Interoperable

Mert Toslali[1], Jonathan Chamberlain[1], Felipe Dale Figeman[1], and Zhengyang Tang[1]

[1]BU EC528 - Cloud Computing

The Open Container Initiative (OCI) has been established to create a open standard for container use regardless of the runtime being used to manage the container. However, the OCI only specifies downloading image then unpacking that image into an OCI Runtime filesystem bundle. It does not standardize lifecycle management of the containers, thus each container implements lifecycle functionality in a different manner. It also does not ensure that consistent standards for the security of containers are present.

In this project, we will study the differences in popular runtimes Docker, containerd, and crio. Our focus will be on developing a service focused on ensuring that Center for Internet Security (CIS) Benchmarks for Docker are satisfied across other runtimes to ensure consistent application of security principles irrespective of container runtime differeces. Long term, the goal is to implement a lifecycle mangement solution that enables a common management framework for controlling containers across enviornments. By implementing a service to validate standard secuirty checks across runtimes, we intend to provide a Proof of Concept that such a common lifecycle management is possible.

Introduction

Virtualization of resources has emerged in recent decades as a means to run multiple OSes on the same hardware. This particularly serves a useful function as this allows multiple applications to coexist on the same server, enabling efficencies in computing such as server consolidation.

Traditional VMs virtualize hardware resources, which results in the VMs taking up more resources. As such, OS-level virtualization, or Containers, have been developed. By sharing OS resources, containers are lightweight and can be spun up quickly while taking up fewer resources. Sharing OS resources such as libraries significantly reduces the need to reproduce the operating system code, and means that a server can run multiple workloads with a single operating system installation. Containers are thus exceptionally light.

Containers are implemented using Linux namespaces and cgroups. **Namespaces** let you virtualize system resources, like the file system or networking, for each container. **Cgroups** provide a way to limit the amount of resources like CPU and memory that each container can use.

Docker, introduced in 2013, is a popular runtime to manage containers as it addresses end-to-end management. However, Docker was initially a monolith with features not inherently dependent on each other being bundled together. As a result, alternative runtimes such as CRI-O and contianerd exist which implement container management at varying levels. [1]

The Open Container Initiative (OCI, https://www.opencontainers.org) has been established to create a open standard for container use regardless of the runtime being used to manage the container. However, the OCI only specifies downloading image then unpacking that image into an OCI Runtime filesystem bundle.

It does not standardize lifecycle management of the containers, thus each container implements lifecycle functionality in a different manner. It also does not ensure that consistent standards for the security of containers are present.

In this project, we will study the differences in popular runtimes Docker, containerd, and crio. Our focus will be on developing a service focused on ensuring that Center for Internet Security (CIS) Benchmarks for Docker are satisfied across other runtimes to ensure consistent application of security principles irrespective of container runtime differeces. Long term, the goal is to implement a lifecycle mangement solution that enables a common management framework for controlling containers across enviornments. By implementing a service to validate standard secuirty checks across runtimes, we intend to provide a Proof of Concept that such a common lifecycle management is possible.

A. Vision and Goals. Currently if someone wishes to launch an image in a container or perform any other lifecycle management functions on it, they must be sure that the scripts are configured correctly for the target container. For instance, launching images in Docker differs from doing so in CRI-O or containerd. This locks individuals and businesses into whichever container runtime they started with unless they invest the time required to edit the configuration and their scripts which holds the commands for target container.

Our short term goal for this project is to enable the set of Container Runtime tests run in the CIS Docker 1.13.0 Benchmark across any container runtime. These tests are specified in Chapter 5 of the document; example checks include restricting Linux Kernel capabilities within containers, limiting memory usage, and avoiding directly exposing the host devices to the containers. Publishing a minumum viable framework for this purpose will enable users to run their security checks using a single script across the most popular containers.

The ultimate goal is to develop an interoperable container runtime tool that allows user to perform common container lifecycle management functions among different runtimes using a single framework (e.g. start/stop execution, ps).

