tripwire

WHITE PAPER

Securing the Entire Container Stack, Lifecycle and Pipeline

FOUNDATIONAL CONTROLS FOR SECURITY, COMPLIANCE & IT OPS

With the rise in popularity of containers, development and DevOps paradigms are experiencing a massive shift, and security admins are increasingly left struggling to figure out how to secure this new class of assets and the environments they reside in. While containers increase the complexity of the ecosystem that security admins are responsible for securing, by shifting to horizontal scaling models containers offer new techniques to quickly remediate vulnerabilities and to increase the odds that compromises are benign—or at least short-lived. However, introducing good security hygiene into the container ecosystem is not a simple task. It entails integrating security into the container lifecycle and ensuring that security is considered and implemented at each stage of the container pipeline. This paper will cover all the high points of what it means to secure the entire container stack, lifecycle and pipeline, focusing on a couple technologies and use-cases to help limit its scope.

Docker Containers & SaaS

While there are multiple container technologies available today, this paper addresses Docker specifically, since it's one of the most widely used. In addition, while containers can be used in many ways, one of the most popular uses is in SaaS applications, and so this paper will focus on that environment. Even if you aren't using containers for a SaaS application, there is a lot you can take away from this paper and adapt for your own purposes.

As you read this paper, keep in mind that you should always perform basic security hygiene for any of the host systems running any of the components involved in your containers or their pipeline. This means for example performing vulnerability assessment and remediation, assessing for compliance against security benchmarks like NIST and CIS, and monitoring critical system files.

What Are Containers?

Before you can really begin to understand how to secure containers, you need to understand what they are and what benefits they provide. At a high level, containers are lightweight, self-contained application bundles that include everything you need to run them—from the code, to the runtime, system tools, system libraries and settings.

Containers are not virtual machines. Unlike virtual machines, containers share the kernel of the host OS they're running on, so the size of containers can be incredibly small. Also, unlike virtual machines, containers start up almost instantly. They also share devices and file system access of the host OS. Because of how lightweight they are, you can spin up many more containers on a host OS than you would if they were hosted as virtual machines.

If used properly, containers provide a large degree of isolation from both the host OS and from other containers running on the system. This can allow you to isolate different applications or application components from one another.

Containers themselves are instances (running or stopped) of container images. Images consist of multiple image layers. The bottom layer is usually a parent image layer which includes the OS (such as Ubuntu, CentOS or Alpine Linux). Each subsequent layer in the image is a set of differences from the layers before it. All image layers are read-only.

When you spin up a container, it creates a new writable layer on top of all the layers in the image, and any changes to the running container occur in this layer.

The Container Lifecycle

It's going to be very difficult to properly secure your containers if you don't understand how they progress through their lifecycles at your organization.

Conceptually, the lifecycle is simple. It consists of three phases: build, distribute and run. As a security professional, you must understand the container lifecycle at your organization and dig into each

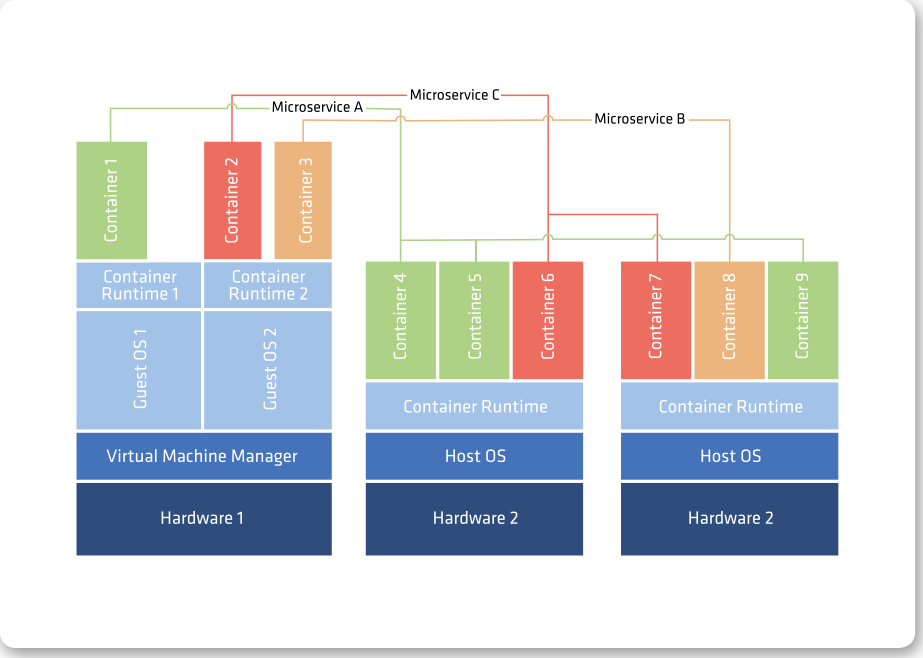

Fig. 1 Unlike virtual machines, containers are incredibly small, can start up almost instantly, and they share the kernel of the host OS

of these phases at a granular level. You need to identify what is built and how are they built:

» How are containers distributed?

