Understanding Container Isolation Mechanisms for Building Security-Sensitive Private Cloud

January 2017

M. Ali Babar and Ben Ramsey

CREST - Centre for Research on Engineering Software Technologies, University of Adelaide, Australia.

Recommended Citation Details: M. Ali Babar, B. Ramsey, Understanding Container Isolation Mechanisms for Building Security-Sensitive Private Cloud, Technical Report, CREST, University of Adelaide, Adelaide, Australia, 2017.

Executive Summary

Security is one of the key factors of consideration for a cloud-based Information and Communication Technology (ICT) infrastructure or applications. The security of a cloud deployment is dependant upon the security of each of the deployed components. This leads to the need for an overview of the security of the containerised technologies that are available to be deployed on a cloud infrastructure. In this second volume of our report on secure and private cloud for submarine mission systems, we report the findings from our study of container technologies; we will be covering the evaluation procedure, each of the container technologies architecture, the isolation mechanisms that are employed in the technologies, how each isolation mechanism is utilised by the container technologies, and suggestions for improvements to the isolation of each technology based on their weaknesses.

There are some major challenges faced while performing this form of analysis of different open source technologies. One primary one is that in open source technologies the documentation has a tendency to be out-of-date, incomplete, or created as a sales pitch for the technology. This can lead to statements within the documentation being misleading and as these technologies are open source the liability will be on the user of the technology. This lead to the analysis needing to be done by verifying each claim of an isolation related mechanism or feature which can be challenging as some of the features claimed are found in other technologies the analysed technology is dependent on.

Our primary finding from analysing the isolation levels of the container technologies is that each of the technologies have their own strengths and weaknesses in terms of isolation. In terms of overall security LXD can be considered the most secure due to the fully unprivileged containers able to be run, however this still has weaknesses, which are covered by other technologies. This shows that in order to select the correct technology for usage, the requirements for the actual workloads need to be considered.

In addition to our findings, this volume has allowed us to identify some key areas of future work in regards to security of container technologies. Some of the key areas include but are not limited to content trust systems, secure container communication protocols, and secure application containerisation processes and deployment.

This volume and its findings are to assist practitioners in the selection of container technologies to utilise for their particular use case by giving a sufficient understanding of the security risks and requirements of each of the technologies being analysed.

Table of Contents

1 Introduction

Container technologies have been increasing in popularity over the past few years. This is due to their portability, minimal resource requirements, simplicity for deployment, and low latency for spawning instances. Whilst a lot of companies are using container technologies such as Docker for application testing to ensure that the application will work on a variety of platforms, the technologies are not being used much in production environments. The reason for this is that there is a large concern about the security of these containers [1].

In recent months, container technologies are being integrated into private cloud software solutions such as OpenStack. This project seeks to evaluate the security risks involved with the usage of containers within OpenStack private clouds in a variety of methods. This project will also be evaluating the technological capabilities of containers in terms of their scalability in relation to the traditional type-2 hypervisor KVM.

Whilst the Docker project is the most prevalent container technology in modern use, other container options are available with different focuses on development. In order to determine the best option in terms of the scalability and security of the container runtime, we will be running experiments on multiple runtimes within different container deployment systems.

2 Project Description

The DST Group wish to investigate the usage of container technologies in private cloud deployments and their impact on the security and scalability of cloud computing in general.

The components of the project include:

- Hardware procurement;
- Laboratory environment setup;
- Investigation into the security of container based technologies and their impact on a private cloud environment;
- Investigation into the security isolation of containers to the host systems;
- Development of container isolation improvements;
- Conduct study on both the technological capabilities and risks of using lightweight containers both in isolation and in private cloud deployments.

3 Related Work

Previously reported container security evaluation work focuses only on a single technology such as Docker [2], [3], or LXC [4]. The work on the operating system level containers reports little detail. For example, “Securing Linux Containers” [5] focuses on outlining possible methodologies of improving the security of different container technologies on Linux. Firstly the details of the methods are not sufficient enough for decision making to take place. And the provided suggestions do not go into any detail of what protections are already in place for the isolation of the containers.

This report provides a detailed analysis of the isolation mechanisms of container technologies and suggests improvements. Some related work by other authors looks into areas such as image security[6], [7] and content trust[5], [8], [9].

4 Evaluation Methodology

4.1 Outline

In order to provide a meticulous and high quality container security evaluation based on the container technologies in a cloud and in isolation, there is a need of a well reasoned procedure. We devised and used the following steps to perform the required tasks for container security evaluation:

1. Identify and carefully read the relevant literature to determine and understand the commonly reported vulnerabilities of the technology to be reviewed;
2. Identify the key isolation mechanisms utilised by container technologies;
3. Analyse the usage of the key isolation mechanisms by each of the technologies; and
4. Suggest improvements for the usage of the key isolation mechanisms of each of the technologies.

4.2 Literature Reviews

As the first step of the process, the reading of the relevant literature can be used to find common vulnerabilities either in the reviewed technologies, or in similar technologies. This step can be used to focus the search of the potential vulnerabilities. Some of the sources of information that were utilised include:

- Academic Papers
- Blog Posts
- Release Notes
- Git Commit Logs

4.3 Isolation Mechanism Identification

In this step of the evaluation process, we identify some of the common mechanisms that the container technologies utilise in order to isolate the processes within the container from the host system and other container instances. In order to do so, we need to read security related literature to see which technologies are mentioned. Additionally, we can also look at the source code for some libraries, which interface with known isolation mechanisms.

4.4 Isolation Mechanism Analysis

In this step of the evaluation process, we take the identified isolation mechanisms from the previous step and analyse how each of them are being used by the container technologies if applicable. At the start of each isolation mechanism section, there will be a summary table that allows for a simple comparison between the different technologies in regards to the particular isolation mechanism.

4.5 Isolation Mechanism Improvement

In this step of the process evaluation, we suggest possible improvements to the usage of the identified isolation mechanisms in order to improve the level of isolation between the host system, the container, and other container instances.

5 Rkt Overview

The Rkt container system is a new container technology developed by CoreOS as an alternative to Docker. It has been designed with a security focus right from the beginning. Different from the Docker engine, which implements the open container specification, rkt is an implementation of the appc specification for application containers.

Other notable implementation of appc include Jetpack [10] and Kurma [11]. However, rkt is the most mature of these implementations and is commonly referred to as the reference implementation.

The rkt application container differs significantly from the other container technologies being analysed as it does not require a daemon process in order to control the lifecycle of the container. Instead each container in rkt is a separate process. This difference is utilised in order to easily integrate within the init systems on Linux (e.g., systemd and upstart), allowing the lifecycle of the rkt container to be completely controlled by these systems. If we attempt to utilise one of the init systems with Docker, then the init system will not have complete control over the lifecycle of the container as all commands will be passed to the Docker daemon.

This gives rkt an advantage over Docker, as this allows for fine lifecycle control of the container applications through service control systems. Additionally, as rkt does not require a daemon to run the applications the overhead of running rkt containers is significantly reduced, as is the possible attack surface.

An unprivileged user can perform some of the functions in rkt. This leads to a more secure container system as running all operations as a privileged user can lead to some flaws in the security having significant impact on the system. The operations that can be done as an unprivileged user include image fetching and signed image verification. This means that if there is a flaw in this component of rkt, then there will be no privilege escalation from a crafted image. However, flaws in the container runtime components will cause a privilege escalation as these operations still need to be run as the root user at this point in time.

Rkt utilises the union filesystem implementation OverlayFS (see Figure 1) [12], which has been merged into the Linux kernel. The usage of union filesystems allows the container to have a single unified view of multiple filesystems.

OverlayFS can be split into two layers, the upper layer is the overlay of the filesystem, with any file residing within this layer having precedence over any in the lower layer. The upper filesystem may be writable even if the lower filesystem is not. If an object is requested to be read only then the following steps are performed by the kernel:

1. Search the upper filesystem for the object;
2. Search the lower filesystem for the object.

OverlayFS also supports multiple lower filesystems allowing many different directories in the host system to be mounted into the overlay. If this is the case when searching the lower filesystem, each of the filesystems will be searched in the order they were specified during the mount. When an object may be modified, i.e., requested as read-write, and the object does not reside in the upper filesystem, then the object will be copied up to the upper layer. This is so that if the lower layer is read only then this is preserved but within the overlay it can be modified as required.

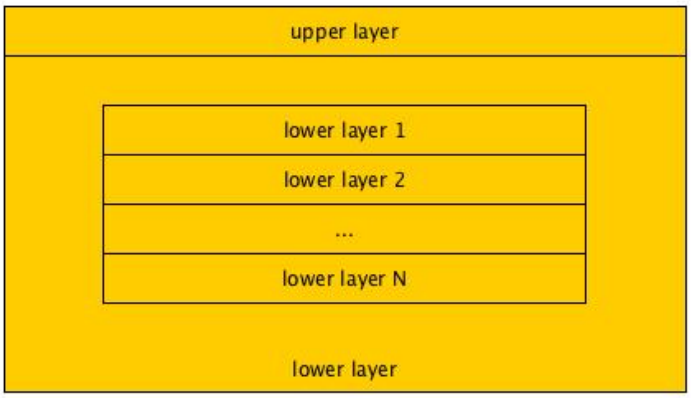

Figure 1: OverlayFS layered architecture view.

When writing to a file in the merged filesystem, the following rules are followed:

- If the destination does not exist in either layer, then a new file is created in the upper layer;
- If the destination exists in the lower layer, but not in the upper layer then a copy is created in the upper layer and the modifications are written to the new copy; and
- If the destination exists in the upper layer, then the modifications are written to the upper layer copy regardless of whether it exists in the lower layer.

At this point in time, the rkt is unable to utilise OverlayFS while still using user namespaces. This implies that using OverlayFS with rkt cannot be done with an unprivileged container. However, running containers while using user namespaces won't allow the container to be fully unprivileged as the rkt run command needs to be run as root at this point in time.

5.1 Execution Stages

During the creation of a rkt container, the rkt binary will go through three different stages of execution each with different purpose and resultant output.