Users/Personas of the Project. The intended user is a software developer who is developing, testing, and managing applications across containers running on different runtimes.

Example Use Case: A software developer would like to launch an image in CRI-O instead of Docker, because he realizes that CRI-O is more adaptable with Kubernetes, and using this capability will provide this application a lot more scalibility. Presently, he needs to deal with changing all the continous-integration scripts in order to be able to test and deploy his application on this new container runtime. With our interoperable framework in place, the developer is at least able to run security checks on the new container runtime without changing their scripts beyond specifying the new target container. In this way, the user's workflow is simplified and can apply a standard across runtimes with minimal effort.

Scope. The runtimes in scope for capatibility for this project will be Docker, and CRI-O. containerd is considered a runtime in scope as a stretch goal.
This project aims to esnure that the framework implements commands that satisfy the CIS Docker 1.13.0 Benchmark related to Container Runtimes across our in-scope runtimes. In doing so, users will be enabled to run their security checks with a single script rather than requiring separate suites for each runtime. The MVP will be considered to be implementing select benchmarks in consultation with our mentor (highlighted in bold in the below list). Implementation of the full suite is a stretch goal.
These benchmarks are specified in pp 126-180 of the Benchmark documentation

Design

Interoperable framework will be a service which exists between developer and container runtimes. Our project aims to implement security checks across runtimes. The ultimate end goal will be to provide a means of lifecycle management of container runtimes in interoperable way. That is, a single script would be able to execute lifecyle commands independent of underlying container runtime.

Implications and Discussion. The underlying structure of the configuration files per runtime will be evaluated, as this is what we use in making each of these checks. The interoperable framework will consist of scripts to be developed in Python. The scripts serve as wrappers for functionality so that the end-user can run commands without caring which container runtime is the target container is running on. Per the OCI standard, Docker, CRI-O, and containerd all use runc to implement low-level functionality. Thus, our intention is to leverage runc to create our interoperable framework. As seen in the above diagram, Docker implements its own image management and APIs for high level functionality on top of the container. Thus, understanding the interaction between the high level functionality and operations in runc is necessary in order to be able to implement the functionality in other frameworks.

CIS Benchmarks. CIS Benchmarks are best practices for the secure configuration of a target system. Available for more than 140 technologies, CIS Benchmarks are developed through a unique consensus-based process comprised of cyber-security professionals and subject matter experts around the world. CIS Benchmarks are the only consensus-based, best-practice security configuration guides both developed and accepted by government, business, industry, and academia.

CIS Chapter 4. cis chapter 4 [Jonathan]

CIS Chapter 5. The ways in which a container is started governs a lot of security implications. It is possible to provide potentially dangerous run-time parameters that might compromise the host and other containers on the host. Verifying container run-time is thus very important. In this project we implement various recommendations to assess the container run-time security that is provided through CIS Chapter 5.
We implement following benchmarks:

5.1 Do not disable AppArmor Profile (Scored). AppArmor protects the Linux OS and applications from various threats by enforcing security policy which is also known as AppArmor profile. You can create your own AppArmor profile for containers or use the Docker's default AppArmor profile. This would enforce security policies on the containers as defined in the profile.

5.2 Verify SELinux security options, if applicable (Scored). SELinux provides a Mandatory Access Control (MAC) system that greatly augments the default Discretionary Access Control (DAC) model. You can thus add an extra layer of safety by enabling SELinux on your Linux host, if applicable.

5.3 Restrict Linux Kernel Capabilities within containers (Scored). By default, Containers start with a restricted set of Linux Kernel Capabilities. It means that any process may be granted the required capabilities instead of root access. Using Linux Kernel Capabilities, the processes do not have to run as root for almost all the specific areas where root privileges are usually needed.

- NET_ADMIN
- SYS_ADMIN
- SYS_MODULE

5.6 Do not run ssh within containers (Scored). Running SSH within the container increases the complexity of security management by making it difficult to manage access policies and security compliance for SSH server. Difficult to manage keys and passwords across various containers. Difficult to manage security upgrades for SSH server It is possible to have shell access to a container without using SSH, the needlessly increasing the complexity of security management should be avoided.