» How are containers run?

» What's the pipeline of tools and components that your containers progress through—and are managed by—throughout their lifecycles?

Examining all of these things will be key to determining what your current container security posture is so that you can identify gaps and propose changes to existing tools or processes. Sit down and map out the lifecycle for your own organization, including the specific tools and technologies that are used to build, distribute and run them.

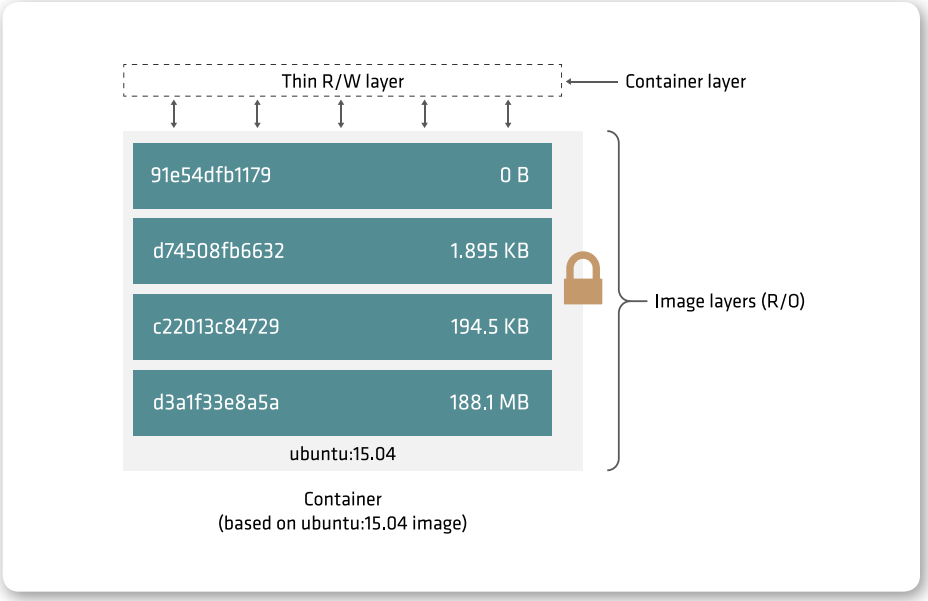

Fig. 2 Images consist of multiple read-only image layers with a writeable layer on top

Example Container Lifecycle

Figure 4 gives an example of a pipeline-based build approach, which means that container creation, distribution and deployment is automated from the moment the code is checked in to when it is deployed to production. The blue lines in the diagram follow the container through the pipeline.

The first step in any container pipeline is going to be the initial code check-in. From there, a continuous integration tool such as Teamcity or Jenkins will pull a parent OS image from an image registry, build a new image by adding additional layers on top of it, and then push the image to a production image registry. Once in an image registry, a container orchestrator such as Kubernetes can pull the image from the registry and deploy it onto a container host running on any platform (such as into an AWS EC2 instance).

