The attached DRAFT document (provided here for historical purposes) has been superseded by the following publication:

Publication Number: **NIST Internal Report (NISTIR) 8176**

Title: *Security Assurance Requirements for Linux Application Container Deployments*

Publication Date: **October 2017**

- Final Publication: https://doi.org/10.6028/NIST.IR.8176 (direct link: http://nvlpubs.nist.gov/nistpubs/ir/2017/NIST.IR.8176.pdf).
- Information on other NIST Computer Security Division publications and programs can be found at: https://csrc.nist.gov/

The following information was posted with the attached DRAFT document:

August 1, 2017

Comments Due: August 25, 2017
Email Comments to: NISTIR8176@nist.gov

Author(s)

Ramaswamy Chandramouli (NIST)

Announcement

NIST requests comments on the release of **Draft NISTIR 8176, *Security Assurance Requirements for Linux Application Container Deployments***.

Application Containers are slowly finding adoption in enterprise IT infrastructures. To address security concerns associated with deployment of application container platforms, NIST Special Publication 800-190 (2nd Draft), *Application Container Security Guide*, identified security threats to the components of the platform hosting the containers and related artifacts involved in building, storing and using container images. It has also proposed countermeasures for the following components: Hardware, Host OS, Container Runtime, Image, Registry and Orchestrator.

To implement the countermeasures one or more security solutions are needed. To assess the effectiveness of the security solutions implemented based on these recommendations, it is necessary to analyze them and outline the security assurance requirements they must satisfy to meet their intended objectives. This is the contribution of Draft NISTIR 8176. The focus is on application containers on Linux platforms.

The security solutions for which security assurance requirements have been derived cover the following areas:

1. Hardware-based root of trust providing integrity for boot process,
2. Configuration options using host OS kernel features and kernel loadable modules,
3. Protection measures for building and storing container images, and
4. Configuration options in Orchestrator tools used for rolling out a production infrastructure involving multiple containers and multiple hosts.

Draft NISTIR 8176

Security Assurance Requirements for Linux Application Container Deployments

Ramaswamy Chandramouli

NIST
National Institute of Standards and Technology
U.S. Department of Commerce

Draft NISTIR 8176

Security Assurance Requirements for Linux Application Container Deployments

Ramaswamy Chandramouli
Computer Security Division
Information Technology Laboratory

August 2017

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Kent Rochford, Acting NIST Director and Under Secretary of Commerce for Standards and Technology

National Institute of Standards and Technology Internal Report 8176
36 pages (August 2017)

Certain commercial entities, equipment, or materials may be identified in this document in order to describe an experimental procedure or concept adequately. Such identification is not intended to imply recommendation or endorsement by NIST, nor is it intended to imply that the entities, materials, or equipment are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including concepts and methodologies, may be used by federal agencies even before the completion of such companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at http://csrc.nist.gov/publications.

Public comment period: *August 01, 2017* through *August 25, 2017*

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: NISTIR8176@nist.gov

All comments are subject to release under the Freedom of Information Act (FOIA).

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof-of-concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems.

Abstract

Application Containers are slowly finding adoption in enterprise IT infrastructures. Security guidelines and countermeasures have been proposed to address security concerns associated with the deployment of application container platforms. To assess the effectiveness of the security solutions implemented based on these recommendations, it is necessary to analyze them and outline the security assurance requirements they must satisfy to meet their intended objectives. This is the contribution of this document. The focus is on application containers on a Linux platform.

Keywords

application container; capabilities; Cgroups; container image; container registry; kernel loadable module; Linux kernel; namespace; Trusted Platform Module.

Acknowledgements

Audience

The target audience for this document includes system architects and system administrators for container stacks in enterprise infrastructures or in infrastructures used for offering container service as part of an overall cloud service.

Trademark Information

All registered trademarks or trademarks belong to their respective organizations.

Executive Summary

Application containers are now slowly finding adoption in production environments due to the following advantages: short development and deployment cycle, resource efficiency through lightweight virtualization, and availability of tools for automating the processes involved. At the same time, security concerns are dictating the pace of adoption. To address these concerns, security guidelines and countermeasures have been proposed by NIST through the Application Container Security Guide (NIST Special Publication 800-190).

The Application Security Guide identified security threats to the components of the platform hosting the containers and related artifacts involved in building containers and storing them prior to launch. Taking into consideration the overall security implications for the entire ecosystem involving containers, the document also provided security countermeasures for and through six entities including Hardware, Host OS, Container Runtime, Image, Registry and Orchestrator.

To carry out these recommendations in the form of countermeasures, one or more security solution are needed. In order for these security solutions to effectively meet their security objectives, it is necessary to analyze those security solutions and detail the metrics they must satisfy in the form of security assurance requirements. This is the objective and contribution of this document.

Linux and its various distributions form the predominant host OS component of the deployed container platforms. Since they are open-source products, sufficient security related information is available to analyze the security solutions that can be configured using features provided by Linux. Hence the focus of this document is on security assurance requirements for security solutions for application containers hosted on Linux. The target audience includes system security architects and administrators who are responsible for the actual design and deployment of security solutions in enterprise infrastructures hosting containerized hosts.

Table of Contents

List of Appendices

List of Figures

List of Tables

1 Introduction

Application containers are now slowly finding adoption in production environments due to the following advantages: short development and deployment cycle, resource efficiency through lightweight virtualization, and availability of tools for automating the processes involved. To address the security concerns in these environments, the Application Container Security Guide (NIST Special Publication 800-190) [1] (referred to in the rest of this document as *Container Security Guide*) identified security threats to the components of the platform hosting the containers as well as related artifacts involved in building containers and storing them prior to launch. Taking into consideration the overall security implications for the entire ecosystem involving containers, the document also provided security countermeasures for and through six entities including Hardware, Host OS, Container Runtime, Image, Registry and Orchestrator.

To implement these countermeasures, one or more security solutions are needed. This document discusses potential security solutions that provide the functionality necessary in countermeasures and the kind of security assurance requirements each should satisfy. These security solutions can be broadly classified as:

(a) Hardware-based root of trust providing integrity for boot process

(b) Configuration options using host OS kernel features and kernel loadable modules

(c) Protection measures for building and storing container images

(d) Configuration options in Orchestrator tools used for rolling out a production infrastructure that involves multiple containers and multiple hosts

The purpose of this document is to examine each of the security solutions in the context of the security objectives they are designed to meet and to develop assurance requirements that they should satisfy in order to be effective. The host OS considered is Linux due to the following:

(a) Ubiquitous adoption in container stacks

(b) Linux distributions are open-source and allow for sufficient security related information to be made publicly available

1.1 Scope of the Document

The functional architecture diagram of a container technology stack is shown in figure 1. In this diagram, the stack is comprised of the Physical Host (or VM), Container OS (which we will refer to as Host OS in this document), Container Runtime, and the multiple containers. Additionally, tasks such as creating a virtual network linking containers within and across container hosts (Container Networking), creating clusters of container hosts (Container Cluster Management), creating pathway programs to identify and discover a specific container providing a particular service (Service Discovery), scheduling of containers across a cluster (Container Scheduling), and scheduling of specific business applications within various containers (Application Scheduling) that are all performed by multiple tools are incorporated under the umbrella of an Orchestrator software. Before actually launching them as containers on various container hosts, templates of components that constitute a container called Container Image are created using various DevOps Tools. These container images are stored in a container registry (Image Management) and are then pulled into container hosts and launched as containers using Container Runtime tools. The container runtime also provides the interfaces for configuring host OS parameters and settings associated with kernel-loadable modules to enable secure deployment of various containers.