At the first stage (stage 0), the rkt binary performs the necessary initial preparation work in order to run the container. Presently the following steps are performed:

- Fetch Application Container Images (ACIs);
- Generate Pod Universal Unique Identifier (UUID);
- Generate Pod Manifest;
- Create required filesystems;
- Setup the stage directories for stages one and two;
- Unpack stage one ACI; and
- Unpack ACIs and copy applications into stage two.

At the second stage (stage 1), the rkt binary will take the filesystem built at stage 0 and begin building the operating environment for the container. The steps that this stage performs are as follows:

- Read ImagePod manifests;
- Generate systemd unit files;
- Create and enter network namespaces unless networking is set to host;
- Start systemd-nspawn or LKVM (native Linux KVM tool);
    - Setup external volumes;
    - Launch systemd as PID 1 within cgroup/namespace; and
    - Systemd within pod launches applications.

The isolation mechanisms that are used by rkt are dependent on the type of stage 1 image that is utilised. There are three possible images for this which are illustrated in Figure 2:

1. Stage1_fly;
2. Stage1_coreos; and
3. Stage1_kvm.

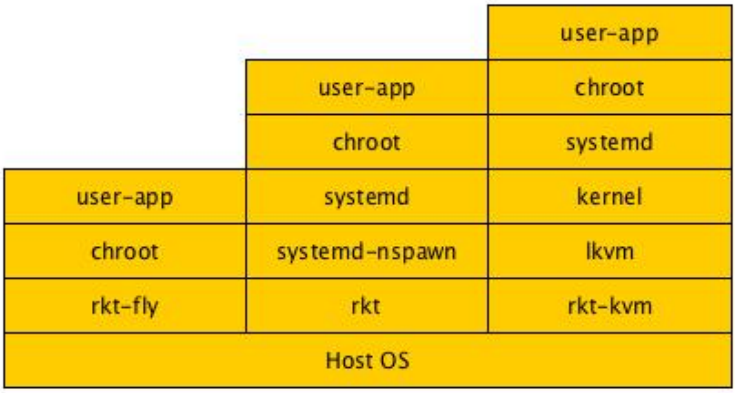

Figure 2: Levels within different stage-1 images for rkt application launching.

From Figure 2, we can see different layers that are utilised within different stage-1 image types. The row above the host system seems to show three different binaries, however, in reality they all use the rkt binary with different stage-1 images path options being set.

The stage1 fly image allows for a unisolated container to be started. This is similar to that of the –privileged containers of the Docker engine.

This image uses the chroot mechanism to launch a user's applications. The reasoning for using the fly feature is to run fully privileged process on the host with the image management and discovery of rkt [13] e.g. for auditing or monitoring processes.

If using the fly feature, many of the rkt isolators and security mechanisms are unable to be utilised. These include network namespace isolation, CPU isolators, Memory isolators, Linux Capability bounding sets, and SELinux. However, if some security is required to be added then the fly container can be launched such that it is wrapped in a systemd unit file [14], [15]. While not as secure as the default isolation mechanisms of rkt, this will provide some protections from malicious applications if the rkt-fly image must be used.

Due to the fly feature using chroot to separate the execution environment, the usage of this has a high risk. chroot has a well-documented way to easily break out of the environment [16]. While each of the stage 1 images is susceptible to a chroot breakout (see Figure 2), the largest security impact occurs with the rkt-fly image.

As at this point in time the rkt run command requires administrative privileges to run, a breakout will cause the user to gain root access. This will circumvent restrictive sudoers configuration for a user only allowed to run rkt as root. In order to prevent this privilege escalation, the sudoers file should not allow a user to utilise a fly image. i.e. should only allow the actual root user of the system to launch this type of container.

The stage1_coreos image is the standard image of the rkt container system. In order to separate the execution of the container from the host system, a systemd_nspawn instance is launched creating the required cgroups and namespaces. The application is launched inside of a chroot environment controlled by systemd. The isolators for rkt are all available for this container image type, as well as the utilisation of SELinux for further protections.

The stage1_kvm should be used for highly sensitive workloads as it allows rkt to launch the ACI with CPU-enforced isolation provided by the Intel Clear Containers initiative [17] with the Intel VT providing isolation for the container. These containers are heavier than the standard image type as instead of going from systemd-nspawn straight to systemd instead it uses LKVM and has its own kernel before utilising systemd. The reasoning of using rkt with KVM is to allow a high degree of isolation for application centric packaging mechanisms, such as the ACI utilised by rkt.

In order to keep the containers lightweight, the stage1_kvm image is designed to be extremely lightweight with limited delays due to virtual machine starting. This has been done by using the kvmtool, which is a small binary for creating native kvm virtual machines. Each pod using the KVM flavor of stage1 has its own virtual machine setup for isolation.

At the last stage (stage 2), the rkt binary is said to be in a running state, this is where systemd changes the root to the container rootfs and executes the supplied exec command. This is the stage where a rkt container is defined to be launched and running the requested applications.

5.2 Pods

In some use-cases, there are some tightly associated applications that should be placed together. In order to accommodate for these use cases, rkt provides the concept of pods. While all of the rkt instances have a pod with a single application, it is possible for a pod to have multiple applications inside. The main idea behind pods is to have a collection of application images being run on top of the stage1 image container, i.e., a pod is a shared namespace for multiple application images.

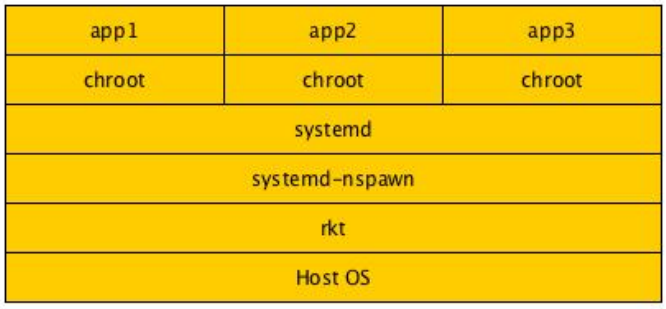

Figure 3: Layer diagram for rkt pods.

From Figure 3, we can see that within a pod the only isolation mechanism being employed is the chroot jails. While this can be a security risk in some cases, a pod cannot be modified to include additional images after it has been launched. The only way applications can be executed after the initial rkt run is to use the enter command. This creates an additional chroot jail within the pod namespace.

This is unable to run any applications that are not already within the stage 2 rootfs of the images within the pod. This highlights the requirement to only have applications that are required during runtime of the pod to limit the attack surface of the container.

The architecture of the pod means that each application within a pod is lightweight and the pods are much lighter weight than if all of the applications were run separately. While each of the applications is being controlled by the same instance of systemd, and are in the same system slice, each of the applications has their own unit file. This will allow future work into isolation mechanisms, such as SECCOMP filtering [18] to be performed at a per-app basis. This feature has a dependency on the appc specification [19] as it will need to be placed in the image manifest.

6 Docker Overview

The Docker container runtime is the most mature technology among the three container technologies we are evaluating for this project. As with all of the other runtimes Docker is an image based system where the image defines the contents and environment of the container instance. The main component of the Docker container runtime is the Docker engine. The Docker engine is a daemon running on the host system under the root user. All of the operations of Docker go through this daemon including the container management, image management, and image building.

The container management of Docker includes the creation, starting, stopping, deleting, and pausing of containers. This lifecycle management of containers from the daemon, mean that both systemd and fleet are unable to have fine-grained control over containers as each action will need to go through a daemon that is a possible point of failure.

The image management of Docker involves the fetching, storing, and deletion of images onto the host system. For Docker to be able to run an image, the image must be imported into the local image store. This can be done in three ways; using the import command, automatically with the run commands, or by using the build command.

The image building within the Docker engine is undertaken when a user invokes the build command referencing a Dockerfile. The Docker engine is responsible for performing each of the required actions to build the image specified by the Dockerfile.

The format of the Dockerfile is that each line in the file represents a single command. Each subsequent command which has an effect on the contained environment adds a new layer to the image being built from the file. When the build command is executing, the changes to the file system that are being made by a command is a new layer. This allows for container image consistency with versions as the commands are placing their effects within the filesystem rather than telling the container to run the command on launch.

For example, a simple apt-get update on a Debian based container will update the programs within the container image at build time allowing for this version to be consistent at later times. Development and Operations (DevOps) teams make good use of this feature of Docker as it allows for a known packaged environment to be used for testing purposes.

Additionally, an image can be used as a base for another image, which can enable developers to build a known baseline image for their application and still have multiple additional layers depending on their requirements.

Docker utilises a different implementation of the union filesystem type than rkt, instead it utilises AUFS [20]. This implementation is not merged into the Linux kernel like OverlayFS is. This implies that AUFS may not work on all Linux distributions. In comparison to OverlayFS, AUFS has the concept of layers being classified as branches that are union mounted into a single directory.

Like OverlayFS, AUFS does allow for read only branches/layers, with objects in these read only branches able to be written to logically but no changes to the actual object are made. This concept of branches being merged into a single mount point is what allows Docker layered image view to be implemented. Each layer of the image can be expressed as a layer of the operating system, with each subsequent layer being placed on top of the previous one. This means that on a read, if a file is present in multiple layers only the highest version in the layer hierarchy of the file will be read as reading searches will be done top down.

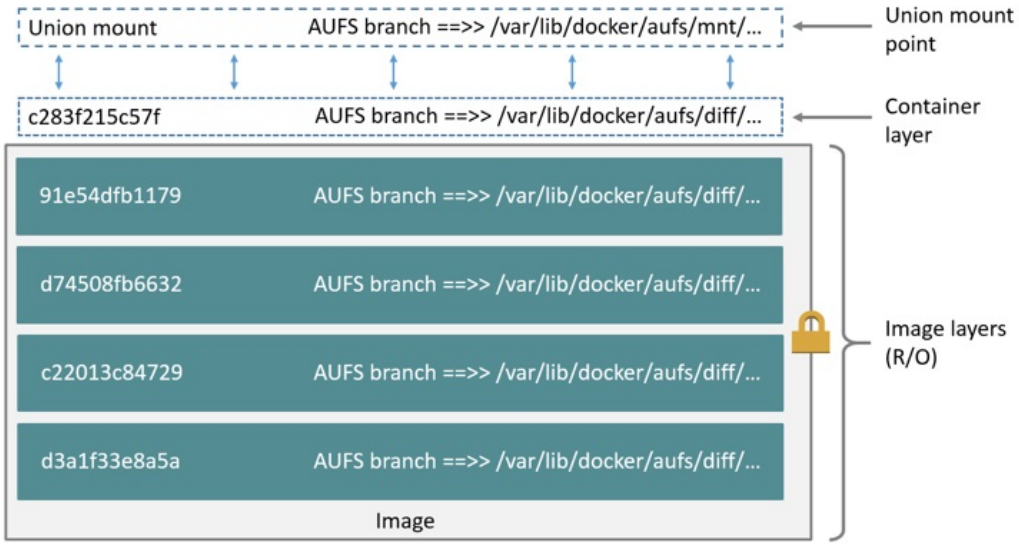

Figure 4: AUFS image layering union mount layout [20].

