Securing Linux Containers

GIAC (GCUX) Gold Certification

Author: Major Hayden, major@mhtx.net

Advisor: Richard Carbone

Accepted: July 26, 2015

Abstract

The components that make Linux containers possible have been available for several years, but recent projects, such as LXC and Docker, have made the technology much more accessible to users. Containers allow for even more efficient utilization of server resources through greater density and faster provisioning. However, securing containers is much more challenging than traditional virtualization methods, including KVM. The isolation layer between the container and the kernel, as well as between each container, is extremely thin. Weaknesses in the kernel or the container configuration can lead to compromises of containers or the entire system. The responsibility of managing the operating system within the container can also become blurry with time, and that can also lead to compromises of the container. Fortunately, Linux security modules, such as SELinux and AppArmor, along with careful configuration and container operating system management, can strengthen the thin walls around each container. Organizations that use mature Dev/Ops practices can also improve security within each container by automating the creation and deployment of container images. This paper will discuss the best strategies for securing a system running containers and the trade-offs that come with each.

1. Introduction

Linux containers provide a virtualized environment for processes on Linux servers with less overhead than virtual machines. System administrators can deploy containers quickly while using fewer server resources.

Multiple technologies come together to make containers possible on Linux, but the majority of the work is centered on a concept called namespace isolation. Namespace isolation allows a server to isolate a process so that it cannot see certain portions of the overall system. For example, process ID (PID) isolation can be used to make a process think that it is the only process running on the server. It would have no access to know that other processes exist, and it would not have the ability to send signals to any of those processes. In addition, a container could run its own *init* process as PID 1 while the host system sees that process as an entirely different PID. (Kerrisk, 2013)

While namespaces provide some security for containers, many people argue that namespaces do not do enough to truly *contain* a container. Namespaces do not cover all aspects of a Linux system the way a KVM virtual machine does. In addition, processes within a container still have some level of access to certain kernel-based filesystems, such as *procfs* and *sysfs*. Kernel capabilities reduce the amount of things that can be done within a container, even by the root user. Combining namespaces and kernel capabilities definitely improves container security, but it is still not enough. (Walsh, 2015)

Mandatory Access Control (MAC) is a key technology for securing containers. The most commonly used MAC technologies for Linux are SELinux and AppArmor. Both are implemented using the kernel's Linux Security Modules (LSM) framework. SELinux provides policy-based security controls that define what processes are able to do on a system regardless of any discretionary access controls (DAC) in use. For example, a directory may be configured with the correct permissions for a particular daemon to create files within it, but a SELinux policy can specify that the daemon not be permitted to write files there. The MAC policy would override the DAC filesystem permissions and the writes would be denied. (Wikipedia)

Control groups, more commonly known as cgroups, provide system administrators with tools that limit resource usage for processes. These limits apply to containers as well since the kernel sees them as processes. Limiters are available for various resources, including CPU usage, disk I/O, network throughput, and memory consumption. A control group contains multiple resource limits, and then multiple processes can be added to the control group. All of those processes will share the resources allocated to the control group. (Menage, n.d.)

Tying all these technologies together to deploy containers involves a new way of thinking about infrastructure. Many software projects, including LXC and Docker, make container deployment easier through simple command line tools and integrations with existing virtualization management platforms. Other projects that assist with deploying large container environments are Kubernetes and Mesos.

Deploying containers also requires a new way of thinking about security. The isolation layer between containers and the host system is extremely thin. This requires careful thought about people, process, and technology. The efficiency gains from running containers can disappear quickly without appropriate security controls and processes.

2. About Linux containers

A container is a way to isolate Linux applications with very little overhead. All containers share the same host kernel, but they have their own virtualized network adapters and filesystems. Containers allow for efficient application deployment and management, but they are not preferred for all workloads. Depending on the application, there are various compatibility, performance, and security limitations that come with containers.

2.1 Server management strategies

System administrators have several methods for hosting applications on a server. Selecting the best one depends on what is best for the application and the business. For example, a business operating in a high-risk environment might choose a hosting strategy that favors security or isolation over performance. Another business might place a higher value on performance and scalability.

2.1.1 Multiple servers with one application on each server

The simplest method for hosting applications is to host one application per server. A simple deployment may have a physical database server connected to a physical application server that resides in a network DMZ. This solution has the best security strategy.