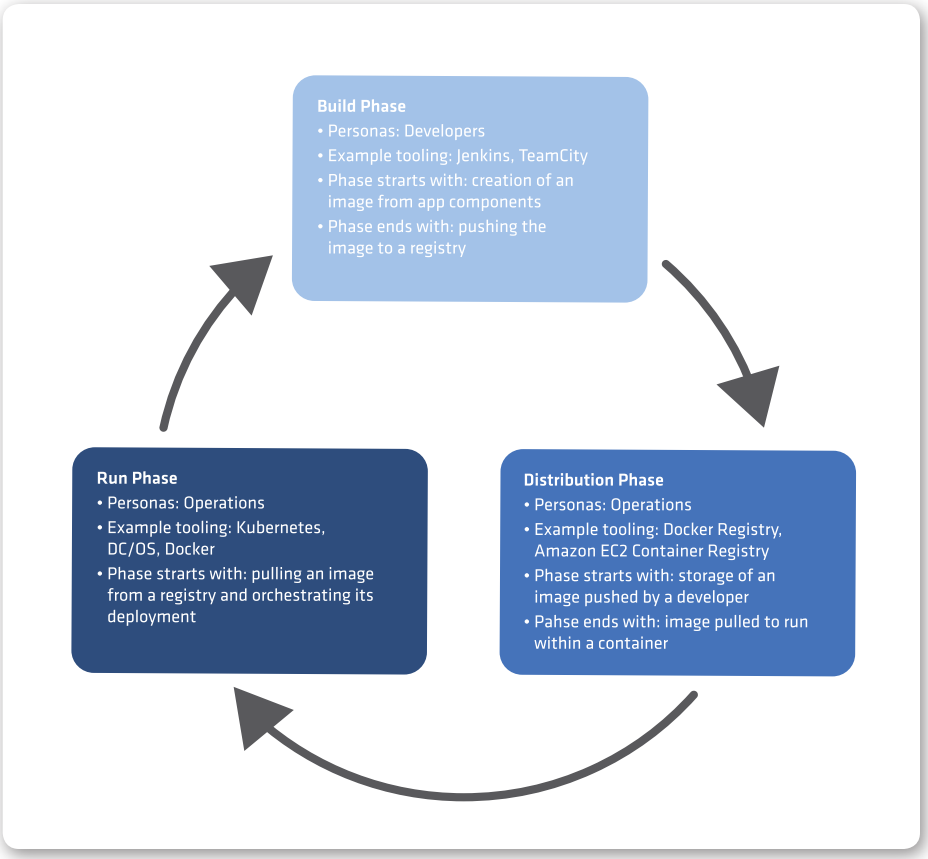

Fig. 3 The container lifecycle: Build, Distribute and Run

This may not be how your own container pipeline looks, if yours is even automated at all, but an automated container pipeline such as this is a solid foundation that will allow you to incorporate automated security controls at each key component to reduce risk and increase your confidence level of the containers you have running in production.

Incorporating Security Into the Pipeline

The first thing you can do is bring a security assessment tool into the build process, as shown by the Security Scanner box in Figure 5. Instead of having your continuous integration tool build your image and immediately push it into a registry, the image should first be pushed into a security tool that can assess that image for vulnerabilities and misconfigurations. Based on a policy of your organization's choosing, the image can be passed or failed, with only passed images getting pushed into the production-ready image registry. This ensures that security assessments take place at the earliest stages of your container development and that at-risk containers never have a chance to make it out into production.

