Container image security: Going beyond vulnerability scanning

Introduction

Container images constitute the standard application delivery format in cloud-native environments. The wide distribution and deployment of container images requires a new set of best practices for ensuring their integrity. While performing image scans to check for known vulnerabilities in operating systems and language packages remains a cornerstone of image security, it is only part of a larger set of security initiatives you need to protect your environments. Understanding the risks at each stage of a container's life cycle will inform decisions around image infrastructure and handling to enhance and maintain your organization's security.

Build

Continuous integration (CI) build infrastructure

If a container image already contains malicious software, it poses an immediate threat at runtime. Security in your CI build infrastructure is as important as it is in your production environment to prevent external vulnerabilities from being introduced into your images. Take these important actions to help secure your build infrastructure and pipelines:

- Limit administrative access to the build infrastructure.
- Allow only required network ingress.
- Manage any necessary secrets carefully and grant only minimal required permissions.
- Carefully vet any third-party sites from which source or other files get pulled, using network firewalls to allowlist access only from trusted sites.
- While using a vulnerability scanner on the resulting images should be standard practice, you also need to ensure the scanner's security and reliability to get trustworthy results.

Base image

If you are building images on an existing, third-party base image, you want to select for a few factors:

- **Trustworthiness** of the base image's source and host. Confirm that the image comes from a known company or open-source group, is hosted on a reputable registry, and that the Dockerfile and source code for all components in the image are available.
- **Update frequency.** Avoid images that are updated infrequently, especially if they do not respond to relevant vulnerability disclosures.

- **Default installed software.** Starting with a minimal base image and selectively installing the tools and libraries your application requires is more reliable than figuring out what packages you can safely remove from an existing image.

You can choose from an increasing number of secure, minimalist base images. For a given central processing unit (CPU) architecture, any recent Linux® base image should run on any Docker installation or Kubernetes container runtime, regardless of the cloud provider or platform. A few options are:

- Google Distroless: Extremely minimal base images, prebuilt for a few common languages. They lack a package installer, so if you need additional software, you can copy files into the image.
- Red Hat® Universal Base Image (UBI): Red Hat Enterprise Linux-based image available without a Red Hat subscription. This set of images comes in three tiers—minimal, standard platform, and multi-service. It also has images for multiple languages.

Alternatively, you can build an original base image.

Exploitable and nonessential software

If malicious attackers do manage to gain access to your running container, you want to give them as few tools as possible to exploit once they get in. Using a minimal base image offers a good start, but if your Dockerfiles all install a standard set of multipurpose tools, you lose the advantage. Keeping the image as minimal as possible has the added benefit of reducing the probability of encountering zero-day vulnerabilities—which will need patching—and keeping the image smaller, which makes storing and pulling it faster.

Restricting the image to required binaries, libraries, and configuration files provides the best protection. In particular, avoid installing the following tools (or remove them if present):

- Package managers such as apt, yum, apk.
- Unix shells such as sh, bash. Removing shells also prevents the use of shell scripts at runtime. Instead, use a compiled language when possible.
- Compilers and debuggers. These should be used only in build and development containers, but never in production containers.

If you install these tools in your production images to perform application debugging, you may want to reconsider your workflows and practices. At minimum, consider creating temporary debugging images when issues that cannot otherwise be diagnosed arise. Kubernetes (as of version 1.16) has alpha support for ephemeral containers, which can be placed in an existing pod to facilitate debugging.

Build versus runtime containers

The build tools you use to generate and compile your applications can be exploited when run on production systems. Remember, containers should be treated as temporary, ephemeral entities. Never plan on patching or altering a running container. Build a new image and replace the outdated container deployments. Use multistage Dockerfiles to keep software compilation out of runtime images.

```
FROM build-image as build
# Build my stuff
# ...
FROM base-image
# Install my packages
# ...
COPY --from=build /out/my-app /bin/my-app
```

Secrets

Never embed any secrets in your images, even if the images are for internal use. Secrets include transport security layer (TLS) certificate keys, cloud provider credentials, secure shell protocol (SSH) private keys, and database passwords. Anyone who can pull the image can extract the secret. Supplying sensitive data only at runtime also enables you to use the same image in different runtime environments, which should use different credentials. It also simplifies updating expired or revoked secrets without rebuilding the image.

As an alternative to embedded secrets, supply secrets to Kubernetes pods as Kubernetes secrets, or use another secret management system.