If an attacker compromises the application server, they must find a way to break into the database server through the limited access provided through the firewall. SELinux can be used to enforce MAC policies and help reduce the impact of any potential vulnerability on the server.

Hosting one application on each server also provides the best performance since applications would not be in conflict with one another for resources.

Unfortunately, the management costs for this type of deployment can be higher than the other strategies. A large capital expenditure is required, and system administrators must maintain multiple separate servers along with network devices. Highly available solutions will require additional servers and network devices since replacing a device involves downtime and physical intervention.

2.1.2 Multiple servers with multiple applications on each server

Hosting multiple applications on a single server reduces cost but leads to greater security challenges. An example of this deployment type is running Linux, Apache, MySQL, and PHP (LAMP) on a single server. This is often the case for various PHP blogging and content management-based system projects, such as WordPress or Drupal.

If this server is compromised, there is a good chance that the attacker will have access to many parts of the system if not the entire system. Using SELinux is strongly advised since it provides a level of isolation between sensitive processes, including Apache and MySQL, and could limit the extent of a compromise. However, if an attacker finds a usable kernel vulnerability, they can gain access to the entire system.

An example of a SELinux policy mitigating a vulnerability on a single system with multiple applications running is referenced in CVE-2007-3304. The vulnerability allows an attacker to modify some internal arrays in Apache that could cause the daemon to send kill signals to other processes on the system. SELinux policies denied access to those processes, and the vulnerability could not be exploited on systems with SELinux in "enforcing" mode with the most recent targeted security policy updates. (RHSA-2007:0556-2, 2007)

Performance can vary when multiple applications run on a single server. The "noisy neighbor" effect of additional processes on the system makes it more difficult for system administrators to hunt down performance bottlenecks, but there are tools[1] available to help with this. It also creates unpredictable response times for latency-sensitive applications. Real-time kernels can alleviate some of the problems with jitter and latency, but using them involves some large, system-wide changes. (Williams, 2008)

2.1.3 Multiple servers with virtual machines, one application per VM

Deploying with virtual machines provides greater isolation between applications without significantly raising costs. Replacing a virtual machine is much easier than replacing physical machines, and it takes much less time. In one phase of a virtualization implementation at Southwestern Illinois College, the cost of ownership dropped 50% over three years. Additional savings came from reduced datacenter space usage, utility expenses, and hardware administration. (Leja, 2010)

---

1 The simplest tools, such as htop and top, will show applications consuming a significant amount of CPU time or memory. Network congestion is easily diagnosed with aggregate tools, including bwm-ng and iptraf-ng, and can be further refined with iftop and tcpdump. The sysstat scripts record fine-grained resource usage statistics over time and allow administrators to tackle performance problems before they become a serious issue.

KVM provides strong isolation between virtual machines mainly through its default use of MAC policies. All KVM virtual machines run as a process and those processes are confined using SELinux policies. In addition, KVM has an API for sVirt that allows for an even more granular application of policies. sVirt ensures that all processes related to a single virtual machine can only manipulate files and devices associated with that virtual machine. If an attacker breaks out of a KVM virtual machine, they can only change files that the virtual machine has access to change via SELinux policies. (IBM, 2011)

VENOM (CVE-2015-3456) was the codename for a serious vulnerability that appeared in 2015 on systems running KVM with Qemu. A vulnerability in the floppy disk controller allowed attackers to write to memory that would cause a guest crash in the best case and execution of arbitrary code on the host in the worst case. The floppy disk controller is rarely used in modern systems but it is enabled by default in most Qemu implementations. Since the vulnerability affected a virtualized floppy disk controller, it did not matter if the physical system had an actually floppy disk drive or controller. SELinux and sVirt prevented attackers from gaining access to other parts of the system by restricting what the Qemu process was able to do on the host. (Walsh, 2015)

Server hardware also provides additional security through technologies in the CPU. Intel's VMX and AMD's SVM are examples of hardware-assisted virtualization technologies that are found in desktop and server CPU's. VMX allows a system to execute root and non-root operations in the CPU. Typical root operations would come from within the host system while non-root operations come from the guest. This allows KVM to avoid switching between rings[2] constantly in software that increases performance and security. The CPU limits which operations can be called in non-root mode. These restrictions keep root users in a virtual machine from running as root on the host system itself. (Intel, 2011)

---

2 Kernel rings are arranged from most privileged to least. As an example, the kernel typically runs in ring zero while most applications run in ring three. If an application in ring three wants to perform an operation with escalated privileges, it must pass through special "gates." Special applications, like *init*, can run in both rings at the same time.