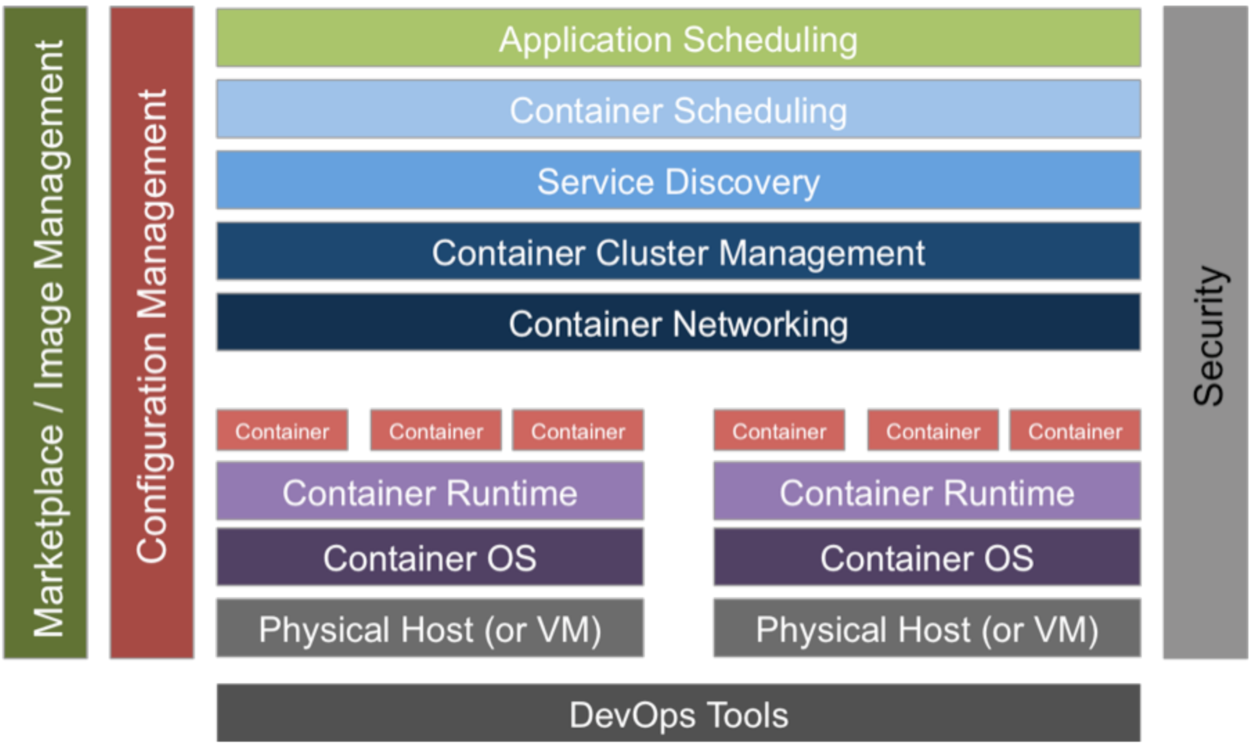

Figure 1 – Container Technology Stack

- As depicted in figure 1, the security functional layer spans all functional layers of the container technology stack. The security solutions covering these layers, however, must be implemented through the following components:
  - (a) Physical Host (i.e., hardware, since container hosting on VMs is out of scope for this document)
  - (b) Container OS (Host OS) interfaces
  - (c) Container Runtime interfaces
  - (d) Image Management and Registry Interfaces
  - (e) Orchestrator Interfaces

The containers running in the container stack can either be system containers or application containers. A container that behaves like a full OS and runs programs such as sshd (secure session establishment) and syslogd (logging capability) is called a system container, while one that runs only an application is called an application container [2]. This document focuses on application containers. Before analyzing the security solutions and identifying the assurance requirements they should satisfy, it is necessary to state the execution model of the application containers and the assumed attack model. First, the application is run within a container as a single operating system process. The container has a copy of the application code itself as well as the software stack (consisting of binaries and libraries) [3]. In most cases, this stack can be assembled using some type of library system, avoiding the need for the developer to build and configure the stack from scratch. These quickly assembled stacks are given different names in different container product offerings (e.g., buildpacks, cartridges, etc.). There are stacks for many of the popular programming language runtimes such as Java, PHP, Node.js, and Ruby. For specialized applications, developers can create their own customized stack. The deployment model in a container architecture may involve running copies of the same application in parallel with separate containers, even those spread across different container hosts. In this scenario, the infrastructure may have a mechanism to distribute incoming requests across all instances of the same application using some form of load balancer.

The attack model assumed here is that the vulnerability in the application code of the container or its faulty configuration (e.g., the container is configured to run in privileged mode) has been exploited by an attacker to take control of and compromise the privilege code in container runtime and host OS kernel where the latter is trusted by the application code in the container to provide some protection guarantees such as process isolation [4]. An example of such an attack is the replaying, recording, modifying, and dropping of a network packet or a file system access. The security solutions discussed in this document are intended to protect the container runtime and host OS against these types of attacks. Solutions to address the inherent insecure characteristics of the application code itself, such as programming bugs, design flaws or execution models, are beyond the scope of this document.

1.2 Document Structure

The remainder of this document is organized into the following sections and appendices:

- Section 2 provides an overview of the functions of various Linux kernel features (Namespaces, Cgroups, Capabilities) and kernel loadable modules in providing security for the containerized stack;
- Section 3 discusses hardware-based security solutions for container environments;
- Section 4 outlines host OS protection measures and their associated assurance requirements;
- Section 5 presents, in detail, several container runtime configuration solutions that guarantee container isolation for artifacts such as processes, filesystems, IPC, and networks. It also presents solutions for limiting resources and ensuring least privilege. All solutions are analyzed, and a set of assurance requirements that must be satisfied are presented;
- Section 6 defines assurance requirements for building and maintaining container images;
- Section 7 briefly discusses assurance requirements for container registry protection;
- Section 8 outlines basic security assurance requirements for Orchestration tools;
- Section 9 identifies some undesirable side effects of some security solutions and the need to exercise caution in the use of such solutions;

- Section 10 summarizes the various security solution areas that were covered in the document;
- Appendix A provides the definition for acronyms used in the document; and
- Appendix B contains a list of references.

2 Security Solutions for Linux Application Container Stack

In section 1.1, the host OS (in this context, Linux) interfaces were listed as mechanisms for implementing security solutions for a container stack. There are two types of interfaces: Linux kernel interfaces and Kernel Loadable Module (or Linux Security Module or LSM) interfaces. The Linux kernel features associated with the former type of interfaces are: Namespaces, Cgroups, and Capabilities. Out of these, the Namespaces and Cgroups kernel features provide isolation of processes running on top of the host OS and can be the driving features for development of the concept of containers. The salient functions of Linux kernel features and kernel-loadable module features are briefly described in the following sections to provide context for the security configurations and solutions analyzed in the subsequent sections.

2.1 Linux Kernel Feature – Namespaces

Namespaces divide the identifier tables and other structures associated with kernel global resources into separate instances. These partition filesystems, processes, users, network stacks, Inter-process communication (IPC) objects, host names, and other components into separate pieces. For example, each filesystem namespace has its own root directory and mount table [2]. These distinct namespaces can then be bundled in any frequency or combination to provide a unique view of resources for each container and subsequent accessibility to them. The restricted view of resources for a process within a container can be extended to a child process. Configuration capabilities, such as remapped root file systems and virtual network devices, are some of the security solutions that can be enabled using the Namespaces feature. The assurance of a security solution based on namespaces depends on the methods used to enforce namespace isolation, which in turn depends on the kind of metadata associated with each namespace that implements the appropriate access control.

The namespace concept has expanded into a general framework for isolating a range of kernel global resources, the former scope of which was system-wide. Thus, the associated API has also grown to include several system calls. However, there are still some resources that are not namespace-aware (e.g., devices).

