Interoperable Application for Issuing CIS Benchmarks

Mert Toslali[1], Jonathan Chamberlain[1], Felipe Dale Figeman[1], Zhengyang Tang[1], and Shripad Nadgowda[2]

[1]BU EC528 - Cloud Computing
[2]IBM T.J. Watson

The Open Container Initiative (OCI) was established to create a open standard for container use regardless of the runtime used to manage the container. However, OCI is limited to specifying downloading of the image and unpacking that image into an OCI Runtime filesystem bundle. It does not contain any specifications for lifecycle management of containers. As a result, there are multiple container runtimes in use, each implementing lifecycle functionality in a different manner. Related, the OCI does not does not ensure consistent standards for the security of containers or images are present across different runtimes. In this project, we study the differences in popular runtimes Docker, containerd, and cri-o. Our focus is on developing an interoperable application focused on running Center for Internet Security (CIS) Benchmarks across runtimes. While written for Docker, most benchmarks are general, e.g. ensuring that memory usage limits exist. As a result, we can ensure consistent application of security principles irrespective of container runtime differences.

Introduction

In recent decades, virtualization of resources has emerged as a means to run multiple OSes on the same hardware. Indeed, Virtual Machines (VMs) are *taken to be an efficent, isolated duplicate of the real machine* (1). This serves a useful function by permitting multiple applications to coexist on the same server, enabling efficiencies in computing such as server consolidation. Traditional VMs virtualize hardware resources; while this achieves the goal of isolating the VM, it can result in inefficiencies when multiple VMs are running, on the same hardware. This especially holds if the VMs are running multiple instances of the same application. In response to this, containerization has been developed in order to achieve operating system-level virtualization. By sharing OS resources, containers are lightweight and can be spun up quickly. As this resource sharing significantly reduces the need to reproduce the OS code, containers take up far fewer resources. This also leads to the advantage of servers running multiple workloads with a single operating system installation.

In order to implement virtualization at the OS level, Containers make use of Linux namespaces and cgroups. **Namespaces** permit the vritualization of resources in each container, such as networking or the underlying file system. **Cgroups** are a feature of the Linux kernel which provide a way to limit and isolate resource usage per container. (2) However, this proved to be largely difficult until the development of Docker. Released in 2013, Docker addresses end-to-end management of containers including defining a container image format and a means to build images, the ability to manage and share images, and manage and run containers. However, none of these features were necessarily dependant on each other. Rather than being packaged into a single monolithic application, each could be implemented as a series of more focused tools which can be used together in whatever combination the user needs (2).

To that end, industry leaders (including Docker) established the Open Container Initiative (OCI) in June of 2015. OCI consists of a Runtime Specification and an Image Specification. (3). These specifications cover the downloading and unpacking of images into an OCI Runtime Filesystem bundle. No uniform standard exists to manage lifecycle functionality; although minimum requirements exist on behavior of certain commands, *how* they are implemented is largely left up to the runtime developer (4). Because OCI does not fully standardize lifecycle management of containers, each container runtime implements this functionality in a different manner. In particular, this results in a lack of consistent standards for the security of containers, as OCI does not include these in its own specifications, and many security concerns are related to lifecycle management.

In this project, we studied the differences in three of the most popular runtimes: Docker, containerd, and cri-o. From this, we developed an interoperable application focused on performing certain industry standard security checks across runtimes. In particular, our focus is on ensuring that the benchmarks specified by the Center for Internet Security (CIS) for Docker are satisfied in cri-o and containerd (5).

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 crio 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.

In particular, in order to evaluate security properties, such as whether memory usage is limited, or AppArmor is enabled, it is necessary to either run the necessary commands by hand, or update validation scripts to reflect the new runtime in use. Our goals for this project were to develop an application capable of running standard security checks (5, Ch.5) on the container. Publishing a minimum viable framework for this

purpose will enable users to run their security checks using a single application across the most popular containers.
In the longer term, this would enable automation of these checks via implementing the application within a service running on each cluster hosting containers. Doing so however was beyond the scope of our immediate project due to time constraints; we discuss in our concluding section the future directions of the project in terms of the longer term vision as well as next steps.

Users/Personas of the Project. The intended user is the Chief Information Security Officer or their designee of a medium- or large-scale cloud organization, who is tasked with ensuring that containers running their clusters meet requirements for secure containers.

Scope. The runtimes within our scope are Docker, cri-o, and containerd. Our MVP was to implement five of the benchmarks related to container runtimes within Docker and cri-o (5, Ch. 5). Ultimately, we were able to configure the application so that 23 benchmarks are evaluated for Docker, 20 for cri-o, and 16 for containerd. Some benchmarks were determined to be Docker specific and not applicable to the other runtimes, others require additional evaluation to determine how to implement. A more detailed discussion of the implementation for individual benchmarks is contained in the section on the Interoperable Application can be found in later sections and appendix.
In addition to implementation of more than five container runtime benchmark checks, our stretch goal also included implementing benchmark checks for container images/image files (5, Ch. 4) across Docker, cri-o, and containerd. Ultimately, we were able to implement five of the eleven image file benchmarks within Docker, and three within cri-o. Discussion of the challenges for the image file benchmarks are contained in the section on the image file benchmark definitions.