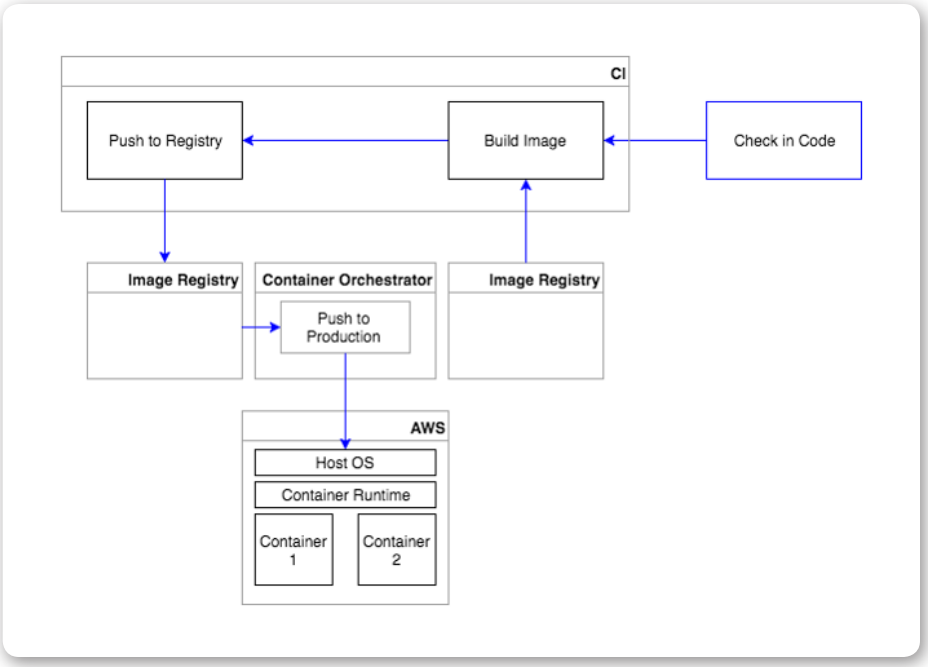

Fig. 4 Example of a pipeline-based build approach

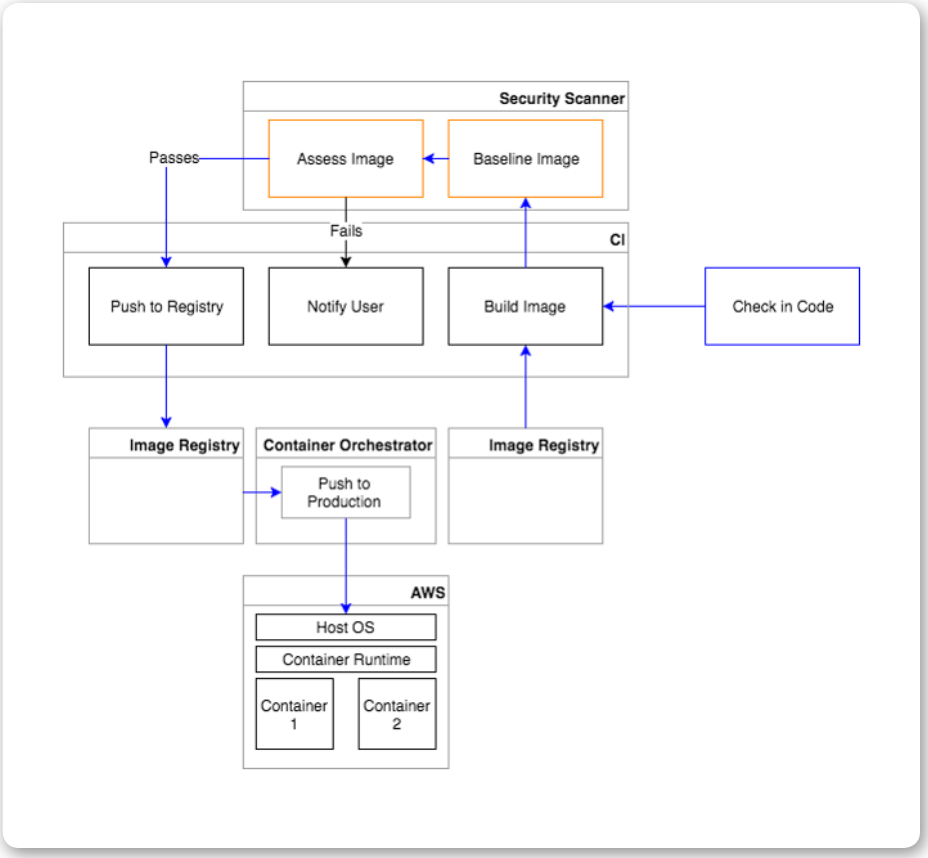

Fig. 5 Assessing images before production

Container Registries

As a part of your build process, the bottom parent image layer will be coming out of a registry. Instead of pulling from a public registry, consider using a private registry that only has images that have been approved by your organization and pre-assessed for vulnerabilities and misconfigurations as shown by the red lines in Figure 6. Once the final image has been built and assessed, it will go into another private image registry.

Ensure that you're using secure private registries by requiring that all connections to them be SSL-enabled to protect your images while in transit. Set up authentication on your registries as an additional layer of protection. Use Docker Content Trust as well, so clients pulling images from your registries can validate the images and ensure they haven't been tampered with.

You may also want to consider a security assessment tool to continually monitor your private registries to ensure that the only images that exist in them are those that have been assessed and approved.

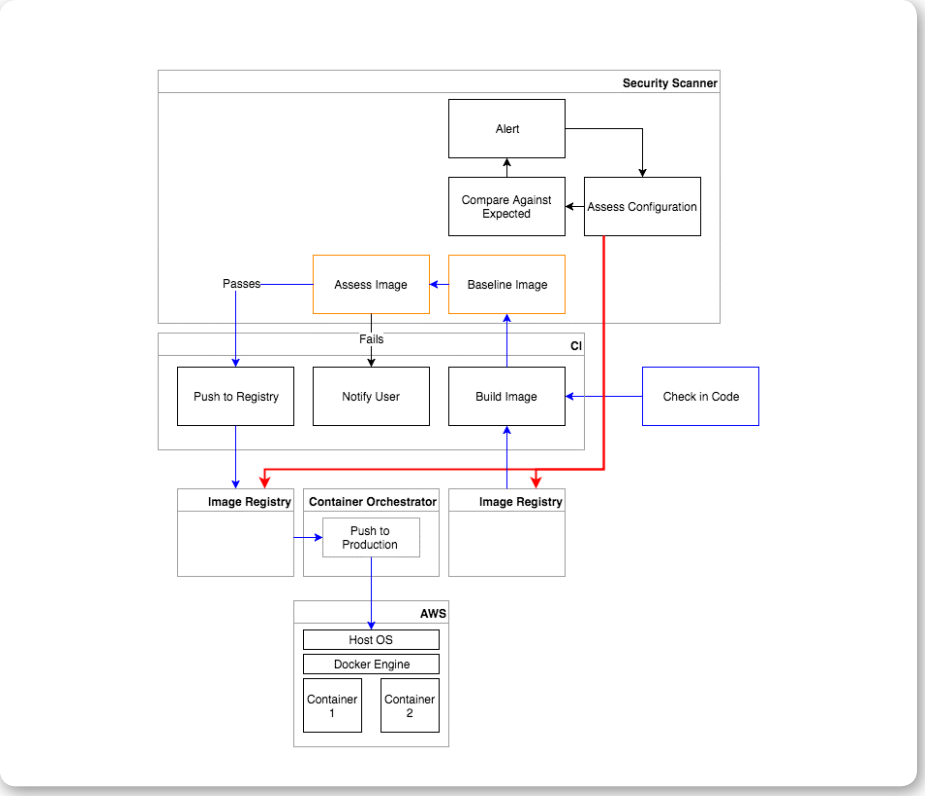

Fig. 6 Consider using a private registry that only has pre-approved images

Continuous Integration

Continuous Integration (CI) tools are often overlooked by security teams as a critical component in the container pipeline. Your CI tool will potentially have access to your code repository, production container registries, access to the security assessment tool, and potentially even access to your cloud platform itself.