2.2 Linux Kernel Feature – Cgroups

Control Groups (Cgroups) are a kernel mechanism for specifying and enforcing hardware resource limits and access controls to a process or a group of processes. Their goal is to prevent a process from hogging all available resources and starving other processes and containers on the host. Thus, Cgroups isolate and limit a given resource over a group of processes to control performance or security. Controlled resources include CPU shares, RAM, network bandwidth, and disk I/O [5]. It can also be used for task control.

The security protection provided by Cgroups are:

(a) Preventing Denial-of-Service Attacks: It can provide protection against denial-of-service attacks preventing situations such as runaway containers by using features such as task freezing via SIGSTOP, setting limits on process ID (PID) using PID Cgroup to restrict the maximum number of processes per user, and specifying network control parameters such as buffer limits and traffic priority levels (enforced by iptables).

(b) Device Integrity Protection: It can restrict access to devices using mandatory access control (MAC) or using a feature that allows the specification a device whitelist.

The configuration of Cgroups is enabled by mounting a special Cgroup virtual filesystem (pseudo-filesystem) similar to /proc or /sys that allows viewing of the state of namespaces and controls. The vulnerability of this mechanism is that attacks, such as unmounting or mounting-over, can invalidate the resource limits set by Cgroups configurations. Cgroups can be configured and managed outside of the container management frameworks since it is a configuration feature purely associated with the kernel of the host OS.

2.3 Linux Kernel Feature – Capabilities

The *Capabilities* feature in Linux kernel helps to partition the extensive set of privileges available to root so that processes (in our context, containers) can be allocated just the privileges needed to perform a specific function. Prior to the introduction of the Capabilities feature, a process that needs to open network sockets must be run as a root to perform this single function. This meant that a bug in the corresponding binary, such as /bin/ping, could allow attackers to gain all privileges for the root on the system [6]. By enabling the capability CAP_NET_RAW, a version of ping can be created that has only the privileges enabled by this capability rather than full root privileges. The security consequence of this is that the potential attackers would gain significantly fewer privileges from exploiting the ping utility.

2.4 Kernel Loadable Modules (or Linux Security Module or LSM)

Kernel Loadable Modules, as the name implies, are modules loaded into the Linux kernel and provide security functions. Examples include SELinux, AppArmor, and Seccomp. SELinux and AppArmor enable specification and enforcement of mandatory access control (MAC) on processes and objects. Seccomp enables specification of system call restrictions, and thus reduces the Linux kernel attack surface.

2.5 Application Container Security Configuration Process

The Linux host OS kernel features—such as namespaces, Cgroups, and Capabilities—can be leveraged to create a secure configuration for each container. Many container runtime products offer APIs to create secure configurations for containers within a host. A typical container runtime, generally accessed through a client, contains a library that directly makes the syscalls and performs work on behalf of its client such as creating the required kernel namespaces, Cgroups, and management of capabilities. Other administrative functions that may have security implications (e.g., lack of availability due to uneven workloads) such as distribution of containers across hosts and the creation of host clusters are managed by a set of tools called Orchestrators.

3 Hardware-based Security Solutions for Containers

The *Container Security Guide*, under the topic of Hardware Countermeasures, recommends a trusted computing model that starts with the measured/secured boot, provides a verified system platform, and builds a chain of trust rooted in hardware. This chain of trust then extends to bootloaders, the OS kernel, and the OS components to enable cryptographic verification of boot mechanisms, system images, container runtimes, and container images. The technical solutions for implementing a trusted computing module (TPM) for a containerized host are outlined in [7]. Two such approaches are discussed in this document as well as the security assurance required for each solution.

Both approaches involve a combination of hardware-based, or physical, TPM and a software-based vTPM (virtual TPM). The difference between the two approaches is in the location where vTPM is placed in the container stack. The security solution where vTPM is placed in the Linux kernel is discussed in section 3.1, and the solution where vTPM is placed in a dedicated container is the topic of section 3.2.

Building a TPM architecture is not the only type of approach for providing trust rooted in hardware for the container stack. Another type of approach that has been proposed is to leverage the trusted execution support of some CPU architectures to protect processes running in a container against attacks from sources inside the same container stack. This includes privileged software in the same stack such as the container runtime and host OS kernel [8]. A mechanism or security solution based on this type of approach is discussed and analyzed in section 3.3.

3.1 vTPM in the host OS Kernel – Security Assurance Requirements

In an architectural approach suggested in [7], a software-based module called vTPM (virtual TPM) is placed into the OS kernel. To make this module available to several containers, it needs to be virtualized. This is accomplished using a kernel module that provides an arbitrary number of software-based vTPMs, which are exposed to containers through the usual mechanisms and present a character device type interface to the container userspace. This functionality can be implemented by having the container runtime (or container manager) ask the host OS kernel to create a new vTPM and assign the virtual device to a container. The vTPMs are linked to the TPM implemented in the hardware platform (referred to as “physical TPM”) that hosts the container stack. The schematic diagram of this architectural approach is illustrated in figure 2.

The security assurance requirements for the above discussed architectural approach can be looked at for the following scenarios:

<u>The host OS is completely trusted:</u> The trust-in-host OS can be established by extending the root of trust from the hardware using the hardware-based, or physical, TPM. Since the host OS is trusted to prevent unauthorized access by containers and processes, it can also be trusted to prevent unauthorized access to the in-kernel vTPM. Moreover, there is the assurance that containers cannot modify the host kernel by loading new modules or by exploiting vulnerabilities in the kernel. Containers can therefore reliably attest to their own state by using the hash extend feature of the vTPM.