Design

As the intent is to examine containers and their associated images to validate they meet the security benchmarks, the Interopable Application and the Interoperable Image Application need exposure to the container and image configuration files, as well as sysfs and procfs. Thus, the applications are deployed on the cluster hosting the containers to be inspected, and exist between the user and the container.
The applications are implemented in Python version 3. At present, the applications are implemented as separate standalone scripts which are called with a specified runtime and container ID on the command line. This was done largely to provide proof of concept that performing the checks are possible, and to facilitate testing. Combining the applications into a single service is part of future work discussed in a later section.
In addition, there is a separate helper function, the Image Approval Manager that works with the Interoperable Image Application. This manager simply takes image ids and adds them to a list of approved images or blocked/unapproved images. This helps to facilitate one of the benchmark checks outlined in the section on container image benchmarks.

Implications and Discussion. Where possible, the Interoperable Application leverages the attributes that are exposed in *sysfs* and *procfs* per process in Linux. sysfs and proc are pseudo-filesystems which provide an interface to kernel data structures. The files under sysfs provide information about devices, kernel modules, filesystems, and other kernel components. For instance, sysfs provides a means to determine if memory usage has been limited, as well as if the CPU priority is set appropriately. proc is primarily used to determine if the AppArmor Profile is enabled for the associated process id.
The use of these interfaces limits the possibility that future updates to the containers causes a change in the configuration file structure, which would then require an overhaul of the applications to ensure that the security checks can still be carried out for the impacted runtime. However, some attributes are only found in the configuration files. This is especially true for any attribute related to the image files. Thus, a major part of this project was dissecting how each runtime operates, and which configuration files are utilized.

CIS Benchmarks. CIS is a community driven organization of volunteer professionals, aimed at refining best practices and tools for information security (6). CIS publishes various Benchmark documents for best practices for secure configuration of systems, including Docker (7). There are no specific Benchmark documents for cri-o and containerd, however the majority of the Docker benchmarks are concerned with attributes which are not Docker-specific.
Thus, our design is focused on determining how the relevant setting or settings for each benchmark are enabled in each runtime, and validate that the benchmark is satisfied. As noted under the challenges section however, in many cases there is not an unambiguous pass condition, instead simply having a fail condition to check against. Where possible we highlighted instances that did not fail but do not necessarily follow a best practice highlighted elsewhere, or require manual follow up before being able to identify a pass/fail, as is the case with most of the image benchmarks.

Container Runtime Benchmarks. Chapter 5 of the CIS Docker Benchmarks documentation specifies security checks for the container runtimes themselves. These checks are the principal focus of this project, as the ways in which containers are started have many security implications. It is possible to provide potentially dangerous run-time parameters which can compromise the host as well as other containers running on the host. This is especially necessary as in many cases, the default behavior is to not set certain attributes. As a result, verifying the properties of a particular container runtime is critical.
There are a total of 31 Benchmarks specified in the documentation. We mention below the benchmarks which were our initial focus in implementing the application. The full

list of benchmarks implemented per runtime is outlined in Figures 1, 2, and 5 in the Appendix. The full explanation of the scopes and intentions of the benchmarks is contained in the benchmark documentation (5, Ch. 5).

5.1 Do not disable AppArmor Profile. AppArmor enforces security policy on the Linux OS and applications. The particular policy is speficied in the AppArmor profile; users can create their own profile or use Docker's default profile in order to enforce policy on the containers (5, pp 126-7).

5.2 Verify SELinux security options, if applicable. The default access control model in Docker containers is Discretionary Access Control (DAC) model. Within DAC, subjects are capable of passing permissions to other subjects. SELinux provides a Mandatory Access Control (MAC) system augments the default DAC model. MAC enables further restrictions on actions which can be made by subjects and prevent permissions from being passed on from one subject to another. This ultimately adds an extra layer of security (5, 128-9).

5.3 Restrict Linux Kernel Capabilities within containers. Docker containers begin with a restricted set of Linux Kernel Capabilities by default. With Linux Kernel Capabilities, any process may be granted the required capabilities rather than restricting the capabilities to root. As such, it is recommended to only have the capabilities necessary to run the container process. As an example, the following capabilities are generally not required for container processes to function as expected: (5, pp 130-1)

- NET_ADMIN
- SYS_ADMIN
- SYS_MODULE

The full list of possible capabilities are contained in the relevant man page (8).

5.6 Do not run ssh within containers. Running SSH within a container increases security management complexity by making it difficult to manage access policies and security requirements, keys and passwords across containers, and managing security upgrades for the SSH server. Rather than SSH, it is recommended to use other methods to interact with the container instance, such as Docker's exec and attach commands, or the nsenter comamnd (5, pp 135-6).

5.10 Limit memory usage for container. By default, no limit is set and the container is able to use all memory on the host. Use of memory limits prevents a situation where other containers on the same host cannot run due to one container using all available memory (5, pp 142-3).

5.11 Set container CPU priority appropriately. By default, containers on Docker use equal sharing of CPU resources. CPU sharing enables prioritization of one container over others. Thus, containers running higher priority process will receive a greater share of the CPU resources ensuring that they are served better (5, 144-5).

5.24 Confirm cgroup usage. By default, containers run under the docker cgroup, although system administrators may define a different cgroup under which the container is supposed to run. However, at runtime it is possible to attach a cgroup other than the one intended to be used, which can result in permissions and resources being granted to the container not allowed by the intended cgroup (5, pp 168-9).