To give an example of the dangers that could exist here, a new attack vector was presented by SpaceB0x at Defcon 25, where he was able to use a victim's public GitHub repository to gain access to their Azure environment and deploy new networks within it without any interaction from the victim. He did this by submitting a pull request to their public GitHub repository that contained some changes to their build scripts, which triggered a build automatically via a webhook, thereby executing his code and gaining root access to their build system, which contained credentials for Azure. He even created a tool called CIDER (Continuous

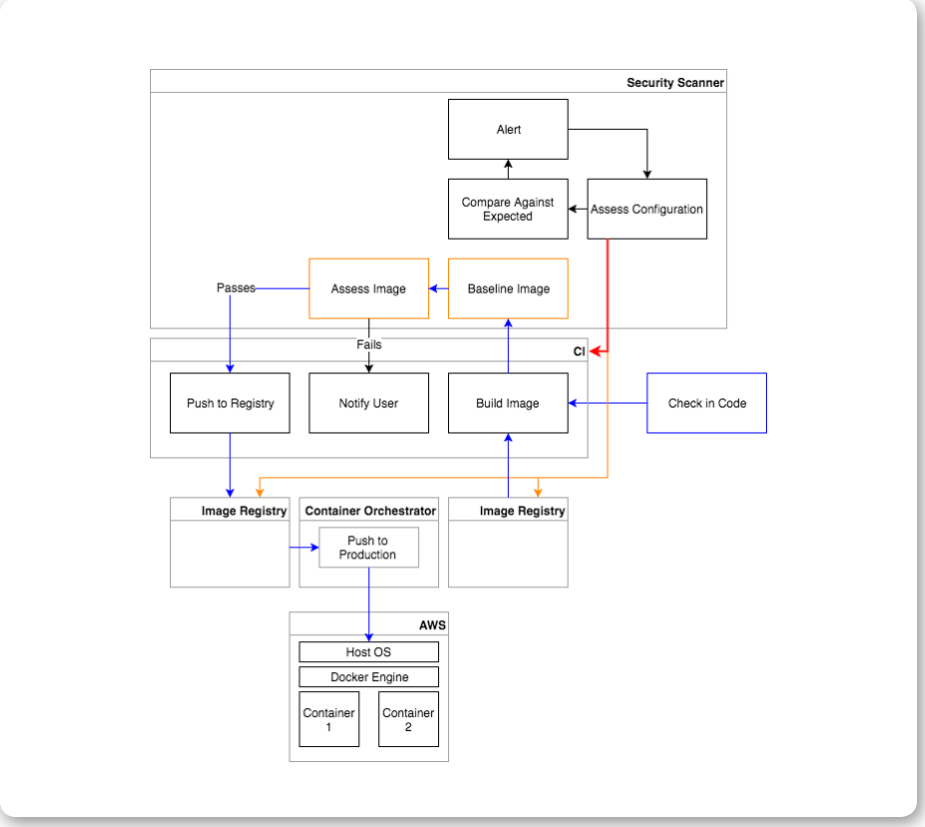

Fig. 7 Continuous Integration tools are often overlooked by security teams as a critical component in the container pipeline

Integration and Deployment Exploiter), to automate exploiting and attacking CI build chains.

What lessons can we take away from this? Aside from not triggering builds of code that haven't been code reviewed (especially from unapproved users), don't use your CI tool for container deployment. Use it to build images and push them to a registry, and then use a separate orchestrator to pull those images and deploy them. This way your CI tool will be incapable of making API calls to your cloud platform.

Don't use webhooks to start builds of code that have not yet been code reviewed. While you probably aren't using a public repository and are using some form of access control on your code repo, there's still risk of building code automatically that hasn't been code reviewed. For a further layer of isolation from your production environment, don't use your CI tool for container deployment. Use it to build images and push them to a registry, and then use a separate orchestrator to pull those images and deploy them; this way your CI tool will be incapable of making API calls to your cloud platform.

Outside of basic security hygiene and Role-based Access Control (RBAC), you can also monitor the CI tool's build jobs for integrity. A change to a build job could inadvertently cause the security assessment process to be bypassed in some way—or a new build could be created or an existing job modified—that pushes unwanted images into your production image registry.

Container Orchestrator

The container orchestrator will be responsible for pulling containers out of your container registry and pushing them out into production, among other things. A compromised orchestrator could mean anything from unwanted containers running in production, to containers running in an insecure configuration.

You can assess your orchestrator against benchmarks such as CIS to ensure that it's configured securely, as shown by the red line in Figure 8. Orchestrators such as Kubernetes typically have tools you can use to further secure your container environment, so it's important that you learn about best practices for your tool of choice. For example, Kubernetes provides the ability to create administrative boundaries between resources, set up network segmentation, and apply security contexts to your containers.

As an example of how you might want to use Kubernetes to further secure your containers, NIST recommends assigning sensitivity levels to your containers, and only have containers of the same sensitivity level running on the same host OS kernel to limit the impact of a compromised container. You wouldn't want a container handling financial data to be running on the same host OS as one running the publically exposed web UI. Kubernetes provides the ability to apply labels to pods (a group of one or more containers) and nodes (a worker machine like a VM or physical machine). These labels are essentially key-value pairs. You can use these labels to assign a sensitivity level to your nodes and pods, and ensure that your pods are only deployed onto nodes with the proper sensitivity levels using nodeSelectors or node affinity.