This concept of layers (Figure 4) means that once all of the different layers are downloaded and placed within the filesystem as required, the applications running inside the container will have a unified view of all of the layers within the system. If two layers have the same file in common, then the version of the file in the layer that is closest to the container layer will be providing the file.

The concept of multiple layered images does pose some security risks. The primary risk is that an image that has been created by someone is not guaranteed to be secure at all. The task of auditing such an image is further compounded by the ability of using base images as in order to audit the image every layer will need to be taken into consideration, which could mean auditing many images if the chain of base images is long.

Our recommendation for images is to put in place rules for the usage of images. These rules may include; all public images being used must be official and signed with Docker Notary [8], all developed images must be correctly versioned and signed, and all base images must be using a particular version of the base (not latest).

The signing of images using Docker Notary ensures that the contents of the image has a verified origin and contents. In the case of an official image, this allows the user of the image to trust that the contents of the image are not malicious.

The versioning of images, both base and developed, is to prevent issues with environment inconsistencies as well as having a known base from the auditors' point of view. The auditor should look at each version of an image and decide, which image versions are suitable as a base, with the list changing over time as security risks and vulnerabilities are discovered.

Also there are suggestions not to use Dockers' pull command as that automatically unpacks the image leading to a possible compromise to the system [21]. Instead, downloading the image as a tar file from a trusted source and then loading into the Docker daemon is a much safer method of obtaining the image. If the image cannot be found at a trusted source it is possible to perform this step yourself by using Dockers' pull on a virtual machine and exporting it once it has been verified to be correct and secure.

7 LXD Overview

The LXD container daemon is a new system for enabling simple creation of LXC containers developed by Canonical. The LXD system consists of two main components:

- A system wide daemon
- A command line client

The system wide daemon provides a RESTful API which can be used either locally, or remotely to manage the lifecycle of various containers. From the announcement[22] of LXD the following features have been implemented:

- Secure by design;
- Scalable;
- Image Based; and
- Live Migration.

By being image based, LXD removes the most challenging aspect of launching LXC containers. This aspect is the task of creating the operating system manifest from an XML document.

The live migration of containers allows for containers to be moved between hosts with minimal downtime. The idea behind it is that a container can be moved from one host to another without being shutdown first. A common way of achieving this is to keep the original container running until the new container has finished starting, and then switching to the new container for providing the required service. This feature highlights that LXD is designed to be utilised in a multi-host setup like a cloud. In fact, LXD was developing an OpenStack driver for the compute engine from the start of development.

The primary use case of LXD is different from that of the Docker and rkt container runtimes. That is, in the rkt and Docker instances these are primarily used to deliver isolated applications to the user. In contrast, LXD is primarily for delivering a full operating system within a container. However, this is not to say that rkt or Docker are not capable of delivering a full operating system within a container, just that this is not the main use case of these systems.

These differences in use-case imply that in the common use cases for the various container runtimes LXD sits between the application containers and the virtual machine instances in terms of resource requirements. The primary difference between LXD and the type-1/2 hypervisor machines is that the type-1/2 virtual machines will have the hardware virtualised which the operating system installed onto it will use instead of the physical hardware. Then it is only the virtualised hardware which has any connections to the physical hardware on the machine the instance is being run on. In contrast LXD has a full operating system emulated within a LXC instance. This LXC instance is still a process which uses the host operating system, and is similar to one of the application container runtimes with a full operating system running on it.

One primary difference between the implementations of LXD and rkt/Docker is the filesystem type being used. Whilst rkt and Docker utilise different implementations, they both use a union-based filesystem with Docker using aufs [23], and rkt using OverlayFS [12]. LXD uses LXCFS [24], which is a Filesystem in USErspace (FUSE) based filesystem.

LXCFS provides an unprivileged filesystem to the container runtime, which provides a cgroupfs compatible view for the unprivileged containers and a set of cgroup aware files. This implies that it provides an emulation of both the /proc, and /sys/fs/cgroup folders for unprivileged containers [25]. This allows LXD to run their containers as unprivileged, whereas the union filesystems that rkt and Docker utilise do not have this emulation and thus needs to retrieve this data from the host.

7.1 Image Specification

An image for LXD can be specified either as one, or two tarballs. In the single tarball image, the content includes a rootfs directory, a metadata YAML file, and an optional template directory. In the split tarball image, the rootfs directory is placed into a separate tarball from the rest of the image content.

This capability of splitting tarballs allows different images to be created from the same metadata, or alternatively from the same rootfs [26]. The rootfs directory or tarball contains an entire Linux filesystem as viewed as if the directory is the containers '/' directory. The full filesystem is required as LXD is a system container engine, rather than the application container engine such as Docker and rkt.

The templates directory contains pongo2 formatted template files, which can be used to modify the contents of particular images based on the metadata YAML file. The metadata YAML file contains all of the information required by LXD for the running of the image.

The two main sections of the YAML file are the properties section and the templates section. The properties section includes information such as description, operating system name, and operating system version. The templates section of the file specifies the output path, rendering times, template file, and the properties.

The output path is the location on the rootfs that the template should be rendered into. The render times specify under what conditions the templates should change. The template file specifies the location to find the template to render within the templates directory. The properties are key-value pairs that are used to replace the place holder values in the templates with what should be used.

8 Isolation Mechanism Overview

8.1 Mandatory Access Control Systems

The term Mandatory Access Control [27] (MAC) is used to refer to security policies which do not rely on voluntary compliance. In contrast, in the Discretionary Access Control (DAC) model utilised by the Linux file permissions system, a user may be the only ones with read/write access to a file, but the root user is able to change the permissions.

MAC systems ensure the security of the objects by assigning labels referring to their sensitivity and comparing this to the clearance level of the user. All information on a MAC secured system will be labelled allowing for the entire system to be secured against particular types of attacks, assuming the MAC implementation is correct and secure.

A MAC control model has the following attributes associated with it:

- Only administrators can change the security labels for resources;
- All data is assigned a security level that reflects its sensitivity, confidentiality, and protection value;

- All users can read from a lower level of classification than the one they are on;
- All users can write to a higher classification;
- All users are given read/write access to objects only of the same classification;
- Time based access control is available; and
- Security characteristic based access control is available.

While containers are able to be run without a MAC implementation being in place, it can lead to a less secure host system in the event of a container boundary being compromised. This is due to the ability of a MAC system ensuring that a process is unable to use system resources that it should not be able to.

While there are many implementations and policy styles for MAC systems, the two we have looked into are SELinux [28] and AppArmor [29].

8.1.1 SELinux

SELinux utilises many different security models in order to perform its job but the primary one that ensures tight security bounds is the type enforcement model. The types are a way of classifying an application or resource using enforcement to define what application label can access what resource label. This type enforcement model also uses inodes to refer to files rather than path names so that a hard link is not able to trick SELinux into allowing access.

SELinux utilises the concept of a centralised policy; this policy may be split over many different files but on the system there is only one policy being implemented at once. This allows for a simpler management of the security of a system. The policy is what tells the system how different components are able to interact.

8.1.2 AppArmor

AppArmor is another MAC implementation that is in competition with SELinux. Instead of being focused around a type enforcement model, AppArmor utilises a name-based enforcement model. Whilst this does allow for a much simpler security policy to be implemented, it is susceptible to being fooled by hard-links to resources.

For example, if a user does not have access to a particular file but creates a hard link of it elsewhere then AppArmor would consider the hard link and the file to be separate entities. However, in a type based enforcement system these would be considered the same entity and the restrictions would be contained.

The main difference between AppArmor and SELinux apart from the enforcement model is that for changes to the policy of SELinux a system reboot is required, whereas in AppArmor policies can be loaded/unloaded as required during uptime [30]. This can lead to security issues if the profile is modified by an unauthorised user, this implies that the policy files need to be protected from unauthorised access and modification.

8.2 cgroups

A cgroup [31] is an extension to the Linux kernel that allows for the aggregation and partitioning of various sets of tasks and their children into hierarchical groups with a specialised behaviour. The default setup is for job monitoring but other subsystems can be hooked into it to allow for limitations to be placed on various groupings.

In terms of containers the cgroups are used to place limits on different system resources in order to support fair resource sharing and prevent some denial of service attacks. However, they do not prevent containers from having an effect on another in some way [32].

8.3 Namespaces

Kernel namespaces are one of the primary methodologies that container systems utilise for isolating containers. A namespace is the way that the kernel allows varying perspectives on a system from different processes. This can be done with containers as each container is a process.

The namespaces are managed by the container implementations and are set up prior to any application being run within a container. This is one of the earlier steps in setting up the isolated environment for containers. Linux is capable of providing the following namespaces [33]:

- Inter-Process Communication (IPC);

- Network;
- Mount;
- Process Identification (PID);
- User; and
- Unix Timesharing System (UTS)

The IPC namespace isolates various different inter-process communication resources. For example, System V IPC objects and POSIX message queues. IPC objects are identified by mechanisms other than file system pathnames. Each IPC namespace has its own set of resources. Any object created in an IPC namespace is visible only to processes that lie within the same namespace. On the destruction of the namespace, all associated objects will also be destroyed.

The network namespace isolates different system resources associated with networking, most notably the network stack. A physical network device can only reside in exactly one namespace, however, the usage of virtual network interfaces can allow for the creation of tunnels between namespaces and create a bridge to a physical network device. When a namespace is freed its physical devices move to the initial network namespace.