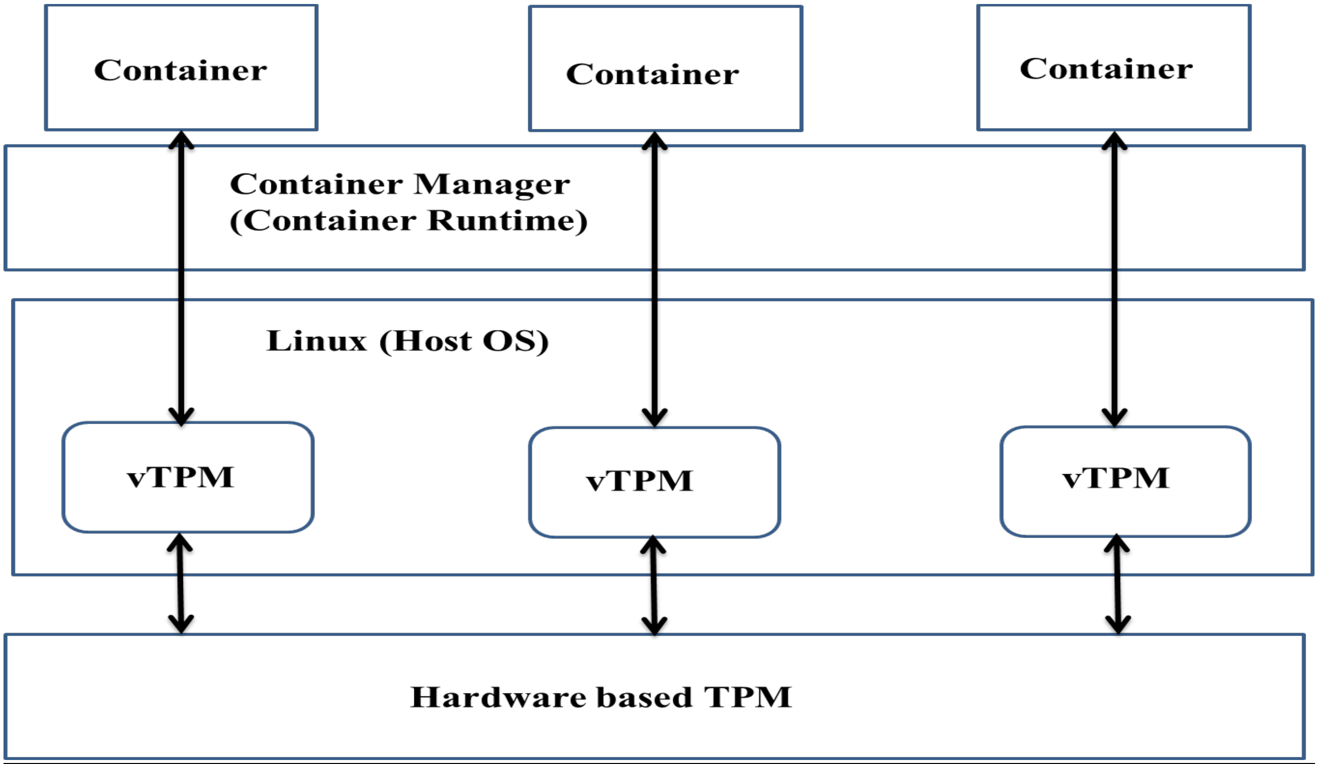

Figure 2 – vTPM Implemented in a Kernel Module

<u>The host OS is not completely trusted, and independent trust is needed on vTPM:</u> To implement trust on vTPM, a scheme using the same mechanism used for establishing hardware TPM (physical TPM) trust has been referred to in [7]. In the physical TPM, the hardware platform provider signs an endorsement key (EK) stating that the TPM is trustworthy. This is then extended by giving each vTPM instance its own endorsement key and deploying protocols for signing the endorsement keys of vTPMs using of the hardware-based TPM.

3.2 vTPM in a dedicated Container – Security Assurance Requirements

The software-based vTPM with the same functionality described in section 3.1 is built and hosted in a dedicated container (referred to as vTPM management container). The schematic diagram of this architectural approach is given in figure 3. This vTPM has two primary features:

(a) Access to hardware-based (physical) TPM

(b) Exposes the vTPM interface to other containers through a communication channel, which can be a local UNIX domain socket or another IPC mechanism. If the IPC mechanism is employed, the container using the vTPM service requires an additional piece of software (denoted as “adapter” in figure 3) that presents the IPC interface as a standard character device. In the container that is hosting the vTPM, a daemon will process requests from other containers instead of a kernel module as it was in the previous case.

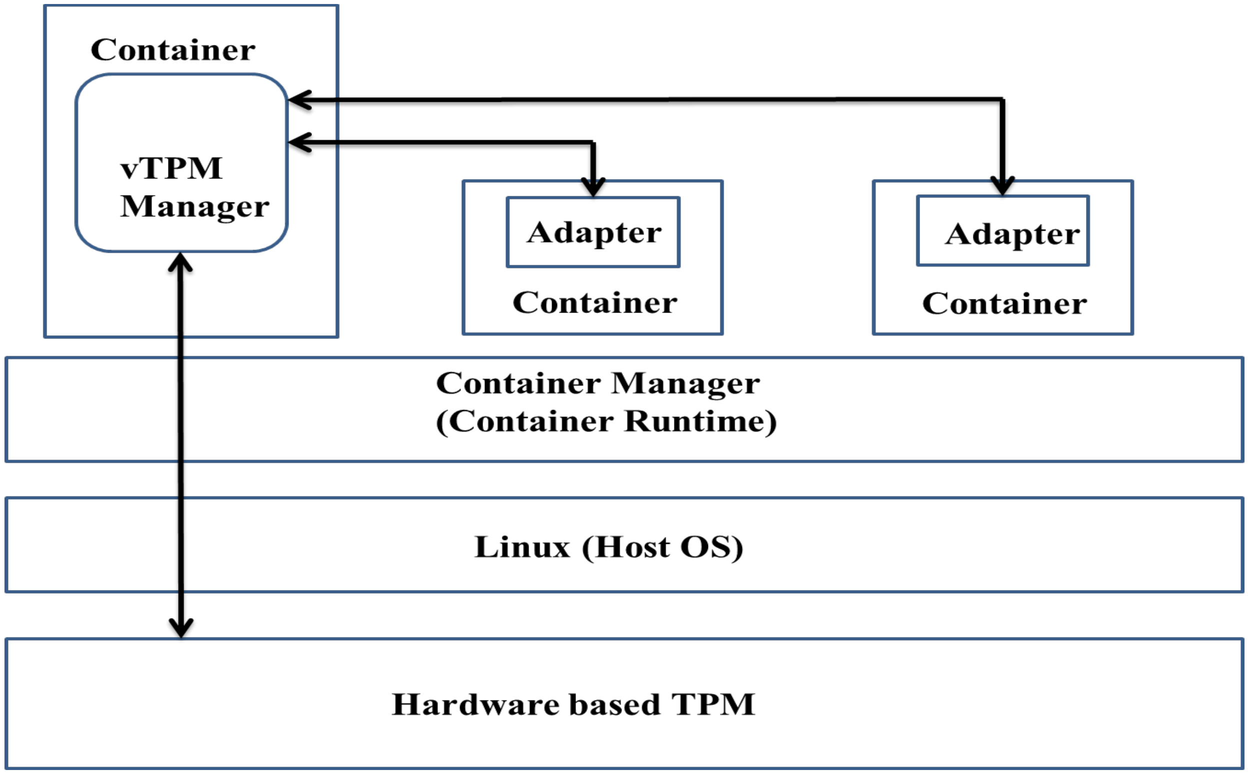

Figure 3 – vTPM located in a dedicated Container

The security assurance provided by this architectural approach is the same as the one provided by the host OS in the container stack. A host OS, such as Linux, provides isolation between processes belonging to different containers through the Namespaces feature. If this functionality works correctly, no process belonging to a different container can access the state of the vTPM deployed in a dedicated container. In other words, the security of this implementation is jeopardized only in the event of a container escape attack. Still, this approach provides less protection than the approach in section 3.1 (vTPM in the host Kernel) since the kernel is more reliable in limiting the kind of access it exposes to the userspace.

3.3 Leveraging Trusted Execution Support of Hardware

In 2015, Intel released the Software Guard eXtensions (SGX) [8] for their CPUs, which provided the hardware mechanism for protecting user-level software from privileged system software using the concept of secure enclaves. An enclave page cache (EPC) is a region of protected physical memory where application code and data reside and are protected by CPU access controls. When code and data in EPC pages are moved to DRAM, they are instantaneously encrypted using an on-chip memory encryption engine (MEE) and then decrypted when they are transferred from DRAM to EPC pages. The integrity of the enclave memory itself is also protected by mechanisms that detect memory modifications and rollbacks. Thus, enclaves are trusted execution environments provided by SGX to applications residing in the container.