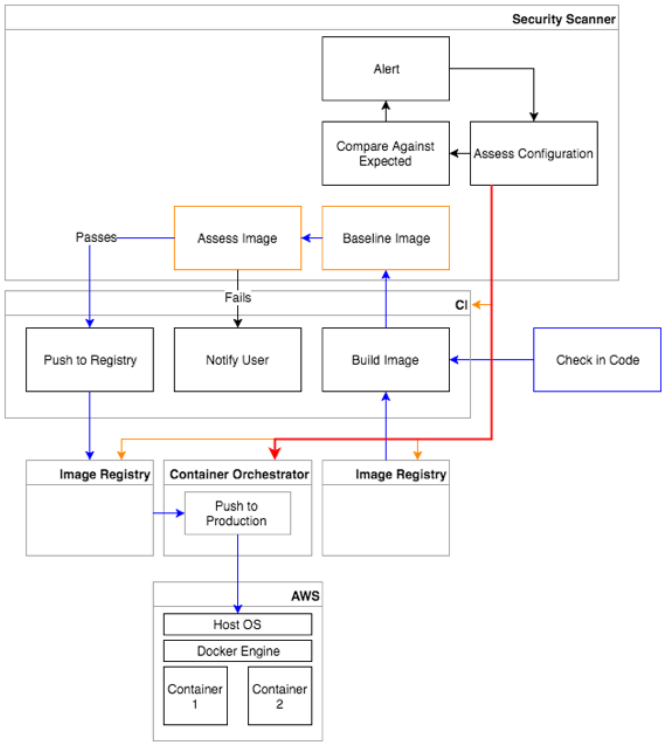

Fig. 8 Assess your orchestrator against benchmarks to ensure that it's configured securely

Securing the Stack

The next step is ensuring that you've secured the entire container stack, meaning all the layers or components involved with a running container on a Host system. As shown by the red lines in Figure 9, this means securing:

- The platform itself, whether that's your AWS or Azure environment
- Host OSes (such as Alpine Linux) running on the platform
- Container technology itself, including the Docker daemon and the Docker container runtime
- Container images and Docker files

Securing the Platform

Securing the platform means ensuring that your AWS or Azure accounts are configured securely. Ideally, you would use an automated assessment tool that can continually assess your accounts to ensure they are in compliance with best-practices and standards. In the case of AWS, there is a CIS Amazon Web Services Foundations policy available (Figure 10) that you can use as a guideline.

Secure the Host OS

To reduce the attack surface as much as possible, your host OS should be designed for the singular purpose of running containers—in other words, it should be lean. This means no services running and no packages installed that aren't specifically used for running your containers. If you're using a traditional Linux OS, this means pairing back services and packages manually, or you could instead use a lightweight Linux OS (such as Alpine Linux) that is specifically designed to be minimal, secure and efficient. There are also purpose-built Linux OSes designed expressly for

One small error...

The Verizon data breach that leaked personal data of six million customers was enabled by the misconfiguration of just one of their S3 buckets, causing it to be publically exposed. This effectively illustrates the fact that a single, simple misconfiguration in your cloud platform can result in catastrophic compromises to the services and data you have hosted there.