The mount namespace isolates a set of file system mount points. This allows for different processes to have differing views of the file system.

PID namespaces [34] isolate the process ID number space allowing multiple processes to have the same PID in different PID namespaces. This feature allows for container suspension and migration to occur without modifying the PIDs of the applications within the container.

In each PID namespace there is an init process, which will always have the PID of one. In most cases an orphaned process within the namespace will be re-parented to this process. This will occur unless one of the ancestors of the orphaned child in the same PID namespace marked itself as the reaper process. If this init process is killed, then the kernel will ensure all other processes within the namespace are also killed using the SIGKILL signal. If the init process is killed, then no additional processes can be created within the namespace. In order for a signal to be heeded by the init process, a signal handler must have been installed for that signal. The SIGKILL and SIGSTOP signals are the exception to this in the case of an ancestor namespace sending the signal.

User namespaces [35] isolate the security-related identifiers and attributes. This includes the user/group IDs, the root directory, keys, and capabilities. A process's user and group IDs can vary inside and outside of the namespace allowing an unprivileged process to have full root privileges within the namespace. A process within a namespace may have full capabilities within a namespace but will not have any capabilities outside of the namespace. In recent releases of the Linux kernel (3.8+), an unprivileged process is able to create a user namespace and the other types of namespaces can be created within this user namespace. This ability is how unprivileged containers can be created. Additionally, if a single clone call has CLONE_NEWUSER flag specified with other CLONE_NEW* flags, the user namespace will be created first and the rest of the namespaces will be given within the new user namespace.

The usage of user namespaces can allow for unprivileged containers to be created. This does not always mean the container is being run by a non-root user, however, that is the best feature. Instead it is defined as residing inside of a user namespace and as such can have a root user (UID 0), which is separate from the host root. This can be done by having a range of subordinate UIDs and GIDs, which the container user namespace will be mapped to. The UTS namespace provides isolation for the hostname and the NIS domain name. This allows for a container to act as a different host than the host system of the namespace.

8.4 Seccomp

Seccomp [36] is a kernel feature enabled in most modern Linux distributions that provides a mechanism for filtering the system calls a process is allowed to call. As the inside of the container is a root process, the ability to utilise system calls is unhampered by the protection mechanisms. By default, seccomp only allows the following four system calls:

- read()
- write()
- exit()

- sigreturn()

This is far too restrictive for most programs, this lead to the Berkley Packet Filter (bpf) extension of seccomp being created, which allows for custom filters to be put in place at a per-thread basis.

The importance of seccomp is that many of the system calls can be dangerous due to the process calling functions that reside in the kernel space, rather than the unprivileged user space that most programs reside in. These additional filters can be built using either a whitelist or blacklist style of filters.

A whitelist is a list of calls that are allowed with any call that is not in the list prevented by default. However, the blacklist is a list of calls that are not allowed with any call that is not in the list allowed by default. From a security point of view, a whitelist is preferable as it is easily documented which system calls are allowed.

However, a blacklist is easier to manage as a call that if found to be dangerous can be placed on the list, whereas with the whitelist each time a new system call is added to the kernel then it needs to be decided whether or not to add it to the list. However, this can lead to a whitelist which is too restrictive as was the case for the original seccomp.

8.5 Linux Kernel Capabilities

Linux kernel capabilities [37] are a mechanism of the kernel to allow for the various privileges of the root user to be split into distinct units. These units can be independently enabled and disabled at a per-thread basis. In Linux systems the root user is the user which has all of the capabilities associated with it. However, if a program requires only a subset of the capabilities of root capabilities allow the kernel to grant only the abilities of root that are required for the process.

While some of the capabilities are only able to perform small numbers of operations there are some capabilities which give the running process more possible privileged operations. For example, the CAP_NET_ADMIN and CAP_SYS_ADMIN are able to perform multiple operations. The CAP_NET_ADMIN capability is able to perform many different types of network related operations, an exception to this is the sethostname[38] syscall which requires CAP_SYS_ADMIN. Additionally, the CAP_SYS_ADMIN grants the ability to perform many actions that if granted liberally can cause a huge security impact to the system.

Capabilities are used by containers to ensure that the container runtime is only able to perform the actions that it requires. This can be done as all instances of containers are a process with unusually utilised properties in order to provide isolation.

8.6 Container Linking Mechanisms

While isolation of fully self-contained applications is simple, in most use cases the applications within the instances will need to communicate outside of their own container. The way that the container runtimes allow for the linking of containers is a major concern for the isolation of the system.

In some of these cases the required knowledge from different containers is only partial, as in a container only needs the output of one of the processes running within a container. Some of the possible solutions for this problem are as follows:

- Namespace sharing
- Network connections
- Usage of a message broker

The problem with each of these solutions is that there needs to be additional controls to ensure that only allowed containers are able to connect to another. Thus the solutions should have the framework built into the container runtimes to allow for this being a simple exercise for the user without compromising the security of the container instances.

9 Mandatory Access Control Systems

The ability to utilise MAC systems by all of the examined container technologies (see Table 1) show that at least some thought has been put into improving the security of the system. If only DAC systems were available

then the security of the containers could not be trusted as they may ignore some of the access controls put in place if they needed to, leading to an issue in the isolation of the containers.

The usage of SELinux or AppArmor is one that depends on the use case for the system. While the isolation levels for SELinux tend to be higher due to the type enforcement policy model it is also more difficult to maintain and modify on a running system.

Table 1: Mandatory Access Control Systems Summary.

	Docker	**Rkt**	**LXD**
MAC supported?	Yes	Yes	Yes
Per-container contexts supported?	Yes	Yes	Yes
MAC enabled secure by default?	Yes	Yes	Yes
Default MAC Profile Available[1]?	Apparmor only	No	Apparmor only
Custom MAC Profile Supported?	Yes	Yes	No

Having per-container, or per-instance contexts of the MAC policy systems is important as these contexts if defined properly will ensure that any processes within the contexts are unable to interact with objects within any other contexts including the Host OS. As all of the container technologies are able to take advantage of this, there is no advantage between the implementations.

The container technology having a default MAC policy file means that the developers of the container technology have acknowledged the need to secure the container outside of its runtime. Having a default profile will add additional security as these profiles will have been generated with how the technology works in mind.

Allowing for the usage of custom MAC profiles means that the system administrator is able to increase, or decrease the level of security from the MAC security contexts. This can be used to further restrict the access of particular resources to the container instances. Having custom profiles supported, allows the containers to be utilised in more use cases where the compliance levels are different.

9.1 Docker Mandatory Access Control Usage

The Docker container engine is able to take advantage of either SELinux or AppArmor depending on the security requirements of the user or systems administrator. This level of customisation is good as it means that Docker is able to run securely in regards to MAC within multiple operating environments as long as at least one of these MAC systems has been installed and properly setup.

Under a default configuration however, Docker utilises the AppArmor MAC implementation if it is installed on the host system. However, in the case that SELinux is installed instead of AppArmor no MAC system is utilised by Docker. In order to utilise SELinux, the Docker daemon must be started with the –selinux-enabled flag set to true. There is no default SELinux profile for Docker, these can be developed by the systems administrator or by the package maintainer for the host operating system. There is a profile created with Docker in mind by Redhat which ships with Redhat operating systems, or is available to download.

9.1.1 Default AppArmor Profile

The Docker git repository contains an AppArmor profile[39] which gives reasonable security for the container setup and runtime using Docker. Some of the rules include:

- Prevent following links to files in /etc, /dev, /sys, and /proc during container setup

By preventing the following of links to files within the host /etc, /dev, /sys, and /proc directories the AppArmor profile prevents information leakage due to a crafted Dockerfile or image which attempts to mount linked files into the container.

1 This is referencing the default profile attached to the container implementation. Most operating systems will have a default profile setup which can be utilised by the container technologies.

9.2 rkt Mandatory Access Control Usage

For rkt the MAC system that is referred to in its documentation is the utilisation of both SVirt[40] and SELinux. The SVirt mechanism is utilised with virtualisation technologies to integrate them into the SELinux mandatory access control system.

The consequences of rkt using SVirt is to reduce the ability of an attacker to obtain full access to the system. In order for an attacker to gain full access from an application or service, they must:

1. Find and exploit a vulnerability in the application/service
2. Find and exploit a vulnerability in the Linux kernel namespaces
3. Find and exploit a vulnerability in the SELinux KSM

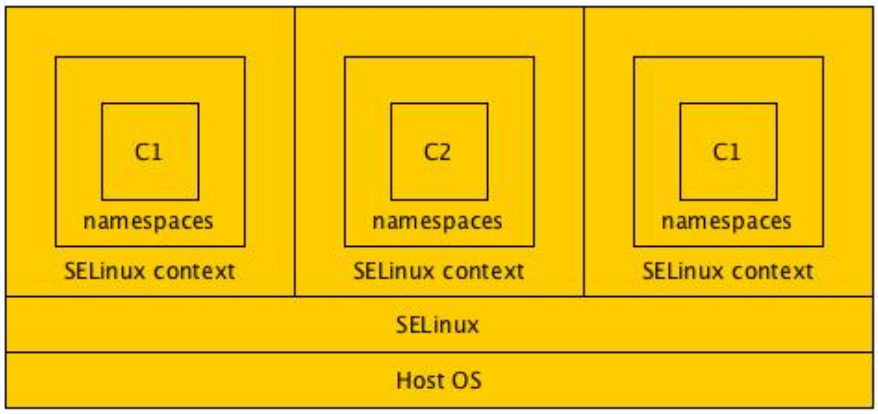

Figure 5: Containment model of rkt containers when utilising SELinux.

From Figure 5, we can see that each container instance has its own SELinux context being defined. The usage of unique contexts allows for each containers resources to be set to have their type set to that of the unique context. This allows for only processes within the context to be able to access the resources with this type. Additionally, a process within this context will be unable to access resources that are not explicitly stated to be accessible to the type for the context.

This is true regardless of the user that launched the instances. Additionally, the attacker will be unable to launch any additional applications on the host system. In order for this SELinux to be useful it must be set to enforcing mode, unfortunately many people disable SELinux due to the difficulty factor of running some applications.