4 Assurance Requirements for Host OS Protection

4.1 Requirements for Generic Host OS Protection

Installing a container-specific OS (as opposed to a generic OS distribution), keeping OS versions up-to-date and patched, logging features that can track anomalous accesses to the OS, and executing privileged operations form the crux of Host OS countermeasures in the *Container Security Guide*. In addition to the above countermeasures, it is also a good OS security practice to disable all unused interfaces (Serial or Proprietary) on the host and minimize the user and administrative accounts and groups. In addition to these, there are Linux-specific patches, such as grsecurity [9] and PaX [10], that are available for Linux distributions. All measures combined should provide the following security assurance for the host OS:

(a) Prevent manipulation of program execution by modifying memory (e.g., buffer overflow attacks)

(b) Prevent attempts to reroute code to existing procedures (e.g., system calls in common libraries)

4.2 Assurance Requirements for Host OS Protection for Container Escape

The host OS should be protected to mitigate threats that result from container escape or breakout, and all containers should be protected from other containers on the host. There are many solutions available in Linux environments that enable these protections, but the three solutions analyzed in this document are SELinux, AppArmor, and Seccomp, all of which utilize kernel-loadable modules (referred to using the acronym LKM, or Linux Kernel Module). SELinux, or Security Enhanced Linux, can be used to assign labels (e.g., type) and categories to processes and objects (e.g., files, sockets) and specify access restrictions (Mandatory Access Control or MAC) between resources belonging to certain combinations of labels and categories. For example, a specific SELinux label can be applied to a container to enforce a security policy (e.g., a container hosting a Webserver can only open ports 80 or 443) [6]. AppArmor is another LKM product that helps enforce mandatory access control policies by applying profiles to processes that enable restriction of privileges they have at the level of Linux capabilities and file access. The controls are thus data-centric and are at a coarser level of granularity compared to SELinux. SECure COMPuting (Seccomp) is a module that can define and enforce an access control method that enables specification of the number of system calls available for an application within a container to interface with the kernel. Limiting system calls provides a restricted execution environment and thus reduces the kernel attack surface. The allowed list (i.e., whitelist) and prohibited list (i.e., blacklist) of system calls for a process are set up using the syscall filter [11].

The overall goal of the kernel-loadable modules, or LKMs, described above is to provide another level of security checks on the access rights of processes and users beyond that provided by the standard file-level access control (discretionary access control, or DAC) in Linux [6]. This goal then drives the following security assurance requirements that need to be satisfied:

(a) A user authorized to run applications in the container should not be allowed access to the above described kernel-loadable modules

(b) If using SELinux, the chcon utility used to label the files and parent folders should be used at the correct levels in the file system hierarchy such that it results in least privileges

(c) If using Seccomp, both a syscall whitelist (a list of allowable calls) and a syscall blacklist (a list of prohibited calls) should be generated. The choice of syscalls in the whitelist for a container should be based on type of application(s) hosted in the container, deployment situation, and container size. The syscalls included in the blacklist are for high risk, possibly vulnerable, known dangerous, and explicitly disallowed ones [11]. Some examples in this category include syscalls that allow for loading kernel modules, rebooting, triggering mount operations, and other administrative calls.

(d) If using Seccomp, the sandboxes created by seccomp filters must not allow the use of the ptrace command. If ptrace is allowed, the tracer can modify the process's system call to bypass the filter and therefore call blocked or restricted system calls.

(e) It should be possible to create container-specific profiles using a combination of configuration options provided by these security modules

(f) A minimal configuration feature that should be available is one that allows for the partitioning of containers in the host to different security domains

(g) It should prevent containers' ability to mount/remount sensitive directories and/or specific system directories critical to security enforcement (Cgroups, procfs, sysfs)

(h) It should be possible to create a security profile for the administrators of container runtime using a combination of the above features

5 Assurance Requirements for Container Runtime Configuration

As already described in section 2.5, all security configuration parameters for containers, except for those dealing with cluster management and scheduling, are set using APIs provided by container runtime. Although most of them involve Linux kernel features (Namespaces, Cgroups, Capabilities) and Linux kernel modules, these tasks have been included under this section since they are performed by the container runtime making syscalls to Linux host OS interfaces. The overall organization of this section is as follows:

(a) Section 5.2 discusses configurations involving Linux's Namespace feature, which provides isolation for various resources

(b) Section 5.3 discusses configurations using the Cgroups feature, which is primarily utilized for setting resource limits and thus preventing denial of service attacks

(c) Section 5.4 discusses configurations using the Capabilities feature, which enables the allocation of least privileges

(d) Section 5.5 discusses the configurations for device isolation, which can be enabled using a combination of Cgroups and kernel-loadable MAC enforcement modules

(e) Section 5.6 discusses configuration parameters that can be set at the time of launching the containers rather than being pre-configured using the functions discussed above

Before analyzing these functions, the need for a configuration feature for the container runtime itself is outlined in section 5.1.

5.1 Requirements for Secure Connection

Container runtime module is implemented with a daemon that listens through a Unix socket and thus enables remote administration of the runtime. It is possible under certain circumstances for members in the administrative group to change the Unix socket to a TCP socket [10]. Any connection to this TCP socket can allow attackers to pull and run any container in privileged mode, thereby giving them root access to the host. The security assurance requirement for the TLS connection involves the encryption and authentication of both sides (container runtime module as well as the client tool used for remote administration) of the connection before establishing the TLS session.

5.2 Requirements for Isolation-based Configurations

5.2.1 Process Isolation for Containers

Process Isolation is a core security requirement for containers to ensure the integrity of various applications running in different containers as well as in the host. A process isolation mechanism in a container environment should meet the following requirements [4]:

(a) Ability to distinguish processes running in different containers from each other and from those running on the host

(b) Limit cross-container process visibility

(c) Prevent certain type of attacks such as:

(i.) A process running in one container influencing a process running in another container using interfaces provided by the OS for process management (e.g., signals and interrupts)

(ii.) A process running in one container and directly accessing the memory of a process running in another container by using special system calls (e.g., the ptrace() allows a debugger process to attach and monitor the memory of a debugged process)

To provide process isolation, a Linux kernel feature called process id (PID) namespace is used. A PID namespace is a mechanism that groups processes and controls their ability to see (e.g., via proc pseudo-filesystem) and interact (e.g., sending signals) with one another. A PID namespace is created using clone() or unshare() system call and is associated with one or more containers. The first process carries the id PID1, and the identifiers for subsequent processes increases sequentially. Thus, the PID namespaces feature also provides PID virtualization. Two processes in different PID namespaces can have the same PID.

5.2.2 Filesystem Isolation for Containers

The goal of filesystem isolation is to prevent illegitimate access to filesystem objects from one container to another and from any container to the host. The filesystem is an OS interface that allows processes to store and share data as well as interact with one another. Access to data for a container application is determined by its access to file systems through the filesystem mount points. Therefore, access to data can be restricted by making the list of filesystem mount points visible and accessible to a container application. This is accomplished through the mount namespace. First, a named mount namespace is created along with a set of file system mount points. This mount namespace is then associated with a process that can only see and issue system calls such as mount ( ) or unmount ( ) on those mount points. It also operates on files that are within that mount namespace and accessible through those mount points. The following are the security solutions for filesystem isolation and their limitations:

(a) All Linux-based OS virtualization solutions utilize a *mount namespace* that allows for the separation of mounts between the containers and the host. This is intended to facilitate customization of the environment visible to users and processes. This feature does not guarantee data isolation between the containers since containers inherit the view of filesystem mounts from their parent and can access all parts of the filesystem even though each container is created within a new mount namespace.

(b) The typical solution for process filesystem access containment is by using the chroot ( ) system call, which binds a process to a subtree of the filesystem hierarchy. This allows a container to share resources with the host by mounting them within the subtree visible inside the container. However, this feature cannot provide the requisite protection in the presence of privileged processes (i.e., processes with the CAP_SYS_CHROOT privilege), which can escape the chroot jail due to the fact that the chroot ( ) system call only affects the pathname resolution.

(c) A better protection for filesystem objects is provided by modifying the root filesystem for processes in a container as opposed to just modifying the root directory (which the chroot

( ) system call enables) [4]. This is enabled by the pivot_root ( ) call, which moves the mountpoint of the old root filesystem to a directory under the new root filesystem and puts the new root filesystem in its place. This provides filesystem level protection since the old root filesystem can be unmounted when it is carried out inside the mount namespace of the container, thus rendering the host root filesystem inaccessible for processes inside the container.

(d) Another filesystem-level protection strategy is to disallow mounting and unmounting of filesystems for processes running inside a jail by default and enforce granular control of this privilege using options in the allow_mount* command.