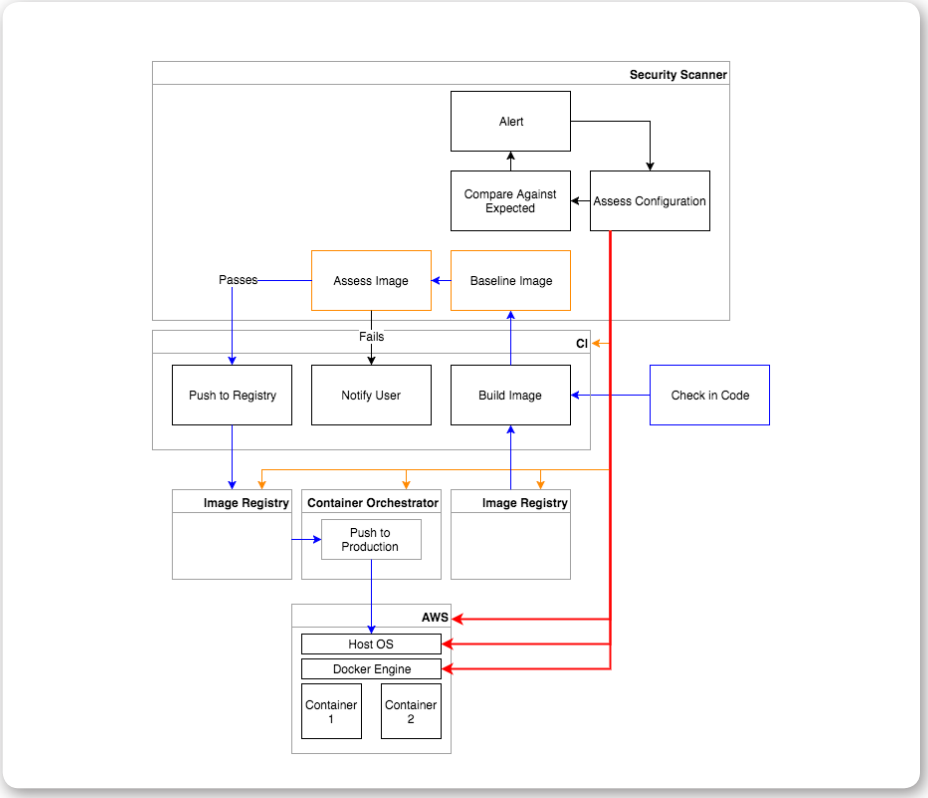

Fig. 9 Secure the entire container stack

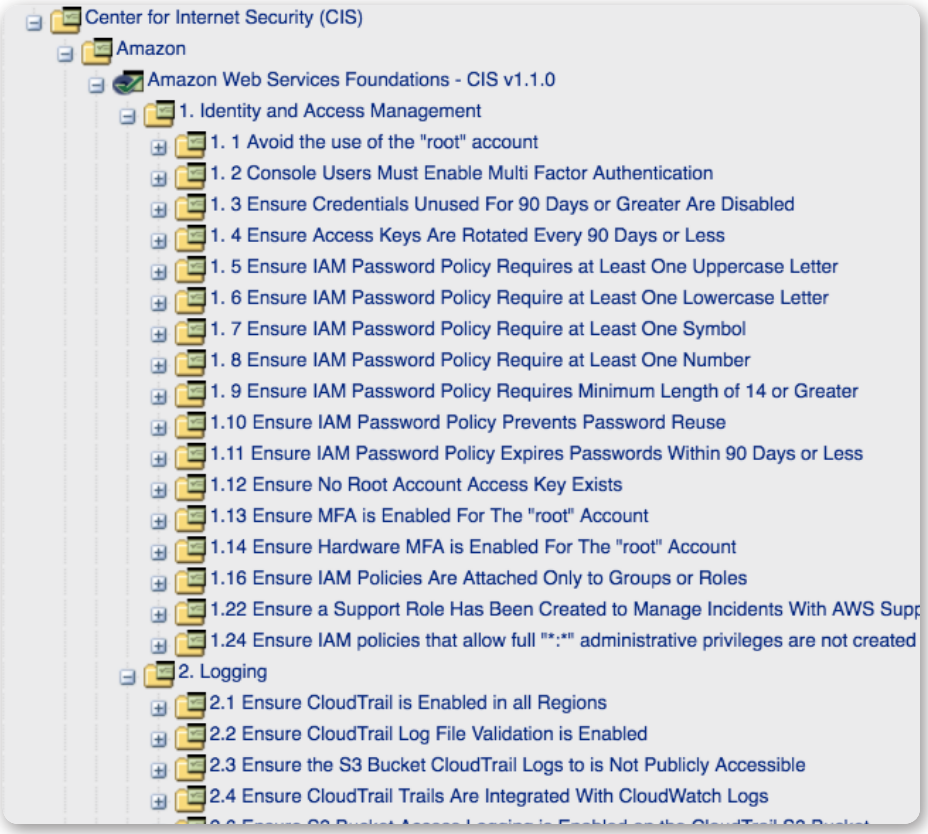

Fig. 10 The CIS Amazon Web Services Foundations policy in Tripwire® Enterprise

running containers (such as CoreOS and RancherOS) that may be worth investigating.

Securing Docker

Docker itself is composed of multiple components, and these also need to be hardened. CIS has worked with the Docker community to create a benchmark policy (Figure 11) that includes best practices for securing both the Docker daemon and container runtime, among others.

There are far too many recommendations in the policy to go through them all, but daemon examples include:

» Restricting network traffic between containers

» Ensuring that the Docker daemon socket is secured

» Using an authorization plugin to configure granular access policies for managing access to the Docker daemon

For the container runtime, examples include:

» Making sure you don't mount sensitive host system directories on containers

» Not sharing various host namespaces with containers to keep them isolated

» Making sure that containers are running under defined cgroups

Use automated security assessment tools to continually assess Docker to ensure that you've configured it securely and that it remains secure.

Take an Immutable Approach

You may be wondering, how do you fix newly discovered vulnerabilities or misconfigurations in your running containers? That is, how do you make changes to your containers?

Ideally, you would strive to take an immutable approach to your container strategy. This means never making changes directly to your running containers. Don't change configuration settings, don't install new packages, and don't upgrade existing packages (even to fix a security vulnerability!). The containers you have running in production should be exactly what you expect them to be based on the images that went through your container pipeline. Any change you need to make to a container in production should involve building a new image, so that the image can make its way back through the pipeline, through the security assessment process, and out into production.

Fig. 11 Docker CIS Benchmark in Tripwire Enterprise

Consider trying to make your containers read-only. You can pass a read-only flag to your Docker run command to create a read-only container, and you can use the tmpfs flag to mount an empty temporary filesystem for specific directories that may need write access for your application to function properly.

It may even be prudent to track the uptime of your running containers in production, as containers that are up for too long have a higher chance of having drifted (changed in some way), and change means uncertainty, which means risk. To keep your running containers fresh, consider periodically destroying and replacing them with new spun up versions. This way they only have a limited amount of time to drift before they're reset back into a known fresh state. This also means that any compromise that might have occurred to them would be short-lived.