5.28 Use PIDs cgroup limit. The PIDs cgroup limit sets a limit on the number of processes which can run within a container at once. In particular, this limits the number of forks which can occur inside the container. Without such a limit, attackers could launch a fork bomb with a single command inside the container, which is capable of crashing the entire system. In such a case, a restart of the host is necessary to make the system functional (5, pp 175-6).

Container Images and Build File Benchmarks. Chapter 4 of the CIS Docker Benchmarks document is focused on the container images/build files. While the primary focus of our work was on the containers themselves, the images are what contain the executable code, libraries, etc. that permit an application to run within the container. Thus, the use of proper build files are important for secure infrastructure. The following specific checks were implemented in at least one runtime:

4.1 Create a user for the container. By default, the build file does not specify a user resulting in root becomming the default user. It is typically good practice to not run the container as root in order to limit user privileges, and by specifying a user within the build file itself, user namespace remapping is not requried to prevent containers from being run as root (5, pp 105-6).

4.2 Use trusted base images for containers. Ideally, containers should come from verified sources or be based on a build file written from scratch by the organization using the container. If from a remote repository, there should be proof that the image was obtained from a trusted secure channel (5, pp 107-8).

4.5 Enable Content trust for Docker. Content trust enables the use of digital signatures from remote registries. This permits verification on the client-side of the integrity and publisher of the image (5, pp 113-4).

4.6 Add HEALTHCHECK instruction to the container image. Healthcheck instructions ensure availability by ensuring that the Docker engine checks running container instances against the provided instruction. If a container has stopped working, this allows the engine to instantiate a new container and exit the non-working container (5, p 115).

4.9 Use COPY instead of ADD in Dockerfile. Using COPY instructions in Dockerfiles copies the relevant files from local host. If ADD is used instead, files could be retrieved from remote registries, and in turn cause malicious files to be unpack from unverified URLs (5, pp 120-1)

In the section on the Interoperable Image Application, we discuss the implementation of checks for the above Benchmarks. In addition, Chapter 4 of the document specifies the following benchmarks which were not implemented; the reasons for not doing so are elaborated on in the Challenges and Take Aways section:

- 4.3 Do not install unnecessary packages in the container
- 4.4 Scan and rebuild the images to include security patches
- 4.7 Do not use update instructions alone in the Dockerfile
- 4.8 Remove setuid and setgid permissions in the images
- 4.10 Do not store secrets in Dockerfiles
- 4.11 Install verified packages only

Interoperable Application

The Interoperable Application is a Python executable, which accepts two parameters from user: the target container's runtime and the container-id. For instance, to evaluate the CIS Chapter 5 benchmarks over a Docker container with id = 0606, a user is expected to run the following command: `./interoperable_app -docker 0606`.
In this section we demonstrate that we are able to perform more than 10 benchmarks over all in-scope container runtimes. For brevity, we highlight the benchmark check implementations that were more challenging. For the full list of benchmarks implemented per runtime, please refer to Figures 1, 2, and 5 in the Appendix.

Docker. Our primary focus in initial development was ensuring the benchmarks could be validated in the the Docker container run-time, as the CIS Benchmarks are defined for Docker. In this subsection, we detail the technical details of how we validate the benchmarks highlighted in the Container Runtime Benchmarks section. First, we find the pid associated with the target container by inspecting */run/containerd/io.containerd.runtime.v1.linux/moby/<container-id>/init.pid*.
With the pid, we are able to evaluate Benchmarks **5.1 Do not disable AppArmor Profile** and **5.2 Verify SELinux security options**. Apparmor and SELinux are security attributes for a given process, and can be verified within procfs. In particular, these values are stored under */proc/<pid>/attr/apparmor/current* and */proc/<pid>/attr/selinux/current* respectively. Exposing these values enables verification of **5.1** and **5.2**.
In order to validate the cgroup assigned to the container, we need to inspect the container's configuration json file. For a particular instance, the configuration file is located at the path */run/containerd/io.containerd.runtime.v1.linux/moby/<container-id>/config.json*. The cgroup path is contained as an attribute within this file. Exposing this value enables verification of the **5.24** benchmark directly, as well as enables further additional checks.
With the cgroup path known, we are able to access *sysfs* of a given container in order to validate Benchmarks **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. Having defined multiple benchmark checks for Docker, we now expand our checks to the other in scope runtimes. In this subsection we detail how the applications performs the benchmark evaluations in containerd. As is noted below, the process is similar, with the primary difference being how containerd stores the relevant configuration files. As before, the first step is to find the target container's pid. In containerd this is accomplished by inspecting */run/containerd/io.containerd.runtime.v2.task/default/<container-id>/init.pid*.
Once the pid is obtained, evaluating benchmarks **5.1 Do not disable AppArmor Profile** and **5.2 Verify SELinux security options** is identical to in Docker, as procfs is related to the process on the host and does not differ based on the container runtime in use.
In order to obtain the cgroup path, we inspect the configuration json file, which here is located on the path */run/containerd/io.containerd.runtime.v2.task/default/*/<container-id>/config.json*. As in Docker, there is an attribute containing the **cgroup** path of a given container. In addition to being able to validate benchmark **5.24** directly, we are able to proceed to verification of benchmarks **5.10, 5.11, 5.28** as in Docker. Again, this is due to sysfs and the cgroup being related to processes on the host rather than anything specific to the container.