(e) Another mechanism to strengthen filesystem isolation is to designate a separate user namespace per container, which maps the user and group ids to a lesser privileged range of host UIDs and groups.

Because of the limitation of each of the above security solutions, the assurance requirements for total filesystem-level protection involves a combination of configurations including mount namespace, chroot, pivot_root, and user namespace needed for:

- Isolating mount points by mount namespace
- Changing the root directory for each process using chrooot
- Changing the root filesystem visible to each process (container) using pivot_root
- Restricting user access scope using user namespace

5.2.3 IPC Isolation for Containers

Inter-process communication (IPC) isolation for containers means that processes in a container must be restricted to communicate via certain IPC primitives only within that same container. An IPC object (or associated mechanism) can be either a filesystem-based IPC object or non-filesystem-based. Filesystem-based IPC objects, such as domain sockets and named pipes, can be isolated using a combination of mount namespace and pivot_root features (section 5.2.2 above) since they prevent processes from accessing filesystem paths outside of their own container.

However, there are other IPC objects such as System V IPC objects, semaphore sets (arrays), shared memory segments, and message queues. These IPC objects can be isolated in Linux with the help of IPC namespaces that allow the creation of a completely disjointed set of IPC objects. Each IPC namespace has its own set of System V IPC identifiers and its own POSIX message queue filesystem. Objects created in an IPC namespace are visible to all other processes that are members of that namespace but are not visible to processes in other IPC namespaces. IPC objects accessible for a process can be listed using the *ipcs* command and removed using the *ipcrm* command.

5.2.4 Network Isolation for Containers

Network level isolation for containers is provided through the network namespace feature. For each network namespace that is created, a set of network devices, IP addresses, IP routing tables, /proc/net directory, and port numbers can be associated with it. Each container can have its own virtual network device and applications that bind to the per-namespace port number space. Suitable routing rules in the host system can direct network packets to the network device associated with a specific container. It is therefore possible to have, for example, multiple containerized web servers on the same host system with each server bound to port 80 in its (per-container) network namespace.

Network connectivity is a core requirement for all production grade applications running on containers such as web apps and multi-tier apps. The containers can be connected using a logical IP network called the overlay network. The typical network configuration on a container platform (consisting of containers, container runtime, host OS and the physical host) involves creating a network bridge on the container host. Each container on a host is connected to that bridge. A router captures Ethernet packets from its bridge-connected interface in promiscuous mode, and captured packets are forwarded over the user datagram protocol (UDP) to router peers running on other container hosts. These UDP ”connections” are duplex, can traverse firewalls, and can be encrypted [12]. Each container is connected to the bridge using a layer 2 (link layer) virtualized network interface (VNI) with a valid Link Layer address or a Network Address Translation (NAT) for layer 3 connectivity. The Linux Layer 2 network isolation is based on the concept of Network Namespace, which allows for the creation of several networking stacks that provide a view of being completely independent of the containers [4].

The simplest configuration for network isolation using layer 2 VNI involves defining a pair of virtually linked Ethernet (veth) interfaces. One of the interfaces is assigned to the same network namespace as the container and the other to the host namespace. A virtual link is then established between the two interfaces, thus connecting the container to physical networks. There are two options for enabling this link [4]:

(a) Network Bridge Device: The veth interface and the host physical interface are connected using a virtual network bridge device. In this option, all container and host interfaces are attached to the same link layer bridge and thus receive all link layer traffic on the bridge.

(b) Routing Tables: Another option is to utilize routing tables to forward the traffic between virtual network interface (to which the container is connected) and physical network interfaces (resident at the host). In this option, containers can communicate with each other only when a network route is explicitly provided.

Security Analysis: The network isolation functionality provided by these two options forces a container process to use a designated virtual network segment or a designated network route (e.g., over a VPN connection). Between the two options, the routing table use presents a slightly higher security assurance than the network bridge device solution since the latter allows a container address to be visible to all containers connected to the bridge.

Another approach to provide network connectivity for containers is to use the MACVLAN interface [13], which also allows each container to have its own separate link layer address. The Virtual Ethernet Port Aggregator (VEPA) is the most widely used mode for configuring this option for isolating the containers. However, complete assurance of network isolation can be provided at the process level in containers only if the namespace-based approaches are augmented with mandatory access controls (MAC) and the isolation of the process from other global namespaces.

5.2.5 User and Group-level Isolation for Containers

Some processes may need some subset of root privileges. The user namespaces feature can be used to restrict the privileges of some user IDs to that needed subset. The user namespace isolates the user and group ID number spaces. In other words, a process's user and group IDs can be different inside and outside of a user namespace. The most interesting case here is that a process can have a normal unprivileged user ID outside of a user namespace while at the same time having a user ID 0 inside of the namespace. This means that the process has full root privileges for operations inside the user namespace, but is unprivileged for operations outside the namespace.

Starting in Linux 3.8, unprivileged processes can create user namespaces, which opens a raft of interesting new possibilities for applications. Since an otherwise unprivileged process can hold root privileges inside the user namespace, unprivileged applications now have access to functionality that was formerly limited to root [4].

5.3 Requirements for Resource Limiting Solutions

The primary protection mechanism for denial-of-service attacks in Linux container environments is the Cgroups feature that enables setting limits for various resources. The “limits” specification feature is restricted not only to hardware artifacts such as CPU, memory, and storage, but also to processes and tasks. In addition to the limits feature, Cgroups enables the designation of a collection of potential “resource hogging tasks” that can be frozen by sending a SIGSTOP signal. It can later be unfrozen by sending a SIGCONT signal [11].

In addition to its main role of preventing against denial-of-service attacks, the Cgroups feature also provides marginal network-level protection with a method (using network classifier Cgroup) that tags network packets with a “classid” value. This can then be used as a parameter for filtering certain packets. (The classid value can also be used for priority handling based on Quality of Service (QoS) requirements, though that feature falls under performance enhancement and not strictly security.)

The following table provides the list of hardware resources for which the Cgroups feature either enables setting up of resource limits or access control.

Table 1– Linux Resource Control using Cgroups

Resource	“Limit” Feature or Access Control
CPU	Specific number of CPUs or amount of “CPU Shares” for a group of processes
Memory	“Hard” and “Soft” memory allocation units for a group of processes
BLKIO	Set disk read or write speeds, operations per second, queue controls, and wait times on block devices designated by major and minor numbers; provides more granular access control compared to filesystem specific controls

Devices	Create a whitelist for devices based on either: (a) Type (character vs block) or (b) Major and Minor numbers

Cgroups configuration should provide the following assurances:

(a) It should not expose container host information, such as the kernel ring buffer via dmesg, which can assist in kernel exploitation or information leaks

(b) It should not allow local disk access, even within user namespaces and mount restricted namespaces via raw disk, device, or mknod access [11].

5.4 Requirements for Least Privilege Configuration for Containers

As already mentioned, the Capabilities feature in Linux can be used to partition the set of root privileges. All container runtime products, such as LXC, Docker, and CoreOS Rkt, come with a default capability profile where some capabilities for containers are enabled and some are disabled [11]. Due to the privilege needs of the application running in the container, some of the defaults have be modified (i.e., some capabilities that have been enabled by default need to be disabled, and some capabilities disabled by default need to be enabled). However, for most applications hosted in containers, the following assurance requirements must be satisfied while configuring the Capabilities feature in Linux:

(a) Capabilities that provide the privilege to manipulate a non-name spaced kernel parameter (e.g., Sys Time) will have the effect of that parameter modified not only for the container but also for the host and for all other containers. Hence such capabilities (e.g., CAP_SYS_TIME) should not be enabled.

(b) Capabilities that provide the broad set of privileges almost equal to that of the root should not be enabled (e.g., CAP_SYS_ADMIN).