9.2.1 CoreOS Supplied SELinux Policy

Rkt does not have a default profile, as the contents of an effective profile can vary greatly between operating systems. In the alpha release of CoreOS (955.0.0) they have provided a SELinux policy which includes rules for the svirt contexts that the rkt containers are placed in. The rkt instances have the following SELinux context (obtained using ps –eZ):

system_u:system_r:svirt_lxc_net_t:s0:cxxx,cxxx

According to the documentation[41] format of the context is:

<user>:<role>:<type>:<level>:<categories>

The two parts that we are interested in are the type and the categories. The categories of the context is the method that CoreOS uses to ensure that each container instance resides within its own context without access outside of the context. This is done by using the same SELinux security context level for all containers.

Due to SELinux being a type based enforcement system the type of the context will tell us what rules the process is required to follow. Using the sesearch tool[42], we found that there is a total of 267 semantics (rules) associated with the type. Some of the key rules include:

- Sysfs_t type is read only (/sys)
- Proc_t type is read only (/proc)
- Var_lib_t can be read, locked, and executed upon (/var/lib)

- Usr_t can be read (/usr)
- Etc_t can be read (/etc)

The var_lib_t type has more liberal rules due to the way that rkt implements the lifecycle of its containers. That is due to there being no daemon to keep track of the state of the container rkt instead utilises file locking and renames to keep track of the state of instances[43]. The usage of SVirt allows for the policies defined on CoreOS host systems, are able to be used by Docker to improve the security of Docker containers running on the host as well.

9.3 LXD Mandatory Access Control Usage

While LXD does support the usage of both AppArmor and SELinux, it does however have a strong link to being used in conjunction with AppArmor over SELinux. This is due to Ubuntu having AppArmor by default, and LXD is developed by Canonical.

Upon investigation we were unable to find the contexts that LXD needed for usage with SELinux. However, the default setup for LXC is that the SELinux profile is unconfined_t which means that LXC will not attempt to change contexts[44]. As well as this the conversion between LXC to LXD does not support custom SELinux contexts, so this default will be utilised[45].

9.3.1 Default AppArmor Profile

The default profile contains the following:

- Deny mounting of filesystems of type devpts;
- Deny mounting of the proc filesystem with read+write;
- Deny mounting of the sys filesystem with read+write;
- Allow mounting of proc filesystem types to /var/cache/lxc;
- Allow mounting of the sysfs filesystem types to /var/cache/lxc;
- Allow the mounting options rw and bind;
- Allow the mounting of the ext family of filesystem types;
- Allow mounting of the xfs filesystem type; and
- Allow mounting of the btrfs filesystem type

These rules allow for the isolation of the listed filesystems of the host or other containers from a container. In particular, if devpts file system types were allowed to be mounted then the pty (pseudo-terminal) devices of the host system would be able to be pulled into the container.

10 cgroups

As cgroups are a part of the Linux kernel, they are utilised by all of the container runtimes (see Table 2) to provide the resource limits for container instances. By utilising cgroups, the containers are able to share the available resources in a fair manner. The mounting of the host systems /sys/fs/cgroups directory onto the container is required by rkt due to the requirement of systemd-nspawn that a specific folder structure needs to be present. This issue arises due to some Linux distributions not including the systemd cgroup, or not mounting the cgroup controllers as required [46].

Table 2: Cgroup Usage Summary.

	Rkt	**Docker**	**LXD**
Cgroups utilised	Yes	Yes	Yes
/sys/fs/cgroups mounted?	Yes	No	No

This mounting of the host systems cgroups directory can lead to information disclosure of what services are running on the host and their memory usage. Docker and LXD both do not require the host systems cgroups directory to be mounted onto the container. Instead they use cgroups on the host system to do the limitations. Thus inside of the containers themselves there are no references to other containers, or host processes. The cgroups shows the running of a container through its init PID, and any child of that will reside inside of the same cgroup due to the hierarchical nature of cgroups.

11 Namespaces

In order to reduce the complexity of the container implementations, the kernel namespaces system is used to isolate various containers from each other and the host system (see Table 3). In order for unprivileged containers to be available to users, the user namespaces must be able to be utilised. However, as the users still need to map to users outside of the container a range of user IDs must be made available. On Linux, this is done by utilising subuid, and subgid ranges with the ranges mapping to the available range for the user namespaces of the container. These ranges should not overlap, otherwise, it is possible for the permissions granted to some of the users within a container to be transferred to users within another container. Additionally, the filesystem being used by the container runtime must be able to support the usage of UID ranges.

Table 3: Linux Namespaces Usage Summary.

	Rkt	Docker	LXD
Uses namespaces for isolation?	Yes	Yes	Yes
Unprivileged containers available[2]?	No	No	Yes
User namespaces supported?	Partial	Yes	Yes

The permissions for the filesystem can be challenging to implement as the containers that are using the filesystem will need to have the same subordinate UID/GID ranges (or at least overlapping). This is due to the owner of the filesystem will have the UID/GID within these ranges, with the effective UID being the mapped value for the container.

LXD utilises the LXCFS FUSE system in order to provide fully unprivileged containers to users. This increases the security considerably as an unprivileged container will be unable to give a process root access to a host system in the event of a breakout. The primary disadvantage of using a FUSE system is the overhead of the bridge between the userspace filesystem and the kernel operations.

While rkt does allow a user namespace to be used, this cannot be done when using the OverlayFS filesystem. The user namespaces are not supported for this filesystem implementation. Even if the user namespaces were supported for OverlayFS, the unprivileged containers would still not be available because rkt requires root access for other operations when launching a container instance, for example, networking setup. Like rkt, Docker does not support unprivileged containers because a privileged user is required for the container launching process. However, with the user namespaces a container boundary breakout does not give the user a root shell, instead their effective user ID is the ID in the subuid range that was mapped to the root user within the namespace.

12 Seccomp

Each of the container technologies utilises seccomp as the methodology of restricting access to the host systems kernel (see Table 4). Without this the contained processes could have unrestricted access to the kernel, posing a serious security risk to the system. Additionally, the seccomp filters are all setup by default thus ensuring that at least some of the risks are being mitigated on a default installation.

Table 4: Seccomp Filters Usage Summary.

	Rkt	Docker	LXD
seccomp utilised?	Yes	Yes	Yes
seccomp filters created by default?	Yes	Yes	Yes
Default filter is a whitelist?	No	Yes	No
Default filter is a blacklist?	Yes	No	Yes
Custom filters per container available?	No	Yes	No

In order to increase security, it is possible to implement a custom filter for each container, only allowing certain system calls to be performed by the container. This can be used to prevent any unexpected system calls from

[2] The unprivileged containers available property is referring to the ability of launching containers without access to the root user.

occurring. While this can be done within the content of the image by creating a seccomp filtered wrapper program, it is better to be able to also control the container in the case of an untrusted container image.

12.1 Rkt Usage of Seccomp

From investigation, the seccomp filters in use are a set of blacklisted system calls defined in systemd-nspawn. The list of blacklisted system calls consists of:

- iopl: change the I/O privilege level
- ioperm: set port input/output permissions
- kexec_load: load a new kernel for later execution
- swapon: start swapping to file/device
- swapoff: stop swapping to file/device
- open_by_handle_at: open file via a handle
- init_module: load a kernel module
- finit_module: load a kernel module from file
- delete_module: unload a kernel module
- syslog: read and/or clear kernel message ring buffer

It can be seen that these system calls are blocked to prevent people from modifying the kernel of the host system, changing permissions, and obtaining system message rings information. The open_by_handle_at system call would allow for a container to have access to any file on the host system. While custom filters are not available at the time of this study, the ability to have the filters is being introduced into the image specification. In order to work around this limitation at this point in time, we can use systemd. As rkt has full integration with init systems like system, we can utilise the SystemCallFilter option of systemd.exec [47]. However, this approach will work for entire pods and we may want to have this feature for single application, which is yet to be implemented.

12.2 Docker Usage of Seccomp

Compared with LXD and rkt, Docker utilises a whitelist for the default seccomp filter. Whilst being an extensive list of allowed calls, this does enable the containers to be able to be utilised for a variety of different applications with little knowledge of seccomp by a user. Whilst being used by default seccomp can be disabled at a per-container level using the –security-option seccomp:unconfined option when running the container. This is useful if the application requires a lot of system calls that are not allowed on the list.

A list of system calls that are not allowed under the default profile, and the reasoning is available at: https://github.com/docker/docker/blob/master/docs/security/seccomp.md

Additionally, Docker allows custom filters to be defined from json files, which can add an additional level of security to a container. These json files are able to specify a default action for system calls, and actions to take for particular system calls. This allows for custom whitelist or blacklist filters to be implemented for a container. They are utilised by a container using the –security-option seccomp:<filename>.json option. Allowing for only the system calls that are intended to be used by the application to be available.

12.3 LXD Usage of Seccomp

In a similar way to rkt, LXD utilises a blacklist filter for seccomp. The list of forbidden system calls in the default filter are as follows:

- umount –f (force umount)
- kexec_load
- open_by_handle_at
- init_module
- finit_module
- delete_module

This means that LXD is much more open to the usage of system calls when compared to rkt and Docker. This is understandable as LXD focuses on full operating system containers, rather than the application containers of rkt and Docker. i.e. LXD will require access to more of the host systems kernel to provide the required functionality.

13 Linux Kernel Capabilities

As the use cases of containers may require a root user, kernel capabilities are utilised to limit the abilities of the internal root user to the outside world. Without this a root user within a container would essentially have full access to the system. The CAP_SYS_ADMIN capability can be considered as essentially root. A process running with CAP_SYS_ADMIN is able to utilise many different administrator operations [37] including volume mounting, perform operations on trusted and security extended attributes, forge UID on sockets, and perform administrative operations on device drivers.

The CAP_DAC_READ_SEARCH allows for the process to bypass file read permission checks, and directory read and execute checks. This capability used to be granted to Docker containers until a bug was found which allowed the host systems filesystem to be walked in order to find any object on the host system that you wished, for example the password or shadow file [48].

Similar to the seccomp, having a whitelist of allowed capabilities is better for security as it is much easier to audit a list of known capabilities than a blacklist of dropped capabilities. This will also limit the capabilities to only those that are required by containers.

