tests with an automated tool like JUnit. Thus each stage can conceptually map to a physical or virtual environment of the type described above such as "Build" or "Unit Test".

The key here is that you can use a tool like Jenkins combined with Docker to entirely eliminate the need for any of these unique physical or virtual environments. Instead you will build a Docker image from a dockerfile in the initial setup of the pipeline, and then push this image to the image registry upon successful completion of the pipeline stages. The image is entirely rebuilt on each new run of the pipeline.

This approach will fix the problem of reintroducing errors into later environments by entirely removing manual configuration changes from the process. In this approach you can't change the configuration of an image either intentionally or accidently within a single stage – you have to introduce any configuration changes into the Docker build process at the very beginning and then let the changes propagate through the entire build pipeline. So for instance, let's consider the simple case of changing the version of a Java runtime environment (JRE). In a traditional approach, with separate physical or virtual machines for each development lifecycle stage, updating this configuration would require changing each environment separately, either manually or through a scripted infrastructure-as-code tool such as Chef or Puppet. In the Docker approach, you would change the dockerfile once to include the new definition, and then re-run the pipeline to repeat all the automated steps from the beginning – creating a new, immutable Docker image at the end.

This pattern is well established as a best practice within the Docker community. For instance, the Docker documentation [Development Pipelines] describes a recommended development pipeline very much in line with the recommendations of this pattern. Likewise [Kubernetes DevOps] is just one of several examples of such pipelines being built for Docker container projects.

At the heart of this pipeline will be the problem of dealing with images appropriately. The first issue to consider with publicly hosted images is that since they are coming from a public repository that they could, potentially, contain malware or other issues that would introduce vulnerability into your system. Thus the need for *Pipeline Vulnerability Scanning* becomes absolutely critical. This results in the need to introduce special stages. such as a *Birthing Pool* into your Pipeline in order to make sure that you are into introducing new types of vulnerabilities into your systems.

2.2 Pipeline Vulnerability Scanner

You are building a new cloud native application that is hosted in a container. You want to use container images that are obtained from existing public registries in order to take advantage of the work of others in the Docker

community. However, you also want to ensure that container images with known vulnerabilities are not deployed to the image registry from the CI/CD Pipeline.

How do you prevent Docker image with known vulnerabilities from being uploaded into your image registry in the first place?

There are several issues that are brought up by the use of an image registry. First of all, you don't want your *Public Image Registry or Private Image Registry* to contain vulnerable images; you want issues to be sorted out before the image is deployed to the registry. You don't want to place the responsibility entirely on the developer; you want them on focusing on writing code. Likewise, you don't want developers to be forced to go through a lengthy and arduous security review.

Therefore,

You should ensure that your CI/CD pipeline has a vulnerability scanner included in one or more of the stages in your pipeline. The scanner will scan your container image for any known vulnerabilities and stop the deployment and report the issue if any issues are found. The static vulnerability scan should check public vulnerability databases such as CVE at a minimum.

The types of scanners that we are referring to are often referred to as static scanners in that what they do is examine the configuration of a Docker image in looking for known vulnerabilities to operating system, language runtime or middleware. This is different from an active scanner, which examines the activity of a running docker image for known malware behavior. An example of the overall build process (a potential model for a Docker build pipeline) including a pipeline vulnerability scan is shown in **Error! Reference source not found.**.