(c) There is no need to enable the capability CAP_SYS_MODULE, which allows for the loading and unloading of kernel modules as this will lead to insecure privilege escalation.

(d) The Capabilities feature should always be used in conjunction with user namespace as any privilege escalation to the process due to enabling some Capabilities by error will be limited to the namespace.

5.5 Requirements for Device Isolation solutions

In Linux, access to devices is enabled by device nodes, which are special files that provide an interface to the host device drivers. Device nodes are separated from the rest of the filesystem, and their nodes are placed in the /dev directory. These nodes are not namespace-aware. The creation of device nodes is performed by the *udevd* daemon process issuing the *mknod* system call. The permission for a process to create device nodes (for accessing block or character devices) is provided by the *CAP_SYS_MKNOD* capability. Containers are given access to device nodes if the corresponding devices are to be shared among containers or between different containers and the host. However, device nodes are security-sensitive since they provide interfaces to device drivers. These drivers present significant attack vectors because they expose interfaces (particularly the storage interface) to code running in the kernel space, which may be abused to gain illegitimate data access, escalate privileges, or mount other attacks.

One possible solution for providing device-level isolation between containers is the use of "device namespace," provided the referenced input/output (physical) devices are namespace-aware. Unfortunately, many Linux kernel distributions do not support the device namespace feature. Where available, this feature can be used to create virtual devices for each container, which can be multiplexed for access to a physical host device. Further, when Linux device drivers controlling physical devices are not namespace-aware and the devices assume only one controlling master host, access privileges for them are hard to securely grant for unprivileged containers unless the device is used exclusively by a single container.

In the absence of the device namespace feature, two features are utilized for controlling access to devices for containers. They are: (a) control groups, or Cgroups; and (b) Mandatory Access Control (MAC) enforcement. The Cgroups subsystem for devices is used to create a whitelist, formatted for devices based on type (i.e., character vs block) and device major and minor numbers. The wild card "all" applies to all device types and major and minor numbers, and it is typically used as a default deny before whitelisting explicit devices [11].

There are two MAC enforcement methods available in Linux environments: Security-Enhanced Linux (SELinux) and Apparmor. In SELinux, Multi-Level Security (MLS) labels or classification labels are applied on processes and data/devices, and the system applies fine-grained policy and type enforcement across the different labels. AppArmor is another MAC system that offers a pathname-based access control (as opposed to filesystem nodes within SELinux). The restrictions can be aggregated to define a profile for a specific application, process, or container. A common weakness for all MAC systems is that the controls it provides can be subverted through direct execution of system calls.

The assurance requirements for device isolation solutions therefore are:

(a) All containers must be prevented from creating new device nodes, and the CAP_SYS_MKNOD capability should not be enabled for them

(b) All mountpoints inside containers should have the nodev flag set to prevent them from being used to create files to access device drivers

(c) All containers should only be allowed to access the following set of devices since they are characterized as safe [4] due to observations given below:

- *Purely virtual devices* – such as pseudo-terminals and virtual network interfaces; the security guarantee comes from the fact that these devices are explicitly created for each container and not shared
- *Stateless devices* – such as random, null, and others; sharing these devices among all containers and the host is safe because they are stateless
- *User namespace-aware devices* – if the device (through the device driver code) supports verifying capabilities of the process in the corresponding user namespace,

then such a device can be safely exposed to a container since the specified restrictions will be enforced

(d) When Cgroups and MAC enforcement systems are both used for controlling access to devices, care should be taken to ensure that their respective rules do not create conflict.

5.6 Requirements for Container Launching Options

Every container runtime product has a command to launch containers that carry many options. The assurance requirements associated with the secure use of this command are stated as a set of options that should be avoided [4]. As a best security practice, containers should not use options that will enable sharing any namespaces associated with the container host when launched [11]. If this is not the case, it may not only enable the container to view the resources/objects associated with that namespace but also manipulate those resources/objects by subverting the isolation provided by static configuration of namespaces for the container. The following table provides the list of namespaces for which sharing the corresponding host counterpart should not be used in the container launch options.

Table 2 – Prohibited Options for Container Launching

Namespace/ Example Resource-Object	Brief Description	Security Threat
UTS	All containers are assigned their own UTS namespace and thus have no need to know the UTS namespace of the host	Processes within the container can see and manipulate the hostname and domain of the host
IPC/ Shared Memory Segment	Shared Memory segments for inter-process communication between application modules are set up for faster communication as they are faster than REST API calls	Processes within the container can see and manipulate host IPC object
Filesystem	Host-sensitive directories should not be mounted in read-write mode as container volumes	Gives containers the ability to modify the files in those directories with a potential to jeopardize host security
Setting net=host in the container launching command	The networking mode for the container should not be set equal to host	This will give privileges to a container that only a host should have (e.g., shutting itself down) or access to networking services that only

Namespace/ Example Resource-Object	Brief Description	Security Threat
		the host needs
Publishing container ports to the host	This is done for setting up communication to and from that container	The default option of publishing to all interfaces should not be used; by specifying the interface that the port should bind to explicitly, traffic into and from the container is restricted to the given interface
Inter-container communication	If it exists, the option to enable blanket inter-container communication must not be enabled; instead, explicit communication channels must be set up between two containers that need to communicate.	Any compromised container can attack any other container on the host

In addition to container launch options that involve objects shared with the host, there are some parameters exclusively applicable to the container that should be set when launching containers.

(a) Containers should always be launched with a specific memory limit to prevent denial-of-service attacks or certain applications leaking memory that may eventually consume all the memory on the host

(b) Containers should always be launched by specifying the number of CPU shares. The default value (Total CPU/number of containers) may not be sufficient for some containers, resulting in denial of service. The number of CPU shares assigned to a container should be such that no container can starve others with default settings. Further, if there exists a group of containers that dominate others in CPU usage, then a lower default value should be assigned to containers in that group to ensure fair distribution of CPU shares.

(c) If the host OS Linux distribution supports a MAC system (e.g., SELinux), a policy template should be set up, the container engine should be started with an option to recognize the template, and the container launching API should have an option to recognize the policy template parameter and include it as part of the launch parameter.

(d) Containers should be launched only with “required” capabilities by initially dropping all capabilities and then adding only the required ones. The following capabilities in general should not be present (i.e., NET_ADMIN, SYS_ADMIN, SYS_MODULE).

6 Assurance Requirements for Image Integrity Solutions

The integrity of the container images is of paramount importance since they are converted to running instances, some of which may host mission-critical applications. The image countermeasures covered in the *Container Security Guide* include recommendations for monitoring images for malware and other vulnerabilities, proper image configuration, separating secrets from image files, and ensuring trust in images through cryptographic signatures and regular updates. The security solutions needed for carrying out these recommendations should include the following assurance requirements:

(a) There should exist a means to create metadata linking each image to its base image

(b) There should exist a feature to rebuild the image automatically if the linked base image changes [6]

(c) When any changes are made to the base image or dependent image (e.g., patching a vulnerability), changes should not be made to the running containers. Instead, the corresponding image should be recreated and the container re-launched using the modified image. Thus, a single master, or golden image, is to be maintained for any service.

(d) When employing "image signing" solutions for digitally signing and uniquely identifying each image, the following requirements should be met [6]:

1. There should be a robust key management to minimize the possibility of key compromise. One approach is to have a PKI system that issues a certificate to each developer exclusively for signing the image. The private key associated with this certificate will then be the "signing key" that is used to sign all container images in a repository.

2. Replay attacks must be mitigated by embedding expiration timestamps in signed container images. Alternatively, a special key can be used to sign the metadata for the repository, ensuring that the images in the repository do not contain stale versions of the image with valid signatures.

(e) In addition to creating a unique identifier for an image using digital signatures, the integrity of individual components of the image can be ensured by using labels such as key/value pairs for each component.