5.9 Do not share the host's network namespace (Scored). This is potentially dangerous. It allows the container process to open low-numbered ports like any other root process. It also allows the container to access network services like Dbus on the Docker host. Thus, a container process can potentially do unexpected things such as shutting down the Docker host. You should not use this option.

5.10 Limit memory usage for container (Scored). By default, container can use all of the memory on the host. You can use memory limit mechanism to prevent a denial of service arising from one container consuming all of the host's resources such that other containers on the same host cannot perform their intended functions. Having no limit on memory can lead to issues where one container can easily make the whole system unstable and as a result unusable.

5.11 Set container CPU priority appropriately (Scored). By default, CPU time is divided between containers equally. If it is desired, to control the CPU time amongst the container instances, you can use CPU sharing feature. CPU sharing allows to prioritize one container over the other and forbids the lower priority container to claim CPU resources more often. This ensures that the high priority containers are served better.

5.24 Confirm cgroup usage (Scored). System administrators typically define cgroups under which containers are supposed to run. At run-time, it is possible to attach to a different cgroup other than the one that was expected to be used. This usage should be monitored and confirmed. By attaching to a different cgroup than the one that is expected, excess permissions and resources might be granted to the container and thus, can prove to be unsafe.

5.28 Use PIDs cgroup limit (Scored). Attackers could launch a fork bomb with a single command inside the container. This fork bomb can crash the entire system and requires a restart of the host to make the system functional again. PIDs cgroup –pids-limit will prevent this kind of attacks by restricting the number of forks that can happen inside a container at a given time.

Interoperable Application

Interoperable application is a Python executable, which accepts two parameters from user. That are, target container runtime and container-id. For instance, to issue CIS Chapter 5 benchmarks over Docker container with id = 0606, user is expected to run command as: `./interoperable_app -docker 0606`.

The minimum acceptance criteria for this course is to enable at least 5 benchmark checks on two different container runtimes. We target Docker and CRI-O as our primary focus. Our stretch goal criteria include ensuring the commands also work with containerd and be able to run the full suite of CIS Docker Benchmark checks from Chapters 5 (Container Runtime checks) and Chapter 4 (Container Images and Build File) on those runtimes.

In this section we demonstrate that we were able to perform more than 10 benchmarks over all container run-times (Docker, Containerd and Cri-o).

Docker. Our primary platform is Docker container runtime. Here, we evaluate the how we implement the CIS checks on Docker. First, we find target container's pid from */run/containerd/io.containerd.runtime.v1.linux/moby/<container-id>/init.pid*. In order to issue **5.1 Do not disable AppArmor Profile** and **5.2 Verify SELinux security options** benchmarks over Docker, we use procfs and pid. Apparmor and SELinux are security attributes for a given process. So these values are stored under */proc/<pid>/attr/apparmor/current* and */proc/<pid>/attr/selinux/current* respectively. By exposing these values, we are able to issue **5.1** and **5.2**.

Per container instance, configuration file is created under */run/containerd/io.containerd.runtime.v1.linux/moby/<container-id>/config.json* file. From this file, we get **cgroup** path of a given container. Exposing this value also corresponds to **5.24** benchmark.

Further, by using the cgroup path, we are able to access *sysfs* of a given container. *sysfs* is a pseudo file system provided by the Linux kernel that exports information about various kernel subsystems. From this file, we are able to perform **5.10, 5.11, 5.28**. **5.10 Memory Limit** of a container is found from: */sys/fs/cgroup/memory/<container-id>/memory.limit_in_bytes*. **5.11 CPU Share** of a container is found from: */sys/fs/cgroup/cpu/<container-id>/cpu.shares*. **5.28 PID Limits** of a container is found from: */sys/fs/cgroup/pids/<container-id>/pids.max*.

Containerd.

Cri-o.

Frakti.

Bibliography

Supplementary Note 1: Something about something

appendix