Virtual machines also support complex network configurations. System administrators can attach virtual machines to different VLANs or apply bridge filtering controls so that virtual machines can only communicate with certain hosts. Stateful firewall rules can be placed between virtual machines and the external network or between individual virtual machines themselves. (IBM, 2011)

2.1.4 Multiple servers with containers, one application per container

Containers provide a lighter-weight alternative to virtual machines. Processes within containers run within a namespace and control group. This ensures isolation from the rest of the system and places limits on resource usage with less overhead than virtual machines. However, containers utilize the existing Linux kernel on the host system, and they boot directly into an *init* system, such as *systemd*. This allows system administrators to fully bring functional containers online, often in less than a few seconds.

Security isolation for containers relies upon namespaces, control groups, and MAC policies. This differs from KVM in that there is no hardware assistance involved and processes within containers can talk directly with the host system kernel. This thinner layer of isolation provides additional performance and lower resource usage since there is no need for an entire operating system to be running in each container. (Walsh, 2013)

Properly configured containers have a security profile that is slightly more secure than multiple applications on a single server and slightly less secure than KVM virtual machines. As with multiple applications on one server, a compromise of one container could lead to compromise of the entire system. For that attack to be successful, the attacker would need to subvert SELinux policies (with MCS separation[3]) and exploit an existing kernel vulnerability to subvert namespace isolation. (Walsh, 2013)

Containers also support the complex networking configurations found in KVM, including bridge filtering, stateful firewalling, and VLANs. Network namespaces ensure that each container has the exclusive use of a virtual network device.

---

3 Multi-category separation, or MCS, provides a way to make more specific sub-policies or categories that apply tighter restrictions to what a process can do with certain files and objects. This is explained in detail in Section 3.4.

2.2 Container frameworks

Assembling all of the virtualization building blocks for virtual machines and containers is challenging. Many container frameworks are available as open source projects, and that makes it much easier to manage the various namespaces, cgroups and security controls. These frameworks include LXC, Docker, *systemd-nspawn*, and Rocket.

One of the first projects created for managing containers is LXC. It contains libraries and command line tools that help system administrators quickly create containers without needing to be experts in the underlying technology. LXC's command line tools can create containers with all available namespaces, cgroups, and SELinux policies set. Users can easily add and remove certain kernel capabilities or adjust cgroups depending on the needs of the application. Libvirt can also manage containers using an LXC library, and it will use MCS separation via the sVirt API for additional security. (Graber, 2014)

Docker is another container management project, and it aims to make container deployment more automated. It provides an API for remote management, and it automatically configures networks when containers are built. Docker utilizes a layered image format that allows users to add their application and its dependencies onto an existing trusted image. As with LXC, Docker configures namespaces, cgroups, and SELinux by default.

Servers running *systemd* can use built-in functionality to launch containers with very little configuration. The *systemd-nspawn* tool will start a container with a single daemon or a full *init* system. It will also configure all of the underlying container isolation technologies by default. (Edge, 2013)

Rocket was introduced in late 2014 as a competitor to Docker. It utilizes *systemd* for launching containers and it has an image management mechanism. The developers of Rocket are also working to standardize a container specification with a goal of providing greater security and compatibility. (Polvi, 2014)

2.3 Container infrastructure management

The container management frameworks from the previous section are helpful for managing smaller container deployments, but many businesses are eager to manage large fleets of container-based infrastructures. As the number of containers per server increases, additional strain is placed on the network and the system administrators that manage all of it.

Google conceived the idea of Kubernetes after managing its own large container infrastructure for many years. It has three main concepts: pods, replication controllers, and services. Pods may consist of a single container or multiple containers. Replication controllers ensure that a certain number of containers are running within pods. Finally, services help make the link between a "service," which could be a website, and the pods that are running containers that contain the website content. Networking with Kubernetes can become quite complex, and another project, Flannel, is under active development to make this process easier. (Paris, n.d.)

3. Securing the container host system

Creating separation and isolation is the key to securing any system since it limits the impact caused by a compromise. Working with a container system is no different. Since processes in containers are just processes on the host, a defense in depth strategy is mandatory.