The ability to implement a list of capabilities to be dropped or retained will further increase the security if it is utilised correctly. This will allow each container to be given just the set of capabilities that it requires to run, limiting the attack surface considerably.

For the default stage-1 image of rkt, the default capabilities being utilised are the defaults for the systemd-nspawn. The following capabilities of systemd-nspawn instances are well known to have security risks [49]:

- CAP_SYS_ADMIN
- CAP_DAC_READ_SEARCH

From Table 5, we can see that Docker has the most restrictive capability sets of the containers by default. However, all of them are capable of providing any level of restriction into the capability sets for the containers using the custom capability granting features of the implementations.

Table 5: Linux Kernel Capability Usage Summary.

	Rkt	**Docker**	**LXD**
Kernel capabilities used?	Yes	Yes	Yes
CAP_SYS_ADMIN granted?	Yes	No	Yes
CAP_DAC_READ_SEARCH granted?	Yes	No	Yes
Whitelist capabilities?	No	Yes	No
Custom capability lists?	Yes	Yes	Yes

14 Container Linking Mechanisms

In this section, we will be discussing the various ways that containers can be linked together to form a system of different, related processes. As containers are heavily used in microservices based deployments the linking of different container instances is needed. LXD is concerned more with full system containers, they are not good for microservice deployments. However, they would be good for distributed system testing or IaaS deployments where the user needs the entire operating system.

Instead of LXD, the application container rkt and Docker would be far more suitable for microservice architectures when using the network linking mechanism. Rkt does allow for applications to share namespaces and thus be linked in that regard using pods. However, this has limited the usefulness and should only be used if two applications are heavily dependent on each other, or if one application is being used as a debug tool for another application in the development environment, e.g., a logger application.

14.1 Network Linking

The most common way of linking containers together would be to utilise some form of virtual networking. The popularity of this approach is probably associated with the ability to move beyond a single host machine for the

applications to a cluster environment for running containers. Additionally, this approach utilises a technology stack that most developers of distributed systems know well: the network.

14.1.1 Docker Approach to Network Linking

The Docker engine allows for user defined virtual networks to be created to associate different sets of containers together. This will allow containers to join the user defined networks as required providing a flexible deployment architecture. The user defined networks can be created utilising either a bridge network or an overlay network. A bridge network is similar to the default networking bridge docker0, except it can be further customised to include various options. Additionally, it is possible to add a filter to the bridge, which docker0 does not have. The overlay network driver provides Docker the ability to have container networks spanning multiple Docker engine hosts. Whilst there have been many approaches to building this outside of Docker, this is a Docker native solution to the problem.

Originally, multi-host networking was a difficult problem with Docker containers as they were designed to be small test environments residing on a single machine. Now the popularity of Docker has increased overtime leading to full deployments being done within Docker this feature was required. By default, the launching of a container will attach the container to the docker0 bridge leading to a security risk. It is highly advisable to utilise the user defined networking features in order to provide security for the Docker container networks. Especially if it is required that some containers should not talk to each other, as any container on the docker0 bridge will be able to communicate with any other container.

14.1.2 Rkt Approach to Network Linking

Similar to Docker, rkt allows for the creation of user defined networks [50] except instead of creating them through command line interfaces rkt container networks are defined in configuration files. The default network is that of a contained network namespace with a veth device that creates a point-to-point connection between a pod and a host. The default network will have a default route within a pod's namespace and a host will have IP masquerading enabled for the outgoing traffic from the pod.

Additionally, the rkt also has a default-restricted network setup on a new installation of rkt. This is the same as the default network except that the default route, and IP masquerading is not established. If a pod does not require any networking at all, then rkt comes with the option to disable a network completely by only having a single loopback interface within the pod. This means that a pod can access networked applications within the pod but not outside of it, and any other pod, or the host is unable to access the networked services within the pod.

If a pod needs to have full access to the network namespace of a host or a calling process, then rkt has the option to use the host network. While this gives simple setup for network deployments, this also poses serious risks as a pod will be able to access all of the networking stack of the calling process, which could be a host itself.

In order to perform port forwarding the image manifest for the application needs to have the ports that it will expose defined, and when run the option –port=<port-name>:<host-port> needs to be included.

As rkt can be fully controlled by system, it is possible to have a container application be socket activated [51]. That is systemd listens on a particular TCP socket for incoming connections and when they occur the rkt container is started in order to process the request. This can lead to a more secure system as containers will only be started when they are required to be, but there will be some request latency due to the start time.

14.1.3 LXD Approach to Network Linking

LXD does not do any automated management of the network, instead it leaves that work to users. In the default setup of LXD, they use a default bridge lxcbr0 that is not connected to any physical device [52]. The reasoning behind not connecting to a physical device is that lxcbr0 uses DHCP, which could cause networking issues with any other computer on the same network the device is connected to. This implies that by default there is no external access to the containers that is good for security but not for most usability.

15 Isolation Mechanism Improvements

In this section, we will summarise various isolation mechanisms provided by different container technologies. We also suggest some improvements that can be made to either within a particular technology or on a host system to build a secure container environment by utilising a particular container technology.

15.1 Rkt Isolation

While being designed with security in mind, the level of isolation between rkt and a host system as well as other containers still require some work. This is primarily due to the immaturity of the rkt project (only reaching v1.0 in February 2016) but this needs to be addressed before it can be utilised in a production environment.

In rkt there are two main areas for concern: firstly, the mounting of the host operating systems /sys/fs directory tree, and the usage of chroot to contain the hosted applications. In the mounting of the hosts /sys/fs directory tree, the major issue is that data can be read which can give information about other containers and the host system in terms of running services, and memory usage of the services.

One method of increasing the isolation in terms of the /sys/fs directory tree is to create a virtualised structure of the filesystem so that the applications can run, but are unable to see parts of the host systems directory tree. This approach is done by LXD, with the LXCFS virtualised cgroup.

In regards to the chroot usage, the main issue is that chroot jails are not secure and do not guarantee that the deployed application will stay in stage 2. Instead the application may move back into stage 1 giving access to data that an application should not have.

Within chroot there are several areas of concern with regards to improving the isolation levels.

1. Chroot is difficult to secure;
2. The linux capabilities isolators are not fully implemented;
3. Where they are implemented they sit at a pod level instead of an app level as required by the specification [53];
4. Modifications require large architectural shift from using an executable runner, to a purely systemd implementation.

Not only is one way of breaking out of a chroot jail documented in the projects man pages, but there are a multitude of ways to circumvent various protections against the nested chroot attack. Additionally to the chroot based breakout if it is possible to create devices within the jail, then creating a block device, or the /dev/kmem device can also be used to breakout of the jail [54].

The difficulty in securing chroot is that some methods are specific to the filesystem format that is being used (i.e. OverlayFS for rkt), e.g., using chattr to make the filesystem unable to create new files. Another possible method of protecting the system is to limit the contents of the jail as much as possible. Unless the administrator prevents the usage of unaudited images, however, this will not work as the attacker will be able to launch an image with what they require within it.

Whilst investigating a possible method of improving the isolation by using the Linux capabilities isolators of appc, it was found that these isolators have not been fully implemented in rkt. Of the two possible options for either whitelisting or blacklisting the Linux capabilities only one had any effect on the system, which is the whitelisting option. However, if this is used, it will only add new capabilities to the capability set of the default systemd-nspawn container, there will be no action taken on an application. This means that even though some action is taken, it does not perform the action required by the specification.

As mentioned above, the implementation of the capabilities as they currently sit at the pod level, instead of the app level. This is due to all logic pertaining to Linux capabilities are involved in the creation of the systemd-nspawn container. While modifying the rkt runtime in order to support the isolators being used, it was found that the modifications required were non-trivial due to several factors. For any application to launch the CAP_SYS_CHROOT capability needs to be granted within the systemd-nspawn container; however from a naïve change to add the isolators with the systemd.exec CapabilityBoundingSet option [47] with no other changes, we found that no application could launch without the capability.

This was due to the fact that instead of using systemd unit files to perform the actions, it utilises an appexec program that creates the chroot jail and runs the service with other setup taking place. This makes it impossible to remove the CAP_SYS_CHROOT capability from the applications without breaking the image. Our solution to this problem was to create a new stage1 image implementation, which utilised systemd to increase the isolation mechanisms being used. In regards to the rkt run command, our solution was to move most of the logic from the appexec program that was used to launch the applications into the systemd unit file for the application.

This allowed for systemd to perform the chroot outside of the executable enabling us to utilise the kernel capability bounding set onto the application executable that is being launched. Our solution was designed as a proof of concept on how to modify rkt in order to increase the security of the containers. As a proof of concept, it succeeds in restricting the kernel capabilities of the applications being launched so that a chroot breakout will not occur.

However, it does not allow the rkt enter command to be executed currently. If the default implementation of rkt enter is utilised, then the enter command could bypass our implemented protections. In order to secure against this issue, our own version of enter needs to be developed. We made some attempts at developing this, however, our attempts were met with challenges yet to be overcome; such as

- Unable to clone the mount namespace in Go due to multi-threaded nature;
- Go has bindings to perform actions with systemd; and
- Systemd can be used to find the capabilities of the application.

Our current incomplete, solution consists of two binaries: enter_systemd and appexec_systemd. The enter_systemd program was written in C and performs actions on entering the pod of the application through namespace cloning and chrooting into the root of the pod. The appexec_systemd program was written in Go and performs actions relating to the application capabilities and running the requested binary. Although our solution is incomplete, it can be considered a usable proof of concept as the enter command should not be useful if the application container image was setup correctly with the minimal number of applications included.

15.2 Docker Isolation

Docker was not developed by following the security by design principles from inception; however Docker has had a lot of security measures implemented since. Its usage of whitelists in regards to various filters and capabilities give it a higher likelihood of being secure, as blacklists will usually miss something. The ability to create custom seccomp filters at a per container level is also a big advantage in terms of security. However, like all similar systems, it does have its weaknesses in terms of security. These weaknesses include the default networking model, MAC integration, and operations are being run as root.

