ITL BULLETIN FOR OCTOBER 2017

NIST GUIDANCE ON APPLICATION CONTAINER SECURITY

Ramaswamy Chandramouli, Murugiah Souppaya, and Karen Scarfone,[1] Editors
Computer Security Division
Information Technology Laboratory
National Institute of Standards and Technology
U.S. Department of Commerce

Introduction

NIST's Information Technology Laboratory has released Special Publication (SP) 800-190, *Application Container Security Guide* and NIST Internal Report (NISTIR) 8176, *Security Assurance Requirements for Linux Application Container Deployments*. Application container technology is increasingly being used to deploy, manage, and maintain applications. Transitioning from traditional application architectures to container-based implementations can have both positive and negative effects on an organization's security. NIST SP 800-190 and NISTIR 8176 are intended to help organizations understand the negative effects and provide practical recommendations for addressing them when planning for, implementing, and maintaining containers.

This bulletin offers an overview of application container technology and its most notable security challenges. It starts by explaining basic application container concepts and the typical application container technology architecture, including how that architecture relates to the container life cycle. Next, the article examines how the immutable nature of containers further affects security. The last portion of the article discusses potential countermeasures that may help to improve the security of application container implementations and usage.

Introduction to Application Container Technology

Operating system (OS) virtualization provides a separate virtualized view of the OS to each application on a server, keeping each application on the server isolated from all others. Each application can only see and affect itself. OS virtualization has become increasingly popular due to advances in its ease of use and a greater focus on developer agility as a key benefit. Today's OS virtualization technologies focus on providing a portable, reusable, and automatable way to package and run applications (apps). The terms *application container* or simply *container* are used to refer to these technologies.

Container architectures often divide an app into many components, each with a single well-defined function. Each app component runs in a separate container, and sets of containers called microservices work together to compose an app. An image is a package that contains all the files required to run a container. An image should include only the executables and libraries required by the app itself; all other OS functionality is provided by the OS kernel within the underlying host OS. With this approach, app deployment is more flexible and scalable. Development is also simpler because functionality is more self-contained. However, there are many more objects to manage and secure, which may cause problems for app management and security tools and processes.

Modern container technologies have largely emerged along with the adoption of development and operations (DevOps) practices that seek to increase the integration between building and running apps. The portable and declarative nature of containers is particularly well suited to these practices because they allow an organization to

[1] Karen Scarfone is a Guest Researcher from Scarfone Cybersecurity.

be consistent among development, test, and production environments. Organizations often use continuous integration processes to put their apps into containers directly in the build process, so that from the beginning of the app's life cycle, consistency of its runtime environment is guaranteed.

Container images are typically designed to be portable across machines and environments, so an image created in a development lab can be easily moved to a test lab for evaluation, and then copied into a production environment to run without modification. The downside of this approach is that the security tools and processes used to protect containers should not make assumptions about specific cloud providers, host OSs, network topologies, or other aspects of the container runtime environment, which may frequently change. More critically, security should be consistent across all of these environments and throughout the app life cycle, from development to test to production.

Figure 1 shows a simplified example of a typical container technology architecture, which shows its three life cycle phases: 1) image creation, testing, and accreditation; 2) image storage and retrieval; and 3) container deployment and management. These three phases are described below.