For example, if a user is root (UID 0) in a container, they are root on the host. If they find a way to escape the confines of the container, they will have *full root access* to the underlying host. This can quickly lead to compromise of other containers on the host and other physical systems on the same network segment.

3.1 Discretionary and Mandatory Access Control

Securing containers requires a deep understanding of Discretionary Access Control (DAC) and Mandatory Access Control (MAC).

As the name implies, DAC policies can be overridden depending on the user running a process. The kernel can use its *discretion* to determine if access should be allowed. Filesystem permissions are a good example of DAC. If a user owns a particular file and they set the mode of the file to 0600, then no other users can read or write to that file. However, the root user could venture into that directory and manipulate the file at any time. The kernel used its discretion to say that another user could not access the file (based on filesystem ownership and permissions), but the root user could access the file because of its superuser privileges. There is an exception: if a user marks a file as immutable, even root cannot manipulate the file until the immutable attribute is removed.

DAC provides good security, and it is very easy to use. Users can list files in a directory and quickly understand which ones they are able to access. In situations where a process may be controlled by an attacker, DAC simply is not enough.

Filesystem access control lists, or ACLs, are helpful as they allow administrators to set more specific DAC policies for accessing files. Once the filesystem is mounted with ACL support, administrators can apply ACL policies to directories and files. For example, a file might be owned by one user with filesystem permissions that only allow that user to edit the file. An administrator could set an ACL that allowed a second user to edit the file without changing the filesystem permissions. This helps avoid using wide-open Linux permissions, such as 0777, in situations where an administrator wants to allow access for one additional user.

In contrast, MAC policies are called mandatory because they cannot be overridden unless the policies are changed or disabled completely. These policies are not as visible on the system and can be difficult to troubleshoot. As an example, an Apache server may have the correct filesystem permissions (DAC) to write to a particular directory but the access is denied due to a MAC policy. This creates a frustrating situation for many system administrators because the reason for the denial is not entirely obvious. Fortunately, all of the denied access is logged in the kernel audit logs. Administrators can install *setroubleshoot* to see simpler explanations of the denials and multiple options on how to allow the access.

SELinux and AppArmor are the two most common implementations of MAC for the Linux kernel. Both have policies that define what a particular process can do on a Linux system but they differ greatly in their implementation. One common element they have is that they work well with libvirt's sVirt API and provide strong levels of separation for containers. They also feed their information about denials into *auditd* for collection and review. Both of them have "learning" modes that log potential denials without actually blocking access.

SELinux is a labeling system where everything receives a label, including processes, files, directories, and other objects. It comes with various policy sets that define how a process with one label can interact with something else that has another label. The kernel enforces those policies. The configuration is extremely granular and can be a challenge to adjust. Fortunately, many common adjustments are done via Booleans that can be quickly toggled on and off. For example, if an Apache server needs to talk to a remote database server, toggling a single Boolean enables that access. (Walsh, 2014)

AppArmor does not apply labels but instead relies on policy files that specify file paths to protect. Policy files contain a reference to a particular executable and what that executable is allowed to do. For example, an application could be limited to certain kernel capabilities or gain permissions to send raw network packets across the network. The configuration and policies are less granular but easier to integrate into existing systems. This is similar to Trusted Solaris. (Wikipedia)

Systems running Red Hat-based distributions, including CentOS and Fedora, will have SELinux available and in its enforcing state by default. Debian-based systems, including Ubuntu, will have AppArmor available but the default enforcement setting differs between versions.

3.2 Kernel updates

One of the biggest weaknesses of any container system is a kernel vulnerability. Since processes in containers are really just isolated processes on the host, any kernel vulnerabilities that allow an attacker to break out of the namespace isolation can be disastrous. Container systems should receive regular kernel updates whenever they are made available by the upstream Linux project or by the Linux distribution currently in use.

Updating a kernel is a two-phase process. The first step is to update the kernel code itself and the second step is a reboot. A new project, Ksplice, has recently been merged into upstream Linux, and its goal is to enable kernel updates without requiring a reboot. This involves carefully compiling a patch and using client tools to load the patched kernel code into the running kernel. (Poimboeuf & Jennings, 2014)

Some system administrators may want to custom compile their kernel to remove certain unneeded features or add experimental functionality. These users may want to consider some of the advanced functionality included in the Grsecurity and PaX kernel patch sets. They can add additional protection for containers as well as other processes on the system that run outside of containers. However, as with MAC, it can become difficult to troubleshoot why a particular application is denied access certain elements of the system.