The default networking model of Docker has been a known issue for quite some time [55]. The major issue is the unfiltered bridge that sits on a host OS. This allows for any Docker container started on the system to communicate with any other. Additionally, the bridge is attached to a physical device allowing for the containers to connect to the outside world by default.

However, as the Docker bridge is assigned an address in the private address space, it is not simple to gain access to the containers from outside of a host. Whilst this attaching to the external network is useful for starting up easily, it does give the containers opportunity to pull or push data to the internet by using the entrypoint command. In order to improve the networking model for isolation, it is recommended to either remove the bridge and have the containers un-networked by default, or by detaching this bridge from the physical network interface on the host so that external communication cannot occur. Whilst the default of being un-networked is the most secure model, it adds serious overhead to getting the networking up and running when it is required. Detaching the bridge from the physical interface still allows containers to communicate unfiltered (this can be remedied fairly easily), but does not allow communication to the outside world.

Whilst it is true that there are other forms of networking available for the containers including a type which allows for multi-host container networking allowing clusters to be setup with just Docker tools. There is an advantage of tightening the security of the default use case, as in most cases this will be the one that is used. The MAC integration of Docker leaves a lot to be desired. Whilst it is true that on AppArmor systems, there is a default profile available, there is no such profile for the SELinux systems.

AppArmor tends to be the default MAC implementation used on Debian based hosts, whereas SELinux is primarily found on RedHat based distributions. Whilst both provide mandatory access controls, AppArmor tends to be less secure as it utilises a path based system, rather than the more complex and secure type based system of SELinux. Besides these differences the fact that Docker is only protected on AppArmor systems, with SELinux systems having protections using the default profile for the OS which does not usually take container technologies into account[3].

[3] This may change due to the increased interest in containers

All of the operations of Docker are privileged due to the fact that all operations go through the Docker daemon. This daemon runs with root-like privileges and is essentially an API sitting on a host, which informs the engine what to do, while the Docker client sends REST requests to the daemon. The rkt runtime does not have many operations being unprivileged, the image fetching, verification, and validation are good start. There have been a number of issues being raised about the image security of Docker [6], [7].

By having these run as a privileged user, any vulnerability can lead to a high risk privilege escalation or remote code execution occurring. Unfortunately, this is deeply ingrained into the Docker's architecture. It is difficult to get any proposed solution approved or implemented due to the monolithic nature of the Docker daemon/engine. One proposed solution can be to split the daemon/engine into parts with an image component that runs in an unprivileged user space.

15.3 LXD Isolation

Our study has determined a few problems with regards to the isolation mechanisms of LXD. The identified issues are related to future maintenance of the security of a system. These refer primarily to the usage of blacklists for the seccomp filters and Linux kernel capability list. One area of potential concern is that networking is done manually by a system's administrator. However, the LXD developers do not see the external networking as a concern of the containers system [52] as that will narrow the options of how to approach networking. However, it does set up a lxcbr0 though this is not attached to any physical device so external network connectivity is disabled by default unlike in Docker.

The restriction of utilising AppArmor as the only supported MAC implementation can lead to issues when the policy of a user is to utilise SELinux as the MAC implementation. However, this is an administrative issue rather than a security issue. The ability to create unprivileged containers as the default option provides a very powerful security isolation feature as if a container breakout was to occur then a privilege escalation attack does not occur at the same time. If an attacker within a container attempts to obtain root access to a host system, the attacker needs to breakout of the container, and find a privileged escalation on the user running the containers. This is in contrast to a privileged container where the breakout process will have the same privileges as the container, which could be root (or at least partial root).

We have discussed the issues with blacklist filters in previous sections, and the issues with them here are the same as before. However, in both of the cases, the filters being applied are different when compared to rkt which also used blacklist filters. In terms of the seccomp filter, LXD has a more permissive filter applied with IO syscalls, and syslog able to be performed. However, this can be directly attributed to the variation in use cases, i.e., the use case of Docker/rkt is that of an application container, whereas the use case of LXD is that of a system container. This means that direct comparisons between Docker/rkt and LXD can only be made at the base level, whereas Docker and rkt can be directly compared as they are in direct competition with each other. For an appropriate comparison, LXD can be compared with a type-2 hypervisor such as KVM [56].

15.4 General Observations

Whilst each of the containers are isolated from the viewpoint of container to the host/other containers with varying levels of success, there are some issues that can affect the runtimes of all sorts of containers. If we instead look at the isolation of the containers from the host perspective, i.e., a root user on the system, then we can see there is a lot of room for improvement. The major issue is that from the host perspective, it is simple to see what processes are running on a particular container through the systemd init system. By running systemctl status, we can see each of the containers and what the PID of each of these are in the global PID namespace. This can allow someone with CAP_SYS_KILL to stop any application running within a container using kill -9 <PID> (send SIGKILL to process with PID=<PID>)

This leads to the conclusion that container technologies should not run on untrusted host machines. If processes are required to be run on untrusted systems, a full type-2 hypervisor such as KVM should be used instead. However, rkt allows custom levels of isolation through different stage1 images where the most secure image is the KVM stage1 image. This does have a much higher memory overhead than the default image, even when using the LKVM implementation but on hosts that are not trusted by the user then this may be the solution to use the rkt image format to create a KVM instance.

16 Concluding Summary

This document reports our work aimed at analysing the isolation mechanisms used by container technologies for securing the contents of the host operating system and other container instances. We have studied the isolation mechanisms applied by three container technologies: Docker, LXD, and Rkt. While the isolation is an important component for securing virtualisation technologies, it is only one aspect of the security for containerised technologies. In order to obtain a complete security analysis of the technologies, it is important to analyse the other security characteristics. It is worth mentioning that whilst Rkt and Docker share the same primary use-case of application deployments, LXD's primary use-case is for the deployment of system containers.

Based on the work carried out for this research project, we are able to make some conclusions that are expected to be useful for practitioners and researchers interested in securing container technologies. We have found that each of the analysed container technologies has different strengths and weaknesses with respect to the isolation mechanisms implemented and the workload to be placed on the selected container technology.

The isolation mechanisms of LXD are able to provide a high degree of isolation between different container instances as well as from the host. The main issue of LXD is that whenever the Linux Kernel adds new system calls, LXD may need an update to the blacklist to ensure that the newly added system calls are not a security risk.

Rkt is an application container technology, which does not require any daemon to be running at all times on a host system. The isolation mechanisms used by rkt are dependent on the stage 1 image when running the container image. Rkt provides three images for stage 1 with different isolation levels. The rkt fly image is the least secure with most mechanisms disabled and should only be used if required for a particularly use case. The default CoreOS image uses systemd-nspawn as the container technology; it provides reasonable isolation but needs improvement with regards to the capabilities being too permissive. The KVM image is used to have a full KVM virtual machine being created for use with the container technology. Due to the immaturity of the rkt project, however, we are unable to make concrete conclusions about the technology as it is changing frequently.

Docker is a container technology focused on the deployment of applications. It is the most mature of the technologies that we have been looking into for this research project, however, it appears not have been designed with security in mind. The isolation of the containers is still a work in progress with the addition of custom seccomp filters bing added, and the utilisation of whitelists for most filters implying that the changes in the Linux Capabilities, and system calls will not decrease the security of Docker in most cases. The major security risk of the Docker containers is that all of the operations go through the Docker Engine, which must be running as a root user. Aside from this issue, the default networking of the Docker containers is considered insecure due to the unfiltered bridge that is utilised; however, Docker does support custom networking setup that means enabling system administers to deploy secure networks.

Acknowledgement

This research was performed under contract to the Defence Science and Technology (DST) Group Maritime Division, Australia.

Bibliography

[1] "Why Docker is Not Yet Succeeding Widely in Production." [Online]. Available: http://sirupsen.com/production-docker. [Accessed: 17-Mar-2016].

[2] U. Gupta, "Comparison between security majors in virtual machine and linux containers," *ArXiv150707816 Cs*, Jul. 2015.

[3] T. Bui, "Analysis of Docker Security," *ArXiv Prepr. ArXiv150102967*, 2015.

[4] K. Agarwal, B. Jain, and D. E. Porter, "Containing the Hype," in *Proceedings of the 6th Asia-Pacific Workshop on Systems*, New York, NY, USA, 2015, pp. 8:1–8:9.

[5] M. Hayden, "Securing Linux Containers."

[6] "Docker Image Insecurity." [Online]. Available: https://titanous.com/posts/docker-insecurity. [Accessed: 08-Sep-2015].

[7] "Docker image 'verification' [LWN.net]." [Online]. Available: https://lwn.net/Articles/628343/. [Accessed: 08-Sep-2015].

[8] "Announcing Docker 1.8: Content Trust, Toolbox, and Updates to Registry and Orchestration," *Docker Blog*. [Online]. Available: http://blog.docker.com/2015/08/docker-1-8-content-trust-toolbox-registry-orchestration/. [Accessed: 15-Sep-2015].

[9] "Introducing Docker Content Trust," *Docker Blog*. [Online]. Available: https://blog.docker.com/2015/08/content-trust-docker-1-8/. [Accessed: 22-Oct-2015].

[10] "3ofcoins/jetpack," *GitHub*. [Online]. Available: https://github.com/3ofcoins/jetpack. [Accessed: 27-Jan-2016].

[11] "apcera/kurma," *GitHub*. [Online]. Available: https://github.com/apcera/kurma. [Accessed: 27-Jan-2016].

[12] "kernel/git/torvalds/linux.git - Linux kernel source tree: OverlayFS Documentation." [Online]. Available: https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/filesystems/overlayfs.txt. [Accessed: 07-Mar-2016].

[13] "CoreOS." [Online]. Available: https://coreos.com/rkt/docs/latest/running-fly-stage1.html. [Accessed: 10-Feb-2016].

[14] "systemd.resource-control." [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemd.resource-control.html. [Accessed: 11-Feb-2016].

[15] "CoreOS." [Online]. Available: https://coreos.com/rkt/docs/latest/using-rkt-with-systemd.html#advanced-unit-file. [Accessed: 11-Feb-2016].

[16] "chroot(2): change root directory - Linux man page." [Online]. Available: http://linux.die.net/man/2/chroot. [Accessed: 16-Feb-2016].