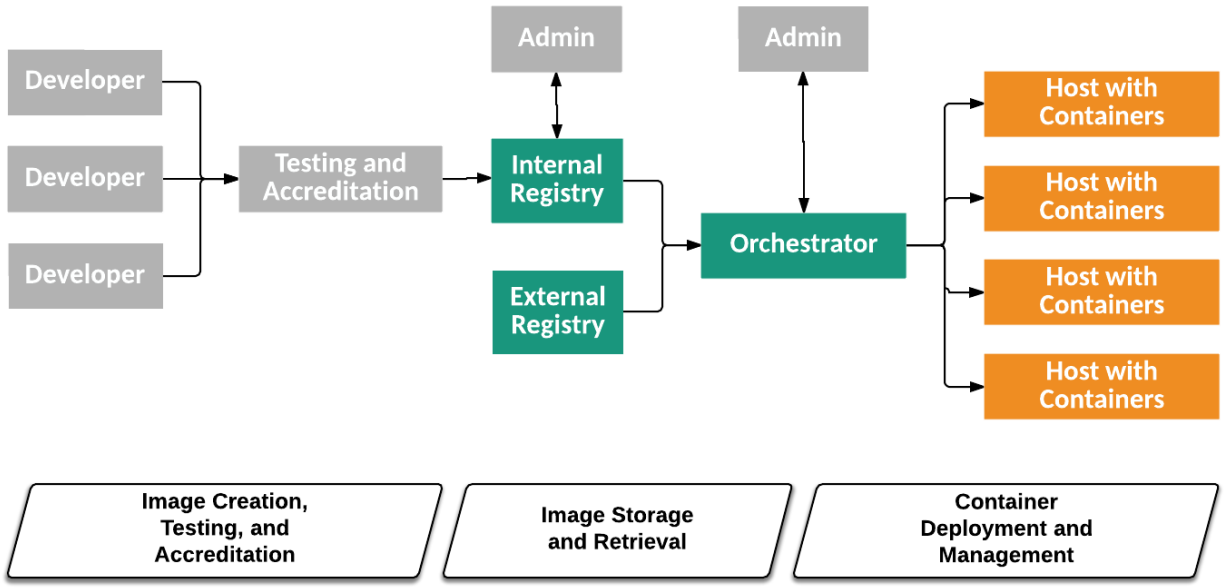

Figure 1. Typical Container Technology Architecture, which shows the three phases of a container's life cycle (bottom three boxes).

(1) Image Creation, Testing, and Accreditation: Software developers create and build app components, and then package them into one or more images. Image creation typically uses build management and automation tools to assist with the continuous integration process. These tools take the libraries, binaries, and other components of an app, perform testing on them, and then assemble images out of them based on a developer-created manifest.

After an image is created, organizations typically perform testing and accreditation. For example, test automation tools and personnel would use the images to validate the app's functionality, and security teams would perform accreditation on the same images. The consistency of building, testing, and accrediting exactly the same artifacts for an app is one of the key operational and security benefits of

containers. Once testing and accreditation has been successfully completed, the images are signed and sent to a registry.

(2) Image Storage and Retrieval: Images are typically stored in central locations to make it easy to control, share, find, and reuse them across hosts. Registries are services that allow developers to easily store images as they are created, tag and catalog images for identification and version control to aid in discovery and reuse, and find and download images that others have created.

Registries provide application programming interfaces (APIs) that enable automating common image-related tasks. For example, organizations may have triggers in the image creation phase that automatically push images to a registry once tests pass. The registry may have triggers that automate the deployment of new images. This automation enables faster iteration on projects with more consistent results.

(3) Container Deployment and Management: Tools known as orchestrators enable DevOps personas – or automation working on their behalf – to pull images from registries, deploy those images into containers, and manage the running containers. This deployment process is what results in a usable version of the app, running and ready to respond to requests. When an image is deployed into a container, the image itself is not changed—a copy of it is placed within the container and transitioned from being a dormant set of app code to a running instance of the app.

The abstraction provided by an orchestrator allows a DevOps persona to simply specify how many containers need to be running a given image and what resources, such as memory, processing, and disk need to be allocated to each. The orchestrator knows the state of each host within the cluster, including what resources are available for each host, and determines which containers will run on which hosts. The orchestrator then pulls the required images from the registry and runs them as containers with the designated resources.

Orchestrators are also responsible for monitoring container resource consumption, job execution, and machine health across hosts. Depending on its configuration, an orchestrator may automatically restart containers on new hosts if the hosts on which they were initially running failed.

The Immutable Nature of Container Operations

What most distinguishes container technologies from other technologies is the concept of immutability in their operations. Most container technologies intend for containers to be operated as stateless entities that are deployed but not changed. When a running container needs to be upgraded or have its contents otherwise changed, it is destroyed and replaced with a new container based on an updated image. This “continuous delivery” automation enables developers to simply build a new version of the image for their app, test the image, push it to the registry, and then rely on the automation tools to deploy it to the target environment.

Container automation enables developers and support engineers to make and push changes to apps at a much faster pace. Organizations may go from deploying a new version of their app every quarter to deploying new components weekly or daily. This approach also has significant potential security benefits because it enables organizations to build, test, validate, and deploy exactly the same software in exactly the same configuration. As updates are made to apps, organizations can ensure that the most recent versions are used by configuring their orchestrators to pull the most up-to-date version of each image from the registry.