Cri-o. In this subsection we describe how we evaluate selected benchmark checks within the cri-o runtime. The process for implementing a particular benchmark check is similar to those within Docker and Containerd.
As before, we again find the container's pid. However, in cri-o this is contained within the state.json file, as opposed to inspecting an init.pid file: */var/lib/containers/storage/overlay-containers/ <container-id>/userdata/state.json*
However, once the pid is obtained, the process to verify benchmarks **5.1 Do not disable AppArmor Profile** and **5.2 Verify SELinux security options** is again identical to that in Docker as with the pid we can use procfs to find the AppArmor and SELinux attributes.
To determine the cgroup path of the container, we again find the corresponding attribute in the configuration json file: *//var/lib/containers/storage/overlay-containers/ <container-id>/userdata/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 validate

benchmark **5.24** directly, as well as leverage *sysfs* to validate benchmarks **5.10, 5.11, 5.28** as in Docker and containerd.
Thus, we can see that the validation of our highlighted benchmarks is relatively similar across runtimes. This is largely because we are able to leverage the interfaces processed by sysfs and procfs to validate attributes associated to the container process. The principal challenge in the implementation was understanding where the relevant attributes necessary were stored, as well as understanding how sysfs and procfs work. As noted below, there exist other configuration files associated with containers which are used in validating the remaining benchmarks.

Other benchmarks. In this section we briefly highlight how to validate the remaining benchmark checks impelmented for each runtime. The Benchmarks are refered to by their number as listed in (5, Ch. 5). The full list is contained in the Appendix, and as noted previously the associated scope for each benchmark is contained within the cited Benchmark document.
For the remaining benchmarks, the Interoperable Application primarily leverages fields within the corresponding configuration files for the target container container. For example, to validate **5.4, 5.7, 5.8, 5.9, 5.12, 5.13, 5.14, 5.15, 5.16, 5.17, 5.18, 5.20 and 5.25** in Docker, we use Hostconfig.json file for a given container and inspect the relevant fields. For a particular container, this file is stored on the path: */var/lib/docker/containers/ <container-id>/hostconfig.json*. To implement checks for benchmarks **5.3, 5.5, 5.21, 5.24** in Docker, our application uses the config.json file located at */run/containerd/io.containerd.runtime.v1.linux/moby/<container-id>/config.json*.

For cri-o, the checks on benchmarks **5.3, 5.4, 5.5, 5.7, 5.8, 5.9, 5.12, 5.13, 5.15, 5.16, 5.17, 5.20, 5.21, 5.24** are implemented within the Interoperable Application by inspecting the container's config.json file. The relevant file is located on the path */var/lib/containers/storage/overlay-containers/<container-id>/userdata/config.json*

For containerd, the container's config.json file is again used to validate benchmarks. Specifically, the Interoperable Application is able to inspect relevant fields to validate **5.3, 5.4, 5.5, 5.17, 5.24, 5.25** using the config.json file per container. This file is found on the path */run/containerd/io.containerd.runtime.v2.task/default//<container-id>/config.json*. The Interoperable Application inspects the container's state.json file in order to validate benchmarks **5.12, 5.15, 5.16, 5.20, 5.21**. This file is located at */run/containerd/runc/default/<container-id>/state.json*

Interoperable Image Application

Currently, the validation on the image benchmarks is performed by a separate application. In addition to ensuring modularization of code, this was also done to prevent interference with the main Interoperable Application. Similarly to that application, the Interoperable Image Application is a python executable which takes the target container's runtime and the container id as input. Thus, if a user wishes to run the CIS Chapter 4 benchmark checks on the image running on container 0606, the user is expected to run the following: `./interoperable_image_app -docker 0606`.
In this section, we demonstrate that we are able to evaluate at least some of the benchmarks discussed in the section on Container Images and Build File Benchmarks within Docker and cri-o, and how these are evaluated.

Docker. Within Docker, each image has a configuration file corresponding to it located within the Docker image directory: */var/lib/docker/image/overlay2/imagedb/content/sha256/<image id>*. This file contains the attributes relevant to most of the benchmarks currently implemented.
To find the id of the image that a container is running, the application inspects the relevant configuration json for the image id: *"/var/lib/docker/containers/ <container id>/config.v2.json* The image id is a distinct attribute within this json file.
Once opened, the application can validate benchmark **4.1, Create a user for the container** by determining if the User field is populated. If not, the user is root and the check fails.
To validate benchmark **4.9, Use COPY instead of ADD in Dockerfile**, the Interoperable Image Application inspects the history attribute, which contains an entry for each instruction used to create the image. If any entry as ADD present, the benchmark is flagged as a failure; if a COPY instruction is present with no ADD instructions, the benchmark is marked as passing. If no ADD or COPY instructions are present, the bemchmark is flagged as not applicable rather than passing, as it is not clear this should be considered a pass based on the wording of the recommendation. This is a design decision which may be changed based on future feedback on best practices.
To validate benchmark **4.6, Add HEALTHCHECK instruction to the container image**, the application checks the image configuration file to determine if the healthcheck attribute is present. If not, the check is flagged as a failure. If an instruction is present, it is flagged as a pass unless the curl or iwr instruction is detected. In that case, the benchmark is flagged as requiring further follow up. This is because there may be additional dependencies required for these instructions to function properly in the associated container.
The remaining benchmarks do not require the image configuration file. To validate benchmark **4.5, Enable Content trust for Docker**, the application checks if the environment variable ENABLE_DOCKER_TRUST is set to 1. If so, the benchmark passes. If not set or set to any other value, the benchmark fails.
To validate benchmark **4.2, Use trusted base images for containers**, the application checks the image id against a pair of lists: an approved list and a block list. If the id is found in the approved list, the benchmark passes. If the id is found in the block list, the benchmark is rejected. If the image id is found in neither list, it is flagged for further follow up, as in this case it is considered to be an unknown image. This requires follow up by a System Administrator or a security

officer to ensure the provencence of the image. Based on this follow up the image can be added to the approved list or block list as appropriate using the Image Approval Manager.