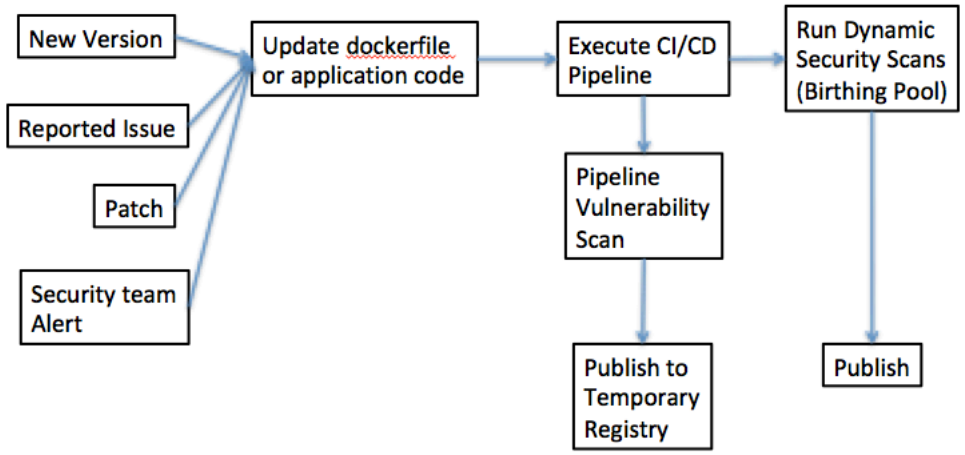

Fig. 3. Docker Build Process

Let's look at **Error! Reference source not found.** in detail. At the beginning of the process, you have a number of different input triggers that can result in the need to update an image – in all cases, what this amounts to is changing the dockerfile to introduce a configuration change, or updating the application code executing within the dockerfile that will be built as part of the CI/CID process. When you do so, this acts as a trigger to the CI/CD build pipeline – in which one step in the pipeline will be a vulnerability scan. That scan must complete successfully before the image can be published to a temporary registry in preparation for running dynamic scans (see *Birthing Pool*).

However, it's not enough to simply include a vulnerability scan as part of a DevOps pipeline, you also need to alert your DevOps, Site Reliability Engineering or run teams of any vulnerability scans as and when they occur.

Especially as vulnerability databases are constantly updated, it may be true that an image that has passed an earlier scan and already been deployed to production may fail a later scan in an earlier stage against an updated database.

You want your images to be scanned regularly; not just when code changes, in case a new vulnerability is discovered post-release. Therefore it is important to have triggers such as notifications from your security team of new vulnerabilities that can begin a new pipeline run. Likewise, you need to have a *Registry Vulnerability Scanner*, in order to catch vulnerabilities post-build, but you don't want to purely dependent on the vulnerability scanner of your image registry.

Open-Source scanners such as Clair [Clair] can provide such functionality and can be easily integrated with CI/CD pipeline tools such as Jenkins. There are multiple other examples of such tools, such as the open source Docker Bench Security [Docker Bench] and the commercial IBM Vulnerability Advisor [Vulnerability Advisor].Using Vulnerability Scanners is another well-established practice in the Docker development community, cited specifically as a best practice in [7 Threats] and [Sumologic].

Using a *Pipeline Vulnerability Scanner* integrated into your CI/CD pipeline increases the overall security of your solution by ensuring that your image is secured from known vulnerabilities and prevents vulnerable images being uploaded into your *Public Image Registry* or *Private Image Registry* in the first place. You should also have a policy that prevents any unsecured images being deployed into production. Only allowing teams to only pull from an *Approved Image Repository* is one way of doing this.

2.3 Birthing Pool

You have a new cloud native application that is based on a container technology (such as Docker) and you want to ensure that container images with known vulnerabilities are not deployed to your environment. However, some vulnerabilities within a container image cannot be picked up by a static vulnerability scanner as they can only be found in a running container.

How do you detect such vulnerabilities without deploying a potentially vulnerable image and into a test or production environment?

Vulnerabilities present in a running container should be isolated to a micro-segmented network where the impact cannot be replicated to other machines. However, if we place an untested image into an environment shared with other development stages, then malware present in that image may affect those other stages.

Therefore,

Create a new environment as part of your overall CI/CD process consisting of an isolated environment called a birthing pool. Run dynamic vulnerability scans within this environment in order to limit the exposure of other Docker runtime environments to potential malware.