3.3 Kernel capabilities

The first set of Linux capabilities appeared in the kernel in the late days of kernel 2.14, and their goal was to split up root privileges into multiple pieces. At the time, some applications were run with setuid root -- which allows a regular user to run an application with root-level privileges. This was required for some applications, such as *ping*, that required raw access to the network. (Bacarella, 2002)

These capabilities allowed normal users to run certain applications with restricted root privileges. In the *ping* example, the executable receives the CAP_NET_RAW capability. That allows it to use raw network sockets without gaining any additional privileges. This is extremely helpful for executables with a vulnerability such as a buffer overflow. In the old model, where executables similar to *ping* were setuid root, a vulnerability in *ping* could allow a normal user to execute arbitrary code as root on the system. (Bacarella, 2002)

Containers also have capabilities applied to them when they start. Frameworks, such as LXC and Docker, choose a minimal set of capabilities to start with and system administrators can choose to add or remove capabilities from the default setting if needed. If a container needs to create a special file with *mknod*, a system administrator might add the CAP_MKNOD capability to the container. That capability remains with the container as long as it is running. (Bacarella, 2002)

3.4 sVirt and MCS separation

sVirt adds powerful separation to a virtualized system in conjunction with MAC implementations such as SELinux and AppArmor.

In the case of SELinux, custom policies are generated per container. This is called Multi-Category Security (MCS). SELinux policies already exist that define what a container's processes can do on a system, but MCS takes it a step further and dynamically creates an additional sub-policy (called a category) that is specific to one container. MCS applies sub-policies, called categories that limit the access from a specific container to specific files or objects. (Walsh, 2009)

This is challenging to understand without a simpler example, and the *SELinux coloring book* provides a great, if not humorous example. Consider a dog and a cat as well as their respective bowls of food. The dog should eat the dog food and the cat should eat the cat food. If the dog tries to eat the cat food, the dog must be denied access. This is how SELinux policies work.

However, what if there are two dogs: a Dachshund and a Great Dane? They will both have dog food in their bowls but they will likely receive different quantities or types of dog food. How do we deny them access to the wrong food when both dogs are labeled as dogs and all of the food is dog food? This is where MCS separation comes into play. We can label the Dachshund and its food as *dog:dachshund* and *dog_food:dachshund*. If the Great Dane is labeled as *dog:great_dane*, it would be denied access to *dog_food:dachshund* because the sub-policy, or category, does not match even though it is still dog food. (Duffy & Walsh, n.d.)

MCS separation ensures that a particular container is only allowed to access the resources that are assigned to it. The sVirt functionality within libvirt ensures that the appropriate labels and categories are applied to the container, its processes, and its system resources as soon as it starts. If container A is compromised, it cannot manipulate the host's resources due to standard SELinux policies. It also cannot manipulate container B's resources due to MCS separation.

3.5 User namespacing

The newest namespace in the Linux kernel deals with the isolation of UIDs. The goal is to ensure that a root user in the container is not actually equivalent to root on the host. This is done through a UID mapping and administrators can choose which UIDs in the container map to a particular UID on the host. As an example, a root user within a container might see themselves as UID 0 while on the host a system administrator sees that root user as UID 31337. These mappings should be done carefully to avoid overlaps and management headaches on highly dense systems.

In the case of a container compromise, this would deny root access on the host to the root user within the container. The attacker would need to find a new method for gaining escalated privileges on the host system. (Kerrisk, 2013)

3.6 Secure Computing Mode (seccomp)

Another option to provide additional security on container systems is to limit the syscalls, or system calls, that processes inside a container can make. The seccomp project allows limits to be placed on which syscalls can be made by those processes. When a process makes a call that is not allowed, the default action is to kill the process. There are user-configurable options that send more friendly signals to the process instead.

Creating a set of allowed syscalls for an application is challenging for two reasons: there are a large set of syscalls available, and it can be difficult to determine which syscalls an application will make when it runs. As of Linux 4.1, the Linux kernel has 378 syscalls, and the list continues to grow. If system administrators do not have an accurate list of the syscalls a particular application might make, they may cause the application to throw errors or crash.