Check for Drift

Once you have running containers in production, you need to continually assess those containers to check for drift (Figure 12), even if you are limiting their uptime by destroying them periodically.So what does "drift" mean? Running containers are based on images that have already gone through an automated security assessment process. This means knowing exactly what vulnerabilities exist in that container, and know how compliant that container is with the standards you care about. If a container has been running for 30 minutes and now all of a sudden a new vulnerability appears, or a drop in compliance has occurred, that means something significant has changed. That's cause for concern. Even if you see your compliance score increase, or a known vulnerability disappear, that too would be a cause for concern. If an immutable container has drifted from the known baseline in any way, you can investigate why and how that has occurred.

Conclusion

This paper has touched on different components that you can take into consideration when securing container stacks, lifecycles and pipelines. Securing the entire container stack includes assessing container images as a part of the build process, continuously assessing running containers for drift, the importance of your continuous integration tools, container registries, container orchestrators, your cloud platform, host OSes, the Docker engine, and taking an immutable approach to your container modification strategy.

Any one of these topics could be a white-paper in and of itself, but hopefully now you are thinking far beyond securing just the containers so that you can start the process (if you haven't already) of mapping out your own container lifecycles and pipelines so you can begin developing a more comprehensive strategy for securing your containers.

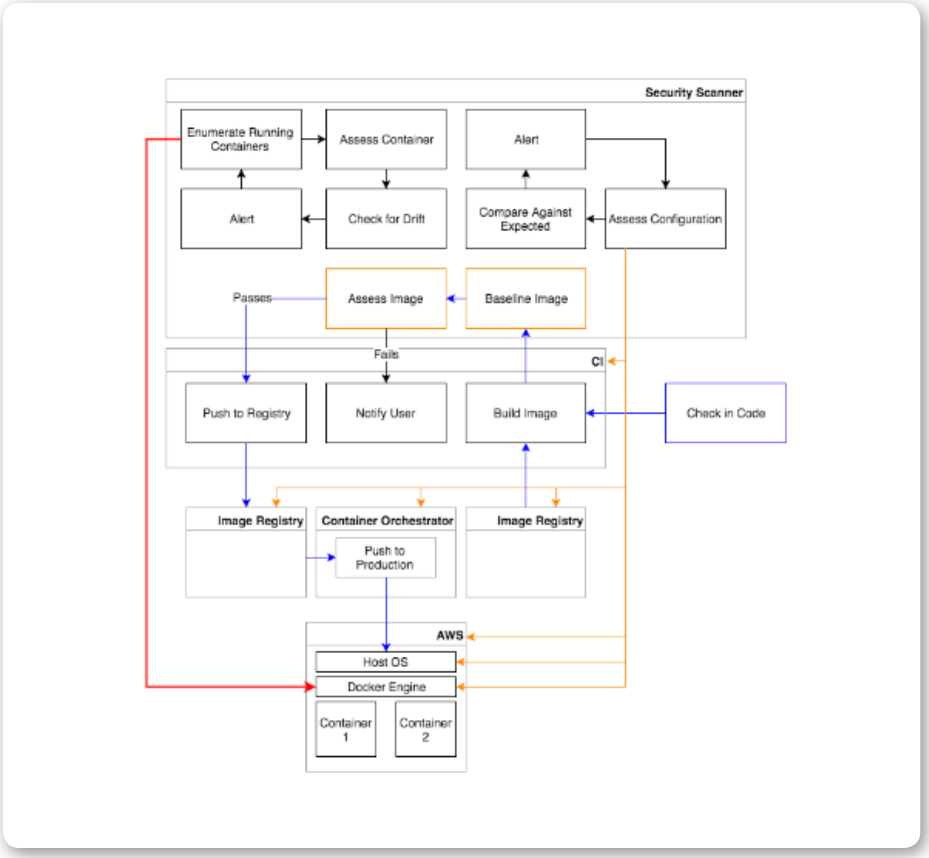

Fig. 12 Continually assess containers for drift

How Tripwire Can Help

Tripwire's foundational controls enable effective and efficient security, compli¬ance and IT operations across physical, virtual, and private and public cloud hybrid environments. These advanced controls provide robust, industry-leading capabilities including file integrity monitoring, configuration management, asset discov¬ery, vulnerability management and log collection. Tripwire addresses the unique needs of DevOps environments with unified management and support for container assessment, CIS benchmarks for Docker and AWS, elastic monitoring, and integrations with Puppet and Chef.

Tripwire is a leading provider of security, compliance and IT operations solutions for enterprises, industrial organizations, service providers and government agencies. Tripwire solutions are based on high-fidelity asset visibility and deep endpoint intelligence combined with business context; together these solutions integrate and automate security and IT operations. Tripwire's portfolio of enterprise-class solutions includes configuration and policy management, file integrity monitoring, vulnerability management, log management, and reporting and analytics. **Learn more at** tripwire.com

The State of Security: Security News, Trends and Insights at tripwire.com/blog
Follow us on Twitter @TripwireInc » **Watch us at** youtube.com/TripwireInc

WPSECSLPj 201710