cri-o. Within cri-o, we adopt a reduced set of these checks, in part due to differences in the configuration files. Of the attributes discussed in the previous section, only the history is contained in the manifest file related to the target image. However, the user being used by the container is contained in the *container's* configuration json file.
Thus, to validate benchmark **4.1** in cri-o, the Interoperable Image App simply inspects the user attribute in the container config.json located at */var/lib/containers/storage/overlay-containers/ <container id>/userdata/config.json*. If the associated uid is 0, the check fails as 0 is the root user id. Otherwise, the check passes as a non root user is present.
The same configuration file contains an id within the image attribute. However, this is not the id associated with the image manifest but rather a digest hash. In order to solve this additional layer of indirection, the Interoperable Image App first opens the images.json file within the overlay-images directory: */var/lib/containers/storage/overlay-images/images.json* This file contains information about all images, including the association between the image id, and the image digest stored in the image configuration file.
Once a matching id is found, the application is able to open the corresponding manifest file: */var/lib/containers/storage/overlay-images/<image id>/manifest*, and inspect the history attribute in order to validate benchmark **4.9**. In addition, with the image id known benchmark **4.2** is verifiable in the same manner as in Docker.
Currently, validation of the benchmarks **4.5, Enable Content trust for Docker** and **4.6, Add HEALTHCHECK instruction to the container image** are not currently implemented for cri-o by the Interoperable Image Application. In the case of the former, this is because signature verification for image content trust in cri-o is handled by a policy.json file and does not involve setting an enviornment variable as in Docker. The policy.json file allows for a refinement of policies based on the source in question. While this allows acceptance/rejection based on specific criteria, a global default is required, which is frequently set to insecureAcceptAnything which does not reject anything. (9). To properly implement this particular check requires additional feedback as to best practices.
For the latter, there exists a code module within libpod to perform healthchecks on containers (10). It is not clear based on this how to validate the Healthcheck instructions, as there is no guidance as to where the module is looking for the check as defined by the container. Thus, further followup is required to determine how to leverage this check within cri-o.

containerd. Currently, the Interoperable Image Application is not implemented within containerd. While we determined the location of the configuration file contianing relevant attributes, */var/lib/containerd/io.containerd.content.v1.content/blobs/sha256*, there is no attribute in the known configuration files for the container that links the container to the image id in such a way that we can know which image to run the checks on. This was filed as work for future follow up in order to avoid stalling progress on the other runtimes.

Challenges and Take Aways

The primary challenge up front was figuring out how the runtimes operate. In particular, frakti was initially part of our in-scope runtimes. While relatively new, it is also in popular use per our mentor at IBM. However, documentation on how to install and run frakti proved to be outdated or otherwise incomplete. It is also not altogether clear whether frakti remains active as sources discussion other containers mention it primarily in passing. As a result, it was decided to remove frakti from scope due to insufficient information and it possibly being deprecated. This did result in wasted person-hours during the initial phase of the project in determining how the targeted runtimes operate.
With respect to implementing the benchmark checks, while verifying that a particular benchmark is not satisfied is fairly straightforward, there exists a parallel issue of the benchmarks not necessarily defining a best practice for what the setting should be set to. There is some logic to this - while what not to do is relatively straightforward to assess, such as flagging a container which does *not* have a memory limit set, what limit should be placed on the container depends on organizational policy and what application the container is running. Thus, while finding the setting within the file is straightforward, whether there should be a means to set a threshold corresponding to institutional policy requires further discussion regarding how best practices should work.
This is especially the case for the benchmarks relating to the images, as many of the checks require manual follow up rather than being able to flag that a setting is missed. For instance, benchmark **4.3, Do not install unnecessary packages in the container** depends in part on what application the image is intended to be running. Other of the image checks require more complex logic in order to validate whether they pass or not, such as verifying whether only verified packages are installed (**4.11**), or using update instructions alone in the Dockerfile (**4.7**).
The biggest take aways as a result are that there is not necessarily a standard value that the various attributes should be set to. In many cases, this can result in a default value which is dangerous for use on live containers. In relying on organizations to set the attributes themselves, this opens the containers and associated images to various attacks including fork bombs which can bring down the host, and infection of cgroups which can lead to exfiltration of container data by taking over kubernetes nodes (11). However, there do exist built in isolation measures which can aid in preventing a single compromised container from creating issues for the other containers on the same host. Thus, the ability to verify whether those isolation measures are active (e.g. AppArmor, SELinux) is critical to container security.
In addition, another take away from the group as a whole is

that once we were able to begin development, we became so caught up in checking off each benchmark as complete, that we failed to stop and reconsider the intended use case. In particular, whether the user we originally envisioned, a developer working with multiple runtimes, was the relevant one as the project evolved to primarily encompass running the security checks within each in-scope runtime. Upon feedback from the instructors, we took this back to our mentor who clarified that this is a tool to be leveraged by the CISO. However, at this point there was little time remaining to perform any additional changes to existing code to facilitate the next steps which emerged from that discussion, which leads us to our concluding section.