Up to this point, we have been considering that the isolation provided by Docker itself, in that each image is functionally isolated from other images by the Docker execution environment is adequate for all types of vulnerabilities that may be found in an image. However, that may not be true. There is the possibility that a side-channel or Docker infrastructure attack (such as forkbomb, see [Baset]) may interfere with the operation of your Docker execution environment and thus affect other Docker images. Thus the need exists to have at least two separate Docker execution environments, each segmented from the other, in order to eliminate this possibility.

An example of all of the different parts of an end-to-end Docker development process, including a separate environment for active vulnerability scanning within a birthing pool, is shown in **Error! Reference source not found.**.

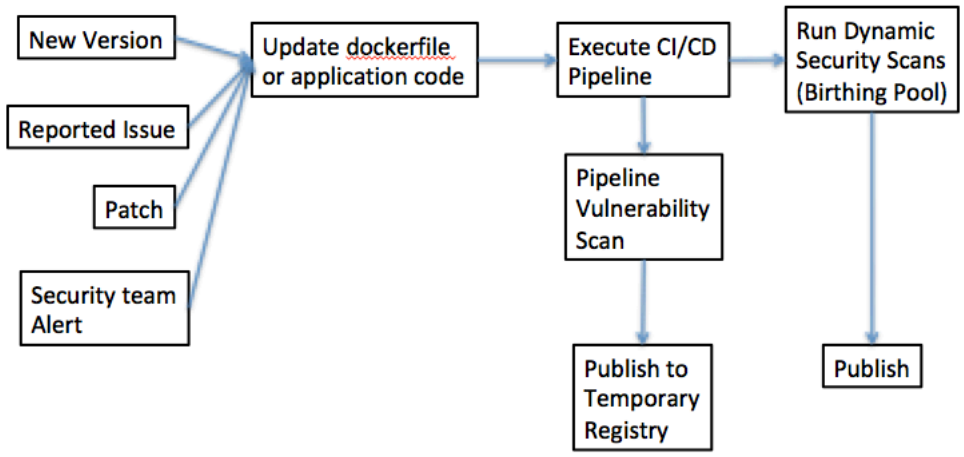

Fig. 4. Stages including Birthing Pool

You should be able to create the new birthing pool environment from scratch using infrastructure as code techniques and tools, such as Terraform. The good news about this approach is that since we are creating an uncustomized, "off-the-shelf" installation of a common runtime such as Kubernetes or Docker Enterprise, that this can be done without introducing the kind of configuration drift caused by team-level specialization we discussed earlier. The birthing pool should contain no sensitive data. This allows you to run your container within the birthing pool and allow you to run a vulnerability scan in this isolated environment and detect any vulnerabilities that can only be found through dynamic scans.

The introduction of the birthing pool means that a dynamic scan can be performed without exposing other aspects of the application estate to the vulnerability. One potential implementation of this, using separate Kubernetes clusters for each different environment in Figure 5: Isolation of the Birthing Pool.

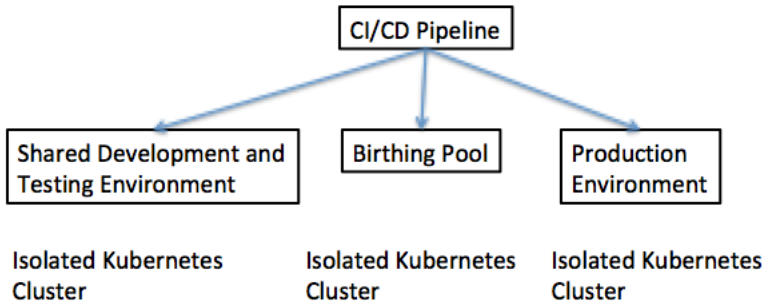

Fig. 5. Isolation of the Birthing Pool

Images should pass through a *Birthing Pool* before they are placed by a *Docker Build Pipeline* into an *Approved Image Repository*.

2.4 Registry Vulnerability Scanner

You have a new cloud native application that is based on a container technology (such as docker) and you want to ensure that your image is clean and secured from any vulnerability such as malware or known security

vulnerabilities. In order to prevent vulnerabilities being deployed into your Container Registry you utilize a *Pipeline Vulnerability Scanner* to check for vulnerabilities on creation of the image during the CI/CD pipeline.