System administrators can use *strace* or *ptrace* to profile an application and get a list of syscalls that are made during normal operation. The accuracy of the list largely depends on the workloads being handled by the application. Users should profile their applications while they are handling regular workloads to get the most complete list of required syscalls. (Edge, 2012)

4. Security within Linux containers

Securing the container host is critical but is only half the battle. The data inside the container, including the application, configuration files, and the operating system, is just as critical to the overall security of the environment.

4.1 Trusted images

All containers start with software that needs to run inside the container. Some containers, such as the ones started with *systemd-nspawn*, can simply re-purpose an existing executable from the host system and launch it in a container. Other containers are built using trusted packages from the host system's distribution with package management tools including *debootstrap*, *yum*, or *dnf*. This allows administrators to install the base OS from trusted sources and cryptographically verify each package.

Docker takes a different approach. Users can simply download an existing image from Docker's public facing index using the *docker pull* command and apply customization on top of it. It is as simple as pulling a CentOS 7 image, writing a short file of customization instructions (called a Dockerfile), and telling Docker to build the container. Docker stores the results of customizations, such as adding new packages or configuration files, as layers and applies those layers onto the original base image.

This simplicity leads to security issues for administrators. Docker has implemented some initial checks for images when they are downloaded to ensure they were not altered during transit, but they do not vouch for the actual content found within the images themselves. It is up to the user to determine which images are trustworthy for their environment. A recent automated study of images available in the public Docker showed that 30% of images contained serious security vulnerabilities. (Gummaraju, Desikan & Turner, 2015)

It is possible for a determined attacker to place an intentional vulnerability or backdoor into in image, and it could go undetected. If the image is used widely in container deployments, it could lead to serious compromises in user data or denial of service. Anyone who builds a container must be able to verify its source or they must build their own containers using trusted tools and packages.

4.2 Container operating system updates

Unlike virtual machines, containers do not provide simple methods for applying operating system updates. Containers are generally considered immutable once they are built, and any updates to the container are often done by building a new container. Docker makes this process easier since the user can simply rebuild the container image with the same Dockerfile and get the latest operating system updates, so long as the Dockerfile contains commands for updating the operating system.

Deploying the new container with the updated images to replace the old container can occur via a number of methods. For one-off or small container deployments, the old container could be stopped and the new container could be started in its place. Downtime is minimal due to the short time required to stop and start containers.

A more automated approach would be to start a new container, or group of new containers, alongside the old containers. Monitoring systems would check the new containers to verify that they are responding properly, and then a quick load balancer change would shift traffic over to the new containers. Kubernetes has features that allow new pods to start alongside old ones, and then the service can be adjusted to point to the new containers.

4.3 Communication between containers

Host protections, such as namespaces and MAC policies, provide strong protection between containers on the host, but it is important to consider how containers can communicate with each other and with the host outside of those protections.

Containers with unfiltered network access can communicate with each other and the host if they are on the same network segment. Since containers have their own virtual network interface (thanks to network namespaces), users can attach the network interface to a variety of network devices.

The simplest solution could be to use a Linux bridge and place all of the containers on the bridge. Communication is simple and fast, but not secure. A stronger solution would be to place containers on bridges with filtering applied or use VLANs to carve up new network segments. Modern systems may be able to use virtual network switching via OpenvSwitch on the host and achieve greater network separation.

It is also important to consider other objects that may be shared between containers, including sockets or shared storage devices. System administrators should carefully consider the consequences of sharing objects between containers to limit the spread of a compromise.

4.4 Security responsibility

Developers appreciate containers because they can package their application, test it alongside its libraries, and verify that it will work in production. Operations teams appreciate containers because they get the applications in a cohesive package along with their dependencies and configurations. However, who owns the security of the container operating system, configuration files, and the application in this new world of containers?

The responsibility of securing the operating system normally falls onto the operations team. However, if developers are writing applications and building a container with their application in it, how do operations teams ensure that the base operating system is secure?

This is where frameworks with layered images, including Docker, can help. Operations teams can carefully maintain a base image with appropriate security controls, configurations, and package updates. As part of that configuration, they can specify where the package manager will receive trusted packages. Development teams can use that base image as the foundation for their containers and then add packages from those trusted repositories. If a serious vulnerability appears, the operations team would quickly update the base image and let the development team know that a container rebuild and redeployment is needed. (Walsh, 2015)

5. Guide: Implementing a secure container

There are many different methods for implementing secure containers, and they vary according to the complexity and size of the deployment. This guide takes a simple approach and uses only free and open source software. At the end of the guide, the reader will have built a secured container running on a Linux host using libvirt's LXC driver.