Future Work

Currently the application requires manual invocation of a particular container on the command line. As our intended user is an organization CISO or their designated security personnel, this is not practical given that within a large organization, containers are spun up and stood down at large volumes, and thus it is not practical to inspect containers one by one. Thus, the current state of our application can be considered as being in more of a proof of concept stage, in that we can run checks for the majority of the specified benchmarks on a given container.

Given the number of containers to inspect, how to ensure scalability to efficiently report on each container is the next step to be considered. We imagine standing up a daemon running on a Kubernetes cluster. As containers are spun up, the daemon would be triggered and automatically run all of the benchmark checks. Reports would be made available to the designated security staff, with alerts if failures are detected or follow ups required (depending on the benchmark and result). One advantage is that all containers on the same cluster would utilize the same runtime, avoiding the need to specify the target runtime on each invocation. Ideally, it would be possible for this daemon service to automatically detect which runtimes are being used, however at present it is not clear how to do so, and thus for the moment it would be necessary to specify this on a configuration level.

In addition, there are still some benchmarks checks which have yet to be implemented. Given the nature of some of the benchmarks, both currently implemented and not, ideally there would be discussions with stakeholders as how to define a best practice in terms of what settings should be specifically set to (as opposed to specifying that they must be set at all). This would require a means to maintain a configuration to define the best practice as defined by a particular organization, one that is more robust than the currently implemented tool to manage approved/blocked images.

Bibliography

1. Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974. ISSN 0001-0782. doi: 10.1145/361011.361073.
2. Ian Lewis. Container runtimes part 1: An introduction to container runtimes, Dec 2017.
3. Linux Foundation. Home - open containers initiative, 2016.
4. Open Container Inititiative. Github:opencontainers/runtime-spec/runtime.md, Oct 2017.
5. Center for Internet Security. *CIS Docker 1.13.0 Benchmark*. Center for Internet Security, 2017.
6. Center for Internet Security. Communities, 2019.
7. Center for Internet Security. Cis benchmarks, 2019.
8. Linux Programmer's Manual. capabilities - overview of linux capabilities, November 2019.
9. Github:image/containers-policy.json.5.md, Mar 2019.
10. Github:cri-o/healthcheck.go, Dec 2019.
11. Daniel Sagi. Dns spoofing on kubernetes clusters, Aug 2019.

Appendix

Container Runtime Benchmark Recommendations. Below is the full list of benchmarks as listed in Chapter 5 of (5), and as referenced by benchmark id in the Ineroparable Application section.

- 5.1 Do not disable AppArmor Profile
- 5.2 Verify SELinux security options, if applicable
- 5.3 Restrict Linux Kernel Capabilities within containers
- 5.4 Do not use privileged containers
- 5.5 Do not mount sensitive host system directories on containers
- 5.6 Do not run ssh within containers
- 5.7 Do not map privileged ports within containers
- 5.8 Open only needed ports on container
- 5.9 Do not share the host's network namespace
- 5.10 Limit memory usage for container
- 5.11 Set container CPU priority appropriately
- 5.12 Mount container's root filesystem as read only
- 5.13 Bind incoming container traffic to a specific host interface
- 5.14 Set the 'on-failure' container restart policy to 5
- 5.15 Do not share the host's process namespace
- 5.16 Do not share the host's IPC namespace
- 5.17 Do not directly expose host devices to containers
- 5.18 Override default ulimit at runtime only if needed
- 5.19 Do not set mount propagation mode to shared
- 5.20 Do not share the host's UTS namespace
- 5.21 Do not disable default seccomp profile
- 5.22 Do not docker exec commands with privileged option
- 5.23 Do not docker exec commands with user option
- 5.24 Confirm cgroup usage

- 5.25 Restrict container from acquiring additional privileges
- 5.26 Check container health at runtime
- 5.27 Ensure docker commands always get the latest version of the image
- 5.28 Use PIDs cgroup limit
- 5.29 Do not use Docker's default bridge docker0
- 5.30 Do not share the host's user namespaces
- 5.31 Do not mount the Docker socket inside any containers

Fig. 1. Benchmarks implemented in Docker.

DOCKER		
BENCHMARK	DEFINITION	IMPLEMENTATION
5.1	Do not disable AppArmor Profile (Scored)	/proc/{pid}/attr/apparmor/current
5.2	Verify SELinux security options, if applicable (Scored)	ps -eZ | grep {pid_container}
5.3	Restrict Linux Kernel Capabilities within containers (Scored)	/run/containerd/io.containerd.runtime.v1.linux/moby/{}/config.json -> ["process"]["capabilities"]["permitted"]
5.4	Do not use privileged containers (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["Privileged"]
5.5	Do not mount sensitive host system directories on containers (Scored)	/run/containerd/io.containerd.runtime.v1.linux/moby/{}/config.json -> ["mounts"]
5.7	Do not map privileged ports within containers (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["PortBindings"]
5.8	Open only needed ports on container (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["PortBindings"]
5.9	Do not share the host's network namespace (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["NetworkMode"]
5,10	Limit memory usage for container (Scored)	/sys/fs/{cgroup_from_config}/memory/{cont_id}/memory.limit_in_bytes
5,11	Set container CPU priority appropriately (Scored)	/sys/fs/{cgroup_from_config}/cpu/{cont_id}/mcpu.share
5,12	Mount container's root filesystem as read only (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["ReadOnlyRootfs"]
5,13	Bind incoming container traffic to a specific host interface (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["PortBindings"]
5.14	Set the 'on-failure' container restart policy to 5 (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["RestartPolicy"]["MaximumRetryCount"]
5.15	Do not share the host's process namespace (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["PidMode"]
5.16	Do not share the host's IPC namespace (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["IpcMode"]
5.17	Do not directly expose host devices to containers (Not Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["Devices"]
5.18	Override default ulimit at runtime only if needed (Not Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["Ulimits"]
5,20	Do not share the host's UTS namespace (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["UTSMode"]
5,21	Do not disable default seccomp profile (Scored)	/run/containerd/io.containerd.runtime.v1.linux/moby/{}/config.json -> ["linux"]["seccomp"]
5,24	Confirm cgroup usage (Scored)	/run/containerd/io.containerd.runtime.v1.linux/moby/{}/config.json -> ["linux"]["cgroup"]
5,25	Restrict container from acquiring additional privileges (Scored)	/var/lib/docker/containers/{}/hostconfig.json -> ["SecurityOpt"]
5,28	Use PIDs cgroup limit (Scored)	/sys/fs/{cgroup_from_config}/pids/{cont_id}/pids.max