(f) Images should be built such that the application(s) in them are not used for any privilege escalation attacks. This can be achieved by disabling the chmod a-s command, which removes the suid bit, or removing setuid and setgid binaries in them [6].

7 Assurance Requirements for Image Registry Protection

The suggested registry countermeasures in *Container Security Guide* include developing secure connections to registries and ensuring that they do not contain out-of-date vulnerable images by pruning them out through an automated process or controlling their accidental deployment through use of discrete version numbers. Some assurance requirements unrelated to these countermeasures yet still critical to processes involving creating, posting, and removing images into and from registries are:

(a) The number of accounts accessing the registry must be limited since the common threat in some environments is account hijacking when a diverse set of clients has access to a container registry. One such environment is the registry maintained by cloud service providers who offer container services.

(b) The permission to create container image registries and add or remove content to registries must be cryptographically protected.

8 Assurance Requirements for Orchestration Functions

The use of an Orchestration platform (consisting of a suite of tools) in a containerized infrastructure is intended to perform the following functions:

- Enable the definition of a cluster (a named group of container hosts that can be managed as a single entity) and schedule containers into the cluster. The cluster configuration should support specification of parameters such as the amount of CPU/Memory to reserve, the number of replicas (i.e., duplicate copies of same container to be run), and the circumstances under which a container should continue to run or be taken offline.
- Enable automated deployment of containers in various clusters/hosts (container scheduling). This is achieved by integrating various automation tools to execute automation scripts as part of an orchestrated workflow and to obtain feedback and status results for those automation tasks. This kind of integration depends on the interfaces that the automation tools provide and the type of formats (open or closed) that they follow [14].
- Provisioning, or defining new container hosts and attaching them to existing clusters

The suggested orchestration countermeasures in the *Container Security Guide* include granular access control of administrative actions based on hosts, containers and images as parameters, use of enterprise-grade authentication services using strong credentials and directories, and isolating containers to separate hosts based on the sensitivity level of the applications running in them. In addition to these countermeasures, the orchestration artifacts should satisfy the following security assurance requirements:

(a) Clusters should have capabilities for logging and monitoring the resource consumption patterns of individual containers to avoid unanticipated spikes in resource usage leading to non-availability of critical resources

(b) The Orchestration platform must be usable on containerized infrastructures with more than one host OS. In other words, the orchestration tools used must be container-host OS-neutral. Using different tools for different container host OS platforms increases the probability of denial-of-service attacks in those environments since the enterprise is not able to obtain a global picture of resource usage for all running containers in the entire containerized infrastructure of the enterprise.

9 Adverse Side effect of some Security Solutions

While discussing a security solution (e.g., using mount namespace) in the context of a security objective (i.e., filesystem isolation), certain augmenting solutions are recommended since the solution under discussion cannot meet the objective by itself. However, there are some security solutions that, irrespective of any augmenting controls, impose certain limitations on the functionality and performance of certain container functions. Despite their direct impact affecting only functional and performance aspects, they may have an indirect impact on certain security parameters. These are discussed below.

9.1 Resource Limiting using Cgroups

The use of Cgroups to limit resource access for processes/containers is included as a security solution because of its potential to mitigate the chances of denial-of-service situations. The Linux control groups (Cgroups) subsystem is used to group processes and manage their aggregate resource consumption. It is commonly used to limit the memory and CPU consumption of containers. A container can be resized by simply changing the limits of its corresponding Cgroup. However, processes running inside a container are not aware of their resource limits [2]. For example, a process can see all the CPUs in the system even if it is only allowed to run on a subset of them; the same applies to memory. If an application attempts to automatically tune itself by allocating resources based on the total system resources available, it may over-allocate when running in a resource-constrained container, thus resulting in denial-of-service to other applications within the same container [2].

9.2 Syscall filters using Seccomp

Setting up system call filters (with whitelist and blacklist) using Seccomp is used as a security solution since system calls are not namespace-aware (ruling out the use of the namespaces feature), though in the presence of malicious processes, this can introduce accidental leakage between containers. However, the choice of system calls to be allowed is based on a current set of applications in the container, and this security solution has the potential to introduce application incompatibility since applications can be migrated between containers for load-balancing reasons.

10 Summary and Conclusions

The security solutions analyzed in this document can be summarized as follows:

(a) Providing authenticity and attestation of integrity for software components of a container stack such as Linux (Host OS), container runtime, and the containers using hardware-based root-of-trust solutions such as TPM and vTPM

(b) Hardware-based protection for shielding one container from another as well as containers from higher privileged software, such as Linux kernel, using the safe execution model provided by hardware architecture (e.g., Intel SGX)

(c) Linux kernel features (Namespaces, Cgroups, Capabilities) and loadable kernel module (LKM) features for protection of the Linux kernel itself and for protecting one container from another

(d) Protection measures for container runtime, container images, container registry, and container orchestration tools.

The conclusion from the analysis is that every security solution must satisfy some security assurance requirements to effectively provide necessary and sufficient security guarantees.

Appendix A—Acronyms

Selected acronyms and abbreviations used in this paper are defined below.

EPC	Enclave Page Cache
IPC	Inter-process Communication
MAC	Mandatory Access Control
MEE	Memory Encryption Engine
NAT	Network Address Translation
PID	Process ID
PKI	Public Key Infrastructure
SGX	Software Guard eXtensions
TPM	Trusted Platform Module
UDP	User Datagram Protocol
UTS	UNIX Timesharing System
VM	Virtual Machine
VNI	Virtualized Network Interface

Appendix B—References

[1] NIST Special Publication (SP) 800-190, Application Container Security Guide (Draft), National Institute of Standards and Technology, Gaithersburg, Maryland, April 2017.

[2] W.Felter, A.Ferreira, R.Rajamony, J.Rubio., An Updated Performance Comparison of Virtual Machines and Linux Containers, IBM Research Report, RC25482 (AUS1407-001), July 21, 2014.

[3] Practical Guide to Platform-as-a-Service, Version 1.0, Cloud Standards Customer Council, http://www.cloud-council.org/deliverables/CSCC-Practical-Guide-to-PaaS.pdf, September 2015.

[4] E. Reshetova et al. Security of OS-level virtualization technologies, Cornell University Library, https://arxiv.org/abs/1407.4245

[5] T. Combe, A.Martin, R. Pietro., To Docker or Not to Docker: A Security Perspective, IEEE Computer, September/October 2016, pp, 54-62.

[6] A. Mouat, Docker Security, O'Reilly Media, 2015.

[7] S. Hosseinzadeh, S. Lauren, and V. Leppanen, Security in container-based Virtualization through vTPM. In Proceedings of IEEE/ACM 9th International Conference on Utility and Cloud Computing, Shanghai, China, Pages 214-219, December 2016.

[8] S. Arnautov et al., SCONE: Secure Linux Containers with Intel SGX, Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16). November 2–4, 2016 • Savannah, GA, USA (https://www.usenix.org/system/files/conference/osdi16/osdi16-arnautov.pdf).

[9] grsecurity, https://grsecurity.net/features.php

[10] Home Page of The PaX Team, https://pax.grsecurity.net/

[11] A. Grattafiori, Understanding and Hardening Linux Containers – Version 1.1, NCC Group Whitepaper, June 2016.

[12] N. Kratzke, About Microservices, Containers and their Underestimated Impact on Network Performance, CLOUD COMPUTING 2015: The Sixth International Conference on Cloud Computing, GRIDs, and Virtualization, pp. 165-169, 2015.

[13] LxC project: http://linuxcontainers.org

[14] B.Kirsch, *What to choose from the top orchestration software on the market*, http://searchitoperations.techtarget.com/feature/What-to-choose-from-the-top-orchestration-software-on-the-market