5.1 Requirements

This example will use a CentOS 7 system with a small package set. The reader will also need a system on which to run the CentOS operating system. Any compatible hardware, virtual machine, or remotely hosted cloud instance will be sufficient so long as it has 10GB of disk space and at least 512MB of RAM.

5.2 Installation

Install CentOS with a minimal package set. There is no need for a graphical interface for this guide. Once the system is installed, verify that SELinux is in the default "enforcing"mode. Run the `getenforce` command to verify the SELinux status. If the command returns `Permissive`, run `setenforce 1` to change to "enforcing" mode.

Update all packages and then install the packages needed for container management:

```
# yum -y upgrade
# yum -y install libvirt virt-install
```

In addition, ensure that the libvirt daemon is running:

```
# systemctl start libvirtd
```

5.3 Bootstrapping the container

The container needs an operating system, and it should be installed using the trusted package manager and packages. This command will install a small CentOS 7 distribution into a directory where libvirt can manage it:

```
# yum -y installroot=/var/lib/libvirt/filesystems/centos7 \
      --releasever=7 install systemd passwd yum \
      centos-release vim-minimal procps-ng iproute \
      net-tools dhclient policycoreutils
```

Our container operating system is now installed and ready to be configured. A password must be set for the root user:

```
# chroot /var/lib/libvirt/filesystems/centos7/bin/passwd root
```

Also, console access must be allowed:

```
# echo "pts/0" >> \
      /var/lib/libvirt/filesystems/centos7/etc/securetty
```

5.4 Launching the container

Before launching the container, libvirt needs to know that it exists:

```
# virt-install --connect lxc:// \
      --name centos7 --ram 256 \
      --filesystem /var/lib/libvirt/filesystems/centos7,/
```

In a few seconds, the container should fully boot and stop at a login prompt:

```
CentOS Linux 7 (Core)
Kernel 3.10.0-229.4.2.el7.x86_64 on an x86_64
```

```
containers login:
```

The root password set in the earlier section should work at the prompt.

5.5 Inspecting the container

After logging in, running `getenforce` should show `Disabled`. However, SELinux is running on the host and is protecting the container. To verify, hold the CTRL key and press "]" to escape the container's console. The container will still be running, but the current console will switch back to the host system.

Review the process list to verify the SELinux labels:

```
# ps efxZ | grep libvirt_lxc | awk '{print $1}'
system_u:system_r:virtd_lxc_t:s0-s0:c0.c1023
```

The context applied to each process of the container is `virtd_lxc_t`. SELinux has policies that determine what a process running with this context on the system can do. However, the additional information in the process listing is the category and is used for MCS. The `s0-s0:c0.c1023` string is the category label applied to the process. If additional containers are running on the host, different MCS labels would be used for each container.

A virtual network device should also appear on the host:

```
# ip link show vnet0
```

This virtual network device was created automatically by libvirt when the container was instantiated with `virt-install`.

In addition, PID namespacing can be verified by inspecting the container's *init* process from the host's perspective:

```
# ps aufx | grep init
root      1638  0.0  0.3  53584  3432 ?        Ss   03:53
0:00  \_ /sbin/init
```

The container thinks that `/sbin/init` is running as PID 1, but the host sees it as PID 1638.

6. Conclusion

Containers give businesses the opportunity to reduce resource usage, manual deployment work, and downtime. They also force businesses to be more nimble and make their actions more repeatable. Containers have changed the game of how development and operations teams work together by creating a deployment mechanism that pulls together the strengths of both teams. Developers have the opportunity to see their applications in production running as they intended, and operations teams gain the redundancy and reliability they need.

Businesses must evaluate if their applications are currently compatible with container technology and if the application is built to scale out to multiple containers. Monolithic applications that require local state are probably not good candidates for containers, but many other applications are. Business processes must also be mature and nimble enough to make containers a reality. The technology is growing stronger every day, and each business must consider if their people and processes have grown in equal amounts.

However, as with any new technology, containers present challenges to our existing security strategy. Traditional defense in depth strategies still work at a high level, but the low-level touch points must be revised. Upstream work in the Linux kernel and various container frameworks continues to push the boundaries of performance and security. As more standardization and automation evolves, securing containers will become a more straightforward process.