Fig. 2. Benchmarks implemented in cri-o.

CRIO		
BENCHMARK	DEFINITION	IMPLEMENTATION
5.1	Do not disable AppArmor Profile (Scored)	/proc/{pid}/attr/apparmor/current
5.2	Verify SELinux security options, if applicable (Scored)	ps -eZ | grep {pid_container}
5.3	Restrict Linux Kernel Capabilities within containers (Scored)	/var/lib/containers/storage/overlay-containers/{cont_id}/userdata/config.json -> ["process"]["capabilities"]["permitted"]
5.4	Do not use privileged containers (Scored)	/var/lib/containers/storage/overlay-containers/{**pod_id**}/userdata/config.json -> ["annotations"]["io.kubernetes.cri-o.PrivilegedRuntime"]
5.5	Do not mount sensitive host system directories on containers (Scored)	/var/lib/containers/storage/overlay-containers/{cont_id}/userdata/config.json -> ["mounts"]
5.7	Do not map privileged ports within containers (Scored)	/var/lib/containers/storage/overlay-containers/{**pod_id**}/userdata/config.json -> ["annotations"]["io.kubernetes.cri-o.PortMappings"]
5.8	Open only needed ports on container (Scored)	/var/lib/containers/storage/overlay-containers/{**pod_id**}/userdata/config.json -> ["annotations"]["io.kubernetes.cri-o.PortMappings"]
5.9	Do not share the host's network namespace (Scored)	/var/lib/containers/storage/overlay-containers/{**pod_id**}/userdata/config.json -> ["annotations"]["io.kubernetes.cri-o.HostNetwork"]
5,10	Limit memory usage for container (Scored)	/sys/fs/{cgroup_from_config}/memory/{cont_id}/memory.limit_in_bytes
5,11	Set container CPU priority appropriately (Scored)	/sys/fs/{cgroup_from_config}/cpu/{cont_id}/cpu.share
5,12	Mount container's root filesystem as read only (Scored)	/var/lib/containers/storage/overlay-containers/{**pod_id**}/userdata/config.json -> ["root"]["readonly"]
5,13	Bind incoming container traffic to a specific host interface (Scored)	/var/lib/containers/storage/overlay-containers/{**pod_id**}/userdata/config.json -> ["annotations"]["io.kubernetes.cri-o.PortMappings"]
5.15	Do not share the host's process namespace (Scored)	/var/lib/containers/storage/overlay-containers/{cont_id}/userdata/config.json -> ["linux"]["resources"]["namespaces"]["type"] == "pid" -> get path
5.16	Do not share the host's IPC namespace (Scored)	/var/lib/containers/storage/overlay-containers/{cont_id}/userdata/config.json -> ["linux"]["resources"]["namespaces"]["type"] == "ipc" -> get path
5.17	Do not directly expose host devices to containers (Not Scored)	/var/lib/containers/storage/overlay-containers/{cont_id}/userdata/config.json -> ["linux"]["resources"]["devices"]
5,20	Do not share the host's UTS namespace (Scored)	/var/lib/containers/storage/overlay-containers/{cont_id}/userdata/config.json -> ["linux"]["resources"]["namespaces"]["type"] == "uts" -> get path
5,21	Do not disable default seccomp profile (Scored)	/var/lib/containers/storage/overlay-containers/{**cont_id**}/userdata/config.json -> ["annotations"]["io.kubernetes.cri-o.SeccompProfilePath"]
5,24	Confirm cgroup usage (Scored)	/var/lib/containers/storage/overlay-containers/{cont_id}/userdata/config.json -> ["linux"]["cgroupsPath"]
5,28	Use PIDs cgroup limit (Scored)	/sys/fs/{cgroup_from_config}/pids/{cont_id}/pids.max

Fig. 3. Benchmarks implemented in containerd.