What if vulnerability is created post deployment of the image into the Container Registry? How do you ensure that vulnerabilities are detected?

You don't want to only rely on the *Pipeline Vulnerability Scanner* to pick up all vulnerabilities in case the CI/CD pipeline is bypassed and someone uploads an image directly. Likewise, you don't want scans to be performed manually but want them to be scanned periodically. You want an image to be scanned not only at a build time just in case a new vulnerability is discovered post-release

Therefore:

Ensure that your image registry has a vulnerability scanner that will both scan your container image for any known vulnerabilities on upload of your image and at a regular intervals. The vulnerability scan should check public vulnerability databases such as CVE at a minimum.

Using a vulnerability scanner integrated with your image repository increases the overall security of your solution by ensuring that your image is secured from known vulnerabilities.
You should set up a policy that:

- Prevents any unsecured images being deployed into production
- Alerts your devops, SRE or run teams of any vulnerability scans as and when they occur
- Provides a scan report at regular intervals that can be reviewed by your devops, SRE or run teams

Cloud Hosted Image Registries such as Dockerhub, IBM Cloud, and [quay.io] all contain this capability by default.

2.5 Multiple Pipeline Vulnerability Scanners

You are developing a new application that is hosted in a container and you want to ensure that container images with known vulnerabilities are not deployed to your environment.

Not all vulnerability scanners use the same vulnerability databases and some are more complete than others. How do we ensure we don't rely on a single source of vulnerability information for discovering and reporting issues?

Vulnerability Scanners come from different vendors and open source teams and as a result, different vulnerability scanners will be updated at different rates. Likewise, different vulnerability scanners use different databases and a specific vulnerability may not be reported in all databases. It is also possible that any particular vulnerability scanner may fail to detect a vulnerability, either because of bugs or because a malware author can attack a vulnerability scanner itself.

Therefore:

Use *Multiple Pipeline Vulnerability Scanners* such as Clair and the IBM Vulnerability Advisor at multiple points in your pipeline.

The best pattern in such a scenario would be to use an open-source scanner such as Clair as part of your CI/CD pipeline and use a different vulnerability scanner for your image registry.

Applying multiple vulnerability scanners to address deficiencies or blind spots in a single scanner or database is a well-known best practice in the security industry documented in [SoftwareSecured]
Using multiple vulnerability scanners within your *Docker Build Pipeline* gives you an increased level of security as there is a reduced risk of vulnerability not being found due a single scanner not being aware of the vulnerability.

2.6 Public Image Registry

You have developed a new container image that you wish to make available to the wider development community. This may be a new container image you wish to make available to your customers, a new open-source application that you wish to make available, or an improved installation of an existing open-source product that you wish to contribute back to the community.

In each of these cases, you should assume that there is no IP restriction and that you wish to make the image available publicly. Likewise, there should be no commercial or licensing implications to distributing the image.

How do you make all supported versions of your image available to the general public in a manner that allows you to easily fix bugs or vulnerabilities, distribute new versions and simplify installation?

There are several reasons that lead developers to want to use Docker images in the first place. For instance, many commercial development teams have high support costs because customers incorrectly install software installations due to complex instructions.

Likewise, development teams wish to be able to quickly and reactively provide frequent patches to existing versions with minimal impact. Critical to this is the ability to distribute new versions of your application and make them available quickly and securely.

Therefore:

Publish all supported versions of your image, correctly tagged with the right version, to a *Public Image Registry* such as Docker Hub. By centralizing the image distribution you are able to harden the image, fix images, fix bugs and then provide an updated version of the image for your application as you fix issues.

If you need to release a new version of your application then you are able to tag the new version of your image and make it available immediately allowing consumers to choose when they should use the new version of your application.

Consumers of your image will be able to just pull the version of the image that they need instantly without having to perform any complex installations.