Some security technology in container environments is non-negotiable. Using Mandatory Access Control implementations, such as SELinux or AppArmor, must be a first step for securing all systems running containers. Users must carefully consider all available avenues for communication between the containers and their host, as well as between the containers themselves.

Security within the containers is often forgotten, but it is just as critical as securing the host. Users must know and trust the source of their container images, their application, and its dependencies. Operations and development teams must communicate about security responsibility, and every piece of the container should have an owner from the time it is built to the time when it is finally decommissioned.

7. References

AppArmor - Wikipedia, the free encyclopedia. (n.d.). Retrieved July 8, 2015, from https://en.wikipedia.org/wiki/AppArmor

Bacarella, M. (2002, May 1). Taking advantage of linux capabilities. *Linux Journal*, (97).

Duffy, M., & Walsh, D. (n.d.). *SELinux coloring book*. Retrieved July 8, 2015, from https://people.redhat.com/duffy/selinux/selinux-coloring-book_A4-Stapled.pdf

Edge, J. (2012, April 15). A library for seccomp filters. Retrieved from https://lwn.net/Articles/494252/

Edge, J. (2013, November 7). Creating containers with systemd-nspawn. Retrieved from https://lwn.net/Articles/572957/

Gummaraju, G., Desikan, T., & Turner, Y. (2015, May 26). Over 30% of official images in Docker hub contain high priority security vulnerabilities. Retrieved from http://www.banyanops.com/blog/analyzing-docker-hub/

Graber, S. (2014, January 1). LXC 1.0: Security features. Retrieved from https://www.stgraber.org/2014/01/01/lxc-1-0-security-features/

IBM. (2011, November). KVM: Hypervisor security you can depend on. Retrieved from ftp://public.dhe.ibm.com/linux/pdfs/LXW03004-USEN-00.pdf

Intel. (2011, May). Intel 64 and IA-32 architectures software developer's manual. Retrieved from http://www.intel.com/Assets/en_US/PDF/manual/253669.pdf

Kerrisk, M. (2013). Namespaces in operation, part 1: namespaces overview [LWN.net]. (n.d.). Retrieved from https://lwn.net/Articles/531114/

Kerrisk, M. (2013, February 27). Namespaces in operation, part 5: User namespaces. Retrieved from https://lwn.net/Articles/532593/

Leja, C. (2010, January 15). Implementing server virtualization at Southwestern Illinois College. Retrieved from https://c.ymcdn.com/sites/www.aitp.org/resource/resmgr/research/swic-server-virtualization-c.pdf

Menage, P. (n.d.). Kernel documentation: cgroups. Retrieved July 6, 2015, from https://www.kernel.org/doc/Documentation/cgroups/cgroups.txt

Paris, E. (n.d.). Kubernetes: What's it do? Retrieved from http://people.redhat.com/~eparis/kubernetes/kube.pdf

Poimboeuf, J., & Jennings, S. (2014, February 26). Introducing kpatch: Dynamic kernel patching. Retrieved from http://rhelblog.redhat.com/2014/02/26/kpatch/

Polvi, A. (2014, December 1). CoreOS is building a container runtime, rkt. Retrieved from https://coreos.com/blog/rocket/

RHSA-2007:0556-2: Moderate: httpd security update. (2007, June 6). Retrieved from https://rhn.redhat.com/errata/RHSA-2007-0556.html

Security-Enhanced Linux - Wikipedia, the free encyclopedia. (n.d.). Retrieved July 6, 2015, from https://en.wikipedia.org/wiki/Security-Enhanced_Linux

Walsh, D. (2009). Secure virtualization using SELinux. Retrieved from https://fedorapeople.org/~dwalsh/SELinux/Presentations/svirt.pdf

Walsh, D. (2013). USENIX: Secure linux containers. Retrieved July 6, 2015, from https://www.usenix.org/conference/lisa13/secure-linux-containers

Walsh, D. (2014). SELinux (presentation). Retrieved from https://dwalsh.fedorapeople.org/Presentations/SELinux/

Walsh, D. (2015, May 19). Is SELinux good anti-venom? Retrieved July 6, 2015, from http://danwalsh.livejournal.com/71489.html

Williams, C. (2008, June 18). An overview of realtime linux. Retrieved July 6, 2015, from http://people.redhat.com/bche/presentations/realtime-linux-summit08.pdf