CONTAINERD		
BENCHMARK	**DEFINITION**	**IMPLEMENTATION**
5.1	Do not disable AppArmor Profile (Scored)	/proc/{pid}/attr/apparmor/current
5.2	Verify SELinux security options, if applicable (Scored)	ps -eZ | grep {pid_container}
5.3	Restrict Linux Kernel Capabilities within containers (Scored)	/run/containerd/io.containerd.runtime.v2.task/default/{}/config.json -> ["process"]["capabilities"]["permitted"]
5.4	Do not use privileged containers (Scored)	/run/containerd/io.containerd.runtime.v2.task/default/{}/config.json -> ["noNewPrivileges"]
5.5	Do not mount sensitive host system directories on containers (Scored)	/run/containerd/io.containerd.runtime.v2.task/default/{}/config.json -> ["mounts"]
5,10	Limit memory usage for container (Scored)	/sys/fs/{cgroup_from_config}/memory/{cont_id}/memory.limit_in_bytes
5,11	Set container CPU priority appropriately (Scored)	/sys/fs/{cgroup_from_config}/cpu/{cont_id}/mcpu.share
5,12	Mount container's root filesystem as read only (Scored)	/run/containerd/runc/default/{}/state.json -> ["readonlyfs"]
5.15	Do not share the host's process namespace (Scored)	/run/containerd/runc/default/{}/state.json -> ["namespaces"]["NEWPID"]
5.16	Do not share the host's IPC namespace (Scored)	/run/containerd/runc/default/{}/state.json -> ["namespaces"]["NEWIPC"]
5.17	Do not directly expose host devices to containers (Not Scored)	/run/containerd/io.containerd.runtime.v2.task/default/{}/config.json -> ["linux"]["resources"]["devices"]
5,20	Do not share the host's UTS namespace (Scored)	/run/containerd/runc/default/{}/state.json -> ["namespaces"]["NEWUTS"]
5,21	Do not disable default seccomp profile (Scored)	/run/containerd/runc/default/{}/state.json -> ["config"]["seccomp"]
5,24	Confirm cgroup usage (Scored)	/run/containerd/io.containerd.runtime.v2.task/default/{}/config.json -> ["linux"]["cgroup"]
5,25	Restrict container from acquiring additional privileges (Scored)	/run/containerd/io.containerd.runtime.v2.task/default/{}/config.json -> ["noNewPrivileges"]
5,28	Use PIDs cgroup limit (Scored)	/sys/fs/{cgroup_from_config}/pids/{cont_id}/pids.max

Fig. 4. Partial Example of output from the Interoperable Application on a Docker container image

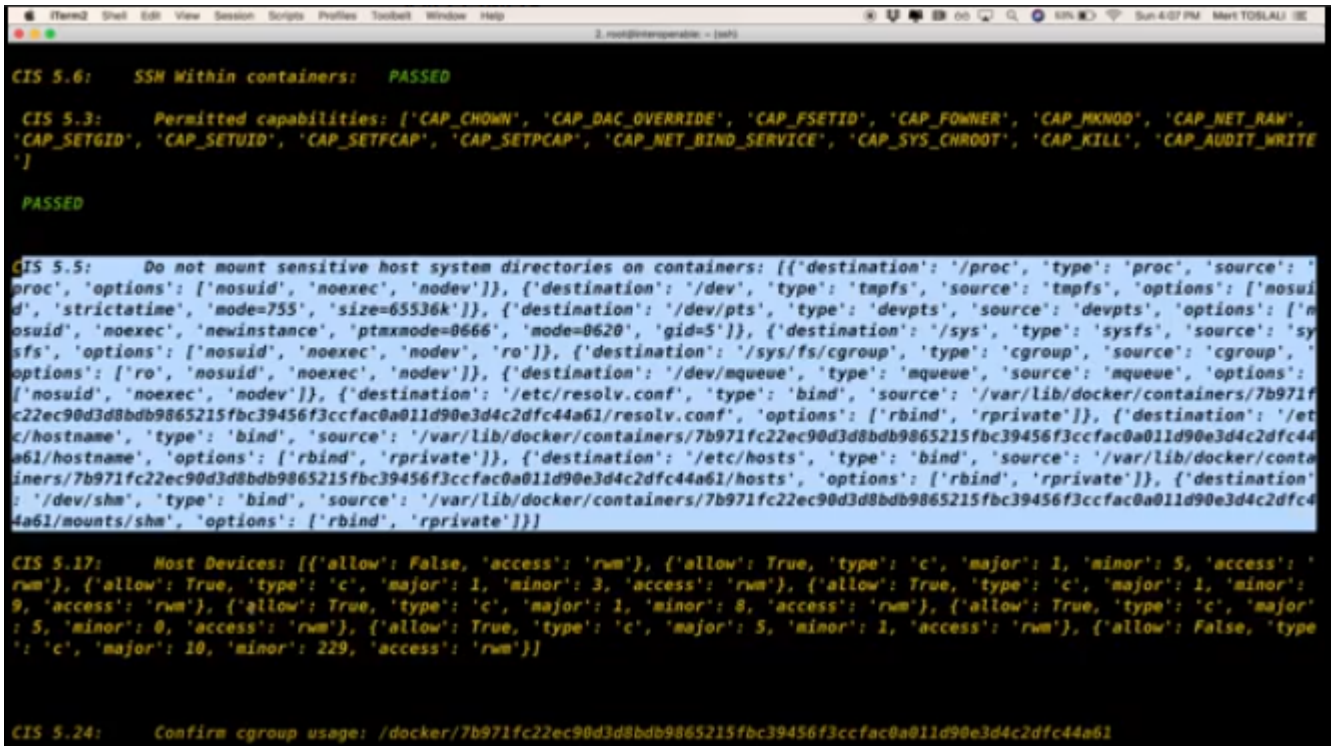

Fig. 5. Example of output from the Interoperable Image Application on a Docker container image