Publishing images to a *Public Image Registry* allows you to centralize the distribution of your application in a secure manner and allows you to provide frequent releases of your application or image in a secure fashion for those images that can publicly distributed. However, the benefits of a Public Image Registry can only be realized if it is available, thus the need for an *HA Container Registry*.

When publishing images to a public registry you should ensure that in your published image that you don't break any licensing concerns by using unlicensed software or IP, and that you do not include any sensitive data such as Keys, Passwords, Infrastructure information such as internal IP's in your images. Any requirement to include these types of information in your images will instead require you to use a *Private Image Registry* instead.

Likewise, you will want to use a *Pipeline Vulnerability Scanner* to ensure you don't pass vulnerabilities to consumers of your images.

2.7 Private Image Registry

You have developed a new internal application that will run on containers and you need to host the image in a container image registry so that the container technology (such as Docker) can pull and run the image on your container environment.

How do you gain the benefits of a *Public Image Registry* (such as Docker Hub) without making all of your images available to the general public?

There are several reasons why you may not be able to use a Public Image Registry. As noted in that pattern, your image may have to contain private information such as license keys or internal network structures. In this case, making your image available in the public would expose your architecture and make it more likely that your system will be hacked. Likewise, you may not want to reveal IP or secrets; if it is made available to the public, there is always a risk your image could be reverse engineered.

Therefore:

You should host your container images in a *Private Image Registry* such as Nexus, or a private registry of your cloud platform. The *Private Image Registry* supports all the same protocols and behaves exactly like a *Public Image Registry*. This means that you have to configure your container servers to pull from the URI of the *Private Image Registry* instead of the default *Public Image Registry* (e.g. Docker Hub).

Using a *Private Image Registry* gives you all the benefits of a public registry but keeps your container images private and secured from users or applications that you should not have access to it.

This pattern has been described as a best practice by many different organizations, such as Bob Cares [BobCares] Center Device [CenterDevice] and Macadamian [Macadamian].

2.8 Approved Public Image Repository

You have developed a new internal application that will run on containers. The solution has dependencies on third party images hosted in a *Public Image Registry*. In general, developers should not have to worry about the use of third party images. However, there is a balance that needs to be struck between ease of use and security.

How do you prevent random images, or multiple versions of existing images, from being pulled from a *Public Image Registry* making your overall Docker installation estate more vulnerable and harder to maintain and operate?

In general, you want to give developers the freedom to use open-source images from the Internet. However, this freedom comes at a cost. For instance, a complete lack of governance could mean that you have multiple versions of an operating system across many applications - all of which would need to be tracked, maintained and patched.

Licensing is also a potential issue. An unapproved image could be used which could create an open-source licensing issue for your application. Likewise, an unapproved image could be used which could create a commercial licensing issue for your organization.

Maintenance of unapproved images can also cause a potential problem. A team may inadvertently create a dependency on an ungoverned image from an amateur programmer that may not maintain it, or an ungoverned image could disappear and then cause your application to fail.

Finally, security is an issue; an ungoverned image dependency could be modified and have malware installed on it.

Therefore:

Create an *Approved Public Image Repository* within your *Private Image Registry* where you have pre-downloaded all images and versions that are approved to be used in your estate. You should lock down your container hosts so that they cannot pull images directly from *Public Image Registries* and must come through your *Private Image Registry*. Only approved image registry administrators should be able to add images to the repository.

It is critically important that images should be run through a *Pipeline Vulnerability Scanner* (in fact, they should be run through *Multiple Vulnerability Scanners*) before they would be added to the Approved Public Image Repository. That ensures that developers can begin from a starting point that is known to be secure.

The governance of the *Approved Public Image Repository* can be configured at multiple levels including:

* Organization
* Program
* Project
* Application

Many *Private Image Registry* solutions possess the ability to configure such patterns and rules as part of the product.

Using an *Approved Public Image Repository* increases the security and governance of your solution while giving developers the freedom to use the best of breed tools and open-source images.