[17] "Clear Linux Project Wraps Containers in Speedy VMs - Intel Software and Services." [Online]. Available: https://blogs.intel.com/evangelists/2015/05/19/clear-linux/. [Accessed: 24-Mar-2016].

[18] "app-level seccomp isolator, whitelist of syscalls · Issue #1614 · coreos/rkt · GitHub," *GitHub*. [Online]. Available: https://github.com/coreos/rkt/issues/1614. [Accessed: 24-Feb-2016].

[19] "spec: add seccomp isolator for Linux · Issue #529 · appc/spec · GitHub," *GitHub*. [Online]. Available: https://github.com/appc/spec/issues/529. [Accessed: 24-Feb-2016].

[20] "AUFS storage driver in practice." [Online]. Available: https://docs.docker.com/engine/userguide/storagedriver/aufs-driver/. [Accessed: 08-Mar-2016].

[21] T. Jay, "Before you initiate a 'docker pull,'" *Red Hat Security*. .

[22] "Linux Containers - LXD - Introduction." [Online]. Available: https://linuxcontainers.org/lxd/introduction/. [Accessed: 22-Feb-2016].

[23] "aufs." [Online]. Available: http://aufs.sourceforge.net/. [Accessed: 07-Mar-2016].

[24] "Linux Containers - LXCFS - Introduction." [Online]. Available: https://linuxcontainers.org/fr/lxcfs/introduction/. [Accessed: 07-Mar-2016].

[25] "GitHub - lxc/lxcfs: FUSE filesystem for LXC." [Online]. Available: https://github.com/lxc/lxcfs. [Accessed: 07-Mar-2016].

[26] "LXD image handling." [Online]. Available: https://raw.githubusercontent.com/lxc/lxd/master/specs/image-handling.md. [Accessed: 22-Feb-2016].

[27] "Mandatory Access Control." [Online]. Available: http://www.cgisecurity.com/owasp/html/ch08s02.html. [Accessed: 18-Feb-2016].

[28] "SELinux Wiki." [Online]. Available: http://selinuxproject.org/page/Main_Page. [Accessed: 19-Jan-2016].

[29] "AppArmor." [Online]. Available: http://wiki.apparmor.net/index.php/Main_Page. [Accessed: 19-Jan-2016].

[30] "AppArmor." [Online]. Available: https://help.ubuntu.com/stable/serverguide/apparmor.html. [Accessed: 18-Feb-2016].

[31] Paul Menage, Paul Jackson, and Christoph Lameter, "cgroups." [Online]. Available: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. [Accessed: 18-Feb-2016].

[32] “Docker security.” [Online]. Available: https://docs.docker.com/articles/security/. [Accessed: 08-Sep-2015].
[33] “namespaces(7) - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man7/namespaces.7.html. [Accessed: 08-Sep-2015].
[34] “pid_namespaces(7) - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man7/pid_namespaces.7.html. [Accessed: 08-Sep-2015].
[35] “user_namespaces(7) - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man7/user_namespaces.7.html. [Accessed: 18-Feb-2016].
[36] “overview - seccompsandbox - Mode of operation for seccomp sandbox - Google Seccomp Sandbox for Linux - Google Project Hosting.” [Online]. Available: https://code.google.com/p/seccompsandbox/wiki/overview. [Accessed: 18-Feb-2016].
[37] “capabilities(7): overview of capabilities - Linux man page.” [Online]. Available: http://linux.die.net/man/7/capabilities. [Accessed: 01-Dec-2015].
[38] “gethostname(2) - Linux manual page.” [Online]. Available: http://man7.org/linux/man-pages/man2/gethostname.2.html. [Accessed: 21-Feb-2016].
[39] “Docker contrib/apparmor/template.go.” [Online]. Available: https://raw.githubusercontent.com/docker/docker/master/contrib/apparmor/template.go. [Accessed: 23-Feb-2016].
[40] “SVirt - SELinux Wiki.” [Online]. Available: http://www.selinuxproject.org/page/SVirt. [Accessed: 11-Feb-2016].
[41] “Security context - FedoraProject.” [Online]. Available: https://fedoraproject.org/wiki/Security_context. [Accessed: 23-Feb-2016].
[42] “sesearch(1): SELinux policy query tool - Linux man page.” [Online]. Available: http://linux.die.net/man/1/sesearch. [Accessed: 23-Feb-2016].
[43] CoreOS, “Lifecycle of a pod in rkt.” [Online]. Available: https://coreos.com/rkt/docs/latest/devel/pod-lifecycle.html. [Accessed: 23-Feb-2016].
[44] “Linux Containers - LXC - Manpages - lxc.container.conf.5.” [Online]. Available: https://linuxcontainers.org/lxc/manpages/man5/lxc.container.conf.5.html. [Accessed: 31-May-2016].
[45] “lxc/lxd: scripts/lxc-to-lxd,” *GitHub*. [Online]. Available: https://github.com/lxc/lxd/blob/b71b6e039f147c5db78b28ac6e45ab876e88d940/scripts/lxc-to-lxd. [Accessed: 31-May-2016].
[46] “rkt/init.go at master · coreos/rkt · GitHub.” [Online]. Available: https://github.com/coreos/rkt/blob/master/stage1/init/init.go. [Accessed: 24-Feb-2016].
[47] “systemd.exec.” [Online]. Available: https://www.freedesktop.org/software/systemd/man/systemd.exec.html. [Accessed: 24-Feb-2016].
[48] “Docker Container Breakout Proof-of-Concept Exploit,” *Docker Blog*, 18-Jun-2014. [Online]. Available: https://blog.docker.com/2014/06/docker-container-breakout-proof-of-concept-exploit/. [Accessed: 15-Feb-2016].
[49] “stage1: rkt pods should not be given CAP_SYS_ADMIN · Issue #576 · coreos/rkt · GitHub.” [Online]. Available: https://github.com/coreos/rkt/issues/576. [Accessed: 15-Feb-2016].
[50] “rkt/networking.md at master · coreos/rkt · GitHub.” [Online]. Available: https://github.com/coreos/rkt/blob/master/Documentation/networking.md. [Accessed: 24-Feb-2016].
[51] “rkt/using-rkt-with-systemd.md at master · coreos/rkt · GitHub.” [Online]. Available: https://github.com/coreos/rkt/blob/master/Documentation/using-rkt-with-systemd.md#socket-activated-service. [Accessed: 24-Feb-2016].
[52] “Network Setup Is Too Hard · Issue #1294 · lxc/lxd · GitHub.” [Online]. Available: https://github.com/lxc/lxd/issues/1294. [Accessed: 24-Feb-2016].
[53] “App Container Executor Spec: spec/ace.md.” [Online]. Available: https://github.com/appc/spec/blob/master/spec/ace.md. [Accessed: 24-Feb-2016].
[54] “Unix and Linux Security: An Introduction - Breaking Out of and Securing Chroot Jails.” [Online]. Available: http://talby.rcs.manchester.ac.uk/~isd/_unix_security/unix_security_intro_securing_network_services.Breaking_Out_of_and_Securing_Chroot_Jails.html. [Accessed: 01-Mar-2016].
[55] J. Nickoloff, “Safer Local Docker Networks: TL;DR Prevent arbitrary inter-container communication by setting icc=false and using container linking.,” *Medium*, 05-Jan-2015. [Online]. Available: https://medium.com/on-docker/safer-local-docker-networks-8ce22127f9df. [Accessed: 08-Sep-2015].
[56] “LXD crushes KVM in density and speed | Ubuntu Insights.” [Online]. Available: https://insights.ubuntu.com/2015/05/18/lxd-crushes-kvm-in-density-and-speed/. [Accessed: 08-Mar-2016].

Appendix A: Glossary

KVM: Kernel-based virtual machine, is a full virtualisation tool that is classified as a type-2 hypervisor. In order to utilise KVM the CPU it is running on must be x86 with virtualisation extensions (Intel VT or AMD-V)

VM: Virtual Machine, is an instance of virtualised hardware with an operating system kernel running on top of it to provide an operating system as if it is running on hardware. The lifecycle of a virtual machine is controlled by hypervisors.

Hypervisor: An application which manages the creation and lifecycle of virtual machines.

Type-1/2 Hypervisor: There are two different types of hypervisors available for usage. The type-1 hypervisor is known as a native, or bare-metal hypervisor, these run directly on a host machines hardware. The type-2 hypervisor is known as a hosted hypervisor as they run on top of a host operating system.

Containers: A process which performs operating system level virtualisation to host either applications or operating systems while sharing the host operating systems kernel. These are much lighter weight than a VM.

Application Container: This type of container is where the contents of the container consist of the bare minimum in order to run the application in the container instance. These are commonly used in micro-service oriented deployments.

System Container: This type of container is where the contents of the container consist of an entire operating system.

Union Filesystem: A type of filesystem where the files are consisting of various paths which are then merged together into a union of these paths.

Whitelist: In regards to security policy and access control lists, a whitelist is a list of allowed resources, or actions with any resources that are not mentioned in the policy being forbidden.

Blacklist: A blacklist is the opposite of a whitelist, in that a blacklist is a list of forbidden resources, or actions with any of these that are not mentioned are implicitly allowed.

DAC: Discretionary Access Control, is a type of access control which can be circumvented by a user with sufficient privileges. E.g. the UNIX filesystem uses DAC for its permission model.

Rule-Based Access Control: Access to various system resources are restricted based on a set of rules set by the systems administrator.

Role-Based Access Control: Access to various system resources are restricted based on the role of a user within the organisation. Each user can only be assigned one role within an organisation.

ACI: Application Container Image, this is the image format utilised by rkt as defined in the appc specification.

Appc: Application Container, this is the specification which rkt implements.

UUID: Universally Unique Identifier, is a standard used in software to uniquely identify different resources.

Systemd: Is an init system (like upstart) which is used to bootstrap the user environment on Linux.

DevOps: Developer Operations, is a process to open communications between the developers and IT administrators to allow for the automated deployment of infrastructure and services.

Pongo2: A Django-like template engine for the Go programming language.

YAML: YAML Ain't Markup Language, is a language for human readable data serialisation. This is used for the ability to create a file which is easily understood by both humans and computers.

REST: Representational State Transfer, is an architectural style with coordinated set of constraints applied to resources within a distributed system