Image scanning

Images that contain software with security vulnerabilities become vulnerable at runtime. When building an image in your CI pipeline, image scanning must be a requirement for a passing build run. Unsafe images should never get pushed to your production-accessible container registry.

While a number of open source and proprietary image scanners are available in addition to cloud scanning services, they do not all provide the same level of coverage. Many scanners check only installed operating system packages. Others may also scan installed runtime libraries for some programming languages. Some may also provide additional binary fingerprinting or other testing of file contents.

When choosing a scanner for your CI pipeline, make sure it provides the coverage you need and supports the base image's package installer database and the programming languages your applications use. You will also need to determine acceptable levels of risk for allowing a build to pass such as any vulnerability below a certain severity—or alternatively, failing builds with fixable vulnerabilities above a certain severity. Make sure your scanner offers a compatible application programming interface (API) or tool that you can plug into your CI pipeline and provides the data you need to evaluate your criteria for failing a build.

Store

Choosing a registry

Once you have built your secure container image, you need to store it somewhere. Using a private, internal registry affords the greatest potential for security and configuration, but it requires carefully managing the registry's infrastructure and access controls. Most cloud providers also offer managed registry services that use the cloud's access management service.

Registries offered as a service exist and support private repositories, and they ease much of the administrative overhead. Security engineers and build engineers will need to choose the best solution for their organization based on their security requirements and infrastructure resources.

Image controls

Some registries support additional controls concerning image identity.

Registries that support using immutable tags on images, preventing the same tag from being reused on multiple versions of a repository's image, enforce deterministic image runtimes. Many audited certifications and site reliability teams need the ability to know exactly which version of a given image, and therefore of an application, is deployed at a given time, which is impossible when every image pull uses the latest tag.

Image signing support provides even greater protections. With image signing, the registry generates a checksum of a tagged image's contents and then uses a private cryptographic key to create an encrypted signature with the image metadata. Clients could still pull and run the image without verifying the signature, but runtimes in secure environments should support image verification requirements. Image verification uses the public key counterpart of the signing key to decrypt the contents of the image signature, which can then be compared to the pulled image to ensure the image's contents have not been modified.

Run

Image scanning

While you should scan your images as part of your standard CI process, build-time scanning does not make runtime scanning unnecessary. Instead, runtime scanning is more important for any third party image you may use and for your own images, which may contain newly discovered security vulnerabilities. You can use custom or third-party admission controllers in Kubernetes clusters to prevent the scheduling of insecure container images.

While some scanners support storing scan output in a database or cache, users will have to weigh their tolerance for outdated information against the latency introduced by performing a real-time scan for each image pull. See the "Image scanning" subsection in the "Build" section of this whitepaper for guidelines for choosing and using an image scanner.

Registry and image trust

In the "Store" section, we discussed the criteria for choosing a registry and some additional security features supported by some registries. While configuring and using a secure registry is important, those protections are diminished or lost when they are not enforced on the client side.

Kubernetes does not offer native support for using secure image pull options. You will need to deploy a Kubernetes admission controller that can verify that pods use trusted registries. For signed image support, the controller would need to be able to verify the image's signature.

Maintain

Vulnerability management

Scanning an image throughout its life cycle is crucial, as is the need for weighing your organization's risk tolerance against maintaining velocity. Your organization will need to generate its own policies and procedures for handling image security and vulnerability management.

Start by defining your criteria for what constitutes an unsafe image, using metrics such as:

- Vulnerability severity.
- Number of vulnerabilities.
- Whether vulnerabilities have patches or fixes available.
- Whether vulnerabilities impact misconfigured deployments.

Next you need to define service-level objectives and procedures for handling these images. Do you want to set a deadline for building replacement images and deploying those to production? Should the deadlines vary by vulnerability severity? Do you want to block the scheduling of containers from existing images when a new vulnerability is discovered?

You also need to define procedures to handle containers with vulnerable images that are already running in production.

Further reading: Implementing Kubernetes-native security with Red Hat

Security platforms purpose-built to protect Kubernetes offer powerful security and operational advantages. Kubernetes-native security applies controls at the Kubernetes layer, ensuring consistency, automation, and scale. Organizations successfully deploy security as code, enabling security that is built in, not bolted on.

Download this whitepaper—Kubernetes-native Security: what is it and why it matters—to find out more about the key features and benefits of Kubernetes-native security and how it is different from existing container security approaches in delivering protections that are purpose-built for Kubernetes environments.