2.9 Highly Available Image Registry

You have an Service Level Agreement (SLA) for your cloud application (such as 99.99 availability) and you need to ensure that you meet your SLA's and that you have a fast time to recovery.

The issue is that in the cold start of a container host, the local cache of the host's image registry will be empty and will require the host to refetch any images from the container registry. This situation is even more likely in a disaster recovery scenario (as it's likely that hosts will have been restarted).

How do you ensure that your container startup will not fail at the worst possible time by having them rely on an unreliable image registry?

The SLA on the overall solution will dictate how much effort you need to put into the availability of your registry solution. A key factor in that is the pull time of an image from a container registry; for instance if the average pull time of an image is significantly greater than the Return To Operations (RTO) objective of your registry, then you will not notice disruptions lasting less than the RTO. A potential complication to determining how well you can meet your overall solution SLA is that a Container Registry as a Service provider may or may not have it's own SLA

Therefore:

Ensure that the container registry has high availability (e.g. has been redundantly deployed in a multi-region, multi-availability zone way) with a matching SLA.

This is a best practice that has been documented for Docker in several places such as [Labourdy]. Many companies advertise commercial container registries (such as [Portworx]) that are highly available, but it is rarely stated why you should care that this is so. You should also ensure that you are using a *Public Registry Proxy* in your container registry to ensure fast fetches of 3rd Party Images. You should also ensure that your cloud provider (or if managed by your client organisation) is able to meet the SLA's that you are committing to. If the container registry is dead and your cache is cleared then you've just lost that data center.

Ideally you should follow practices of bulk-head isolation and ensure that you have isolated and affinitized registries to your regions. In that way, if you lose your registry then it's only lost for a single region, not for every region. Of course, the affinity should only extend to those cases where it is required for performance reasons; if you can fetch an image from a distant registry when your local registry is down and that can still be done within your overall solution SLA, that is a valid option.

Using a *High Availability Registry* (with matching SLA) for a container registry (including a *Public Registry Proxy*) will ensure that the Image registry remains available even in a disaster scenario. Likewise you will experience increased availability of the image registry, as there will only be a single dependency instead of depending upon multiple providers, each with different SLA's.

2.10 Public Registry Proxy

You have developed a new internal application that will run on containers. The solution has dependencies on third party images hosted in a *Public Image Registry*.

How should you handle access to images in *Public Image Registries* when you are concerned about the overall performance, availability and the overall attack surface of your estate?

Although you have no issue with the usage of the third party images so long as they are approved for usage, pulling these images from a Public Image Registry can have several issues. First, many images are quite large and take significant time to pull. Second, there are SLA's on the overall solution including the CI/CD pipeline. Finally, there are SLA's on the time to recover in a disaster recover scenario that must also be met.

In addition, in many situations, Security is paramount and locking down external access for container hosts is important. In these situations, Auditing and governance of which container images (and versions) are commonly used across the estate

Therefore:

Configure your *Private Image Registry* to proxy all images from the public repositories rather than allowing the container hosts to pull the images directly from the *Public Image Registry*.

Using a *Public Registry Proxy* will increase the overall availability of your solution as all container images are cached locally on your container registry meaning that if you need to pull your container (such as a restart of the container host) then the local server cache does not need to pull the image from the *Public Image Registry*. In addition this increases the security of your environment as access to the outside world can be locked down (i.e. each server will not require a connection to the public hub). Finally this will increase the overall availability of the solution, as there are fewer dependencies on external clouds.

3. DISCUSSION

Now that you understand the base Docker build pipeline patterns, you can see how these patterns could be combined and extended into an overall reference architecture for providing Image Management Services. This is an emerging area for pattern mining for us.

REFERENCES

[7 Threats] "7 Docker Security Vulnerabilities and Threats." Sysdig. June 25, 2018. Accessed July 09, 2018. https://sysdig.com/blog/7-docker-security-vulnerabilities/.

[Baset] Baset, Salman. "Unraveling Docker Security: Lessons From a Production Cloud." LinkedIn SlideShare. December 29, 2016. Accessed September 16, 2018. https://www.slideshare.net/SalmanBaset/unraveling-docker-security-lessons-from-a-production-cloud-70513798.