This means that all vulnerability management, including patches and configuration settings, is typically taken care of by the developer when building a new image version. With containers, developers are largely responsible for the security of apps and images instead of the operations team. This change in responsibilities often requires much greater coordination and cooperation among personnel than was previously necessary.

The immutable nature of containers also has implications for data persistence. Rather than intermingling the app with the data it uses, containers stress the concept of isolation. Data persistence should be achieved not through simple writes to the container root file system, but instead by using external, persistent data stores such as databases or cluster-aware persistent volumes. The data containers use should be stored outside of the containers, so when the next version of an app replaces the containers running the existing version, all data is still available to the new version.

Security Countermeasures for Application Container Technology

To help organizations understand the security concerns regarding application container technology, NIST SP 800-190 identifies security threats to platforms hosting the containers as well as the technology components involved in building containers and storing them prior to launch. Taking into consideration the overall security implications for the entire ecosystem, the document recommends six levels of security countermeasures: image, registry, orchestrator, container, host OS, and hardware.

NISTIR 8176 builds on this information to examine potential security solutions that provide the necessary countermeasures, as well as the kind of security assurance requirements each solution should satisfy. Because security solutions for containers vary significantly based on the host OS, NISTIR 8176 is scoped to focus on Linux-based environments only, which enables detailed security assurance requirements to be defined.

There are too many potential solutions and associated security assurance requirements to cover in this bulletin, so here are two examples of such solutions, called hardware-based countermeasures and host operating system-based countermeasures, which are discussed in more detail in NISTIR 8176.

(1) **Hardware-Based Countermeasures**: Implementing a trusted computing model starts with measured/secured boot, provides a verified system platform, and builds a chain of trust rooted in hardware. This chain of trust then extends to bootloaders, the OS kernel, and the OS components to enable cryptographic verification of boot mechanisms, system images, container images, and other components. There are two approaches to a trusted computing model solution, and they both involve a combination of a hardware-based (physical) trusted platform module (TPM) and a software-based (virtual) TPM (called a vTPM). The difference between the approaches is where the vTPM is placed: in the Linux kernel or in a dedicated container.

Each of these approaches has different security assurance requirements:

- If the vTPM is placed in the Linux kernel and the kernel is completely trusted, containers can reliably attest to their own state.
- If the vTPM is placed in the Linux kernel and the kernel is not completely trusted, the hardware platform provider can sign an endorsement key stating that the TPM is trustworthy. This can then be extended by giving each vTPM instance its own endorsement key and using the hardware-based TPM to sign the endorsement keys.
- If the vTPM is placed in a dedicated container, the host OS can provide isolation between processes belonging to different containers through the namespaces feature. However, this approach provides less protection than if the vTPM were placed in the kernel because the kernel is more reliable in limiting access.

(2) **Host OS-Based Countermeasures**: The host OS should mitigate threats involving "escape" from a container, particularly to protect each container from all other containers on the same host. Many access control solutions are available in Linux environments. These solutions use kernel-loadable modules referred to as Linux Kernel Modules (LKMs). These solutions enforce access control policies in a variety of ways, such as labeling each container and only allowing actions authorized for that label, and specifying which system calls are available for an application within a container for interfacing with the underlying kernel. The goal of these LKMs is to provide another level of security checks on the access rights of processes and users beyond that provided by the standard file-level access control.

Security assessment requirements that need to be satisfied include the following:

- A user authorized to run applications in a container should not be allowed access to the LKM-based access control solutions.
- If supported, both a syscall whitelist (allowable system calls) and a blacklist (prohibited calls) should be generated for each container. Whitelists should be based on the type of application hosted in the container, the deployment situation, and the container size. Blacklists should include high-risk calls, such as ones that allow loading LKMs, rebooting the host, and triggering mount operations.
- Containers should be prevented from mounting/remounting sensitive directories and/or specific system directories critical to security enforcement (e.g., cgroups, procfs, sysfs).

Summary

NIST SP 800-190 and NISTIR 8176 provide information on the basics of application container technology, the potential security issues of implementing and using this technology, and the countermeasures that should be used and the security assurance requirements that should be met to address those issues. NIST SP 800-190 offers general guidance on the topic of application container technology security, while NISTIR 8176 complements NIST SP 800-190 by taking a more detailed look at Linux container security.

ITL Bulletin Publisher: Elizabeth B. Lennon
Information Technology Laboratory
National Institute of Standards and Technology
elizabeth.lennon@nist.gov