[BobCares] "Easy Steps to Setup Your Docker Private Repository." Bobcares. June 29, 2018. Accessed July 09, 2018. https://bobcares.com/blog/docker-private-repository/.

[Brown 2016] "Patterns of Microservices Architectures", Pattern Languages of Programs Conference 2016 Proceedings, The Hillside Group, February 17, 2017, Accessed July 21, 2018, www.hillside.net/plop/2016/papers/proceedings/papers/brown.pdf

[CenterDevice] "Docker Registry or How to Run Your Own Private Docker Image Repository." Codecentric AG Blog. December 21, 2016. Accessed July 09, 2018. https://blog.codecentric.de/en/2014/02/docker-registry-run-private-docker-image-repository/.

[Clair] Coreos. "Coreos/clair." CoreOS Clair Documentation. May 07, 2018. Accessed July 09, 2018. https://github.com/coreos/clair.

[Datanyze] Datanyze Containerization Market Share. Accessed July 09, 2018. https://www.datanyze.com/market-share/containerization.

[Development Pipelines] "Docker Reference Architecture: Development Pipeline Best Practices Using Docker EE." Docker - RBAC Example Use Case. Accessed July 09, 2018. https://success.docker.com/article/dev-pipeline.

[Docker Bench] Docker. "Docker-bench-security Documentation." GitHub. July 01, 2018. Accessed July 09, 2018. https://github.com/docker/docker-bench-security.

[DockerBlog] "It's Here: Docker 1.0." Docker Blog. June 09, 2014. Accessed August 05, 2018. https://blog.docker.com/2014/06/its-here-docker-1-0/.

[Jenkins], "Jenkins User Documentation", September 16, 2018, Accessed September 16, 2018,. https://jenkins.io/doc/

[Kubernetes Devops] Ibm-cloud-architecture. "IBM Cloud Native Devops Reference Architecture for Kubernetes." GitHub. June 28, 2018. Accessed July 09, 2018. https://github.com/ibm-cloud-architecture/refarch-cloudnative-devops-kubernetes.

[Labourdy] "Highly Available Docker Registry on AWS with Nexus", Mohamad Labourdy, Personal Blog, Accessed July 21, 2018, http://www.blog.labouardy.com/highly-available-docker-registry-on-aws-with-nexus/

[Macadamian] "Creating a Private Docker Registry." Macadamian. February 07, 2017. Accessed July 09, 2018. http://www.macadamian.com/2017/02/07/creating-a-private-docker-registry/.

[Portworx Survey] "2017 Annual Container Adoption Survey: Huge Growth in Containers." Portworx. April 14, 2017. Accessed August 05, 2018. https://portworx.com/2017-container-adoption-survey/

[Portworx] Portworx, "Run the Docker Registry with High Availability", Portworx documentation, Accessed July 21, 2018, https://docs.portworx.com/applications/docker-registry.html

[Quay.io], Quay.io, Accessed October 6, 2018, https://quay.io

[Sumologic] Tozzi, Chris. "Docker Security - 6 Ways to Secure Your Docker Containers." Sumo Logic. August 02, 2017. Accessed July 09, 2018. https://www.sumologic.com/blog/security/securing-docker-containers/.

[SoftwareSecured] "Choosing a Vulnerability Scanner | Software Secured Blog." Software Secured. June 27, 2018. Accessed July 09, 2018. https://www.softwaresecured.com/choosing-a-vulnerability-scanner/.

[Synopsis] https://www.synopsys.com/blogs/software-security/container-adoption-numbers/

[Vulnerability Advisor] Isci, Canturk. "Vulnerability Advisor Comes to Your Cloud with IBM Cloud Private." Medium. April 13, 2018. Accessed July 09, 2018. https://medium.com/ibm-cloud/vulnerability-advisor-comes-to-your-cloud-with-ibm-cloud-private-38a6afeab302.