ch #6 Workload Considerations [part one]

LFS 260

Objectives

- Use static and dynamic container scanning tools.
- Apply AppArmor profiles to further constrain containers.
- Learn about SELinux implementation.
- Discuss how the Falco project can improve cluster security.
- Discuss immutable containers.

Before You Download Image

The best place to begin security is before software has ever been accessed. Only allowing trusted repositories is the first step. Even then, utilize keys and hashes to make sure the package has not been modified after the original programmer finished their work.

If you can, also encourage developers to provide small microservices, using as small a base image as possible to lower the potential complexity and better analyze what issues the microservice could contain. Even with these steps, ensure that you are not adding any software to the cluster except that which has cleared static and dynamic scanning, as we will learn more about in this chapter.

Before Running a Container

It would be better to know of an issue with code prior to running it. This is called Static Analysis, which is used prior to Dynamic Analysis; checking how software actually runs in a protected, or sandbox environment.

Images could be downloaded and analyzed, or scanned, via **static** and **dynamic** analysis, then added to the local repository. Another way to scan images would be through the use of a Dynamic Admission Controller. You would first set up a tool like Sysdig or Aqua, and then add a ValidatingWebhookConfiguration, which uses a webhook whenever a CREATE or UPDATE call is made.

A static analysis of a container, or the source code used to build the container, looks through the code for statements, declarations, and possibly errors in the code. More complex scanners may be able to analyze the entire source code and all the possible actions it may attempt, though the complexity of the scan may be imperfect. Further analysis can be performed after the source code has been compiled and is object code

There are several tools to choose from to scan and analyze images, such as Docker Bench, Clair, Trivy, etc

Docker Bench

Docker has been maintaining Docker Bench, a tool similar to kube-bench, which could be run to investigate images without needing to be part of a Kubernetes cluster. This tool is called docker-bench-security. The output and checks are similar, without the suggested fixes to each issue being included. As a POSIX-compliant tool, you should be able to run the **docker-bench- security** command on Linux distributions, as well as MacOS.

The Docker Bench tool is now a few releases behind CIS and has not been updated since November 2019, when Mirantis took over Docker and promised to continue the open source projects. It may be updated and current at some point in the future.

Clair

Another tool to analyze images in a static manner is Clair, which was developed by Quay.io, now a company owned by Red Hat. The tool is divided into two parts and has three functional divisions.

One part of Clair is considered a service wrapper and provides an HTTP Interface, a Notifier, and Notification Storage. The other part is referred to as the ClairCore, and handles downloading vulnerability information and comparing that against an index of existing images.

- The first phase is **indexing**, which begins with a manifest submitted to Clair. Clair uses this information to download the image layers, scan them, and generate an *IndexReport*.
- The second phase is **matching**, when an *IndexReport* is compared against known vulnerabilities. These are downloaded on a regular basis. Be aware that after starting Clair you may have to wait for the download before being able to match.
- The third phase is when a **vulnerability** is found within an *IndexReport*. Depending on the configuration of the notification, the notifier will take action.

Trivy

While Clair uses alpine-secdb (which covers back-ported fixes), it is not complete. In fact, it may have half as many issues as are currently discovered.

Trivy is a simple and comprehensive vulnerability scanner for containers. Each time trivy is run, it retrieves a more complete *vuln-list* to analyze. As this list is downloaded from Alpine Linux, it is most complete when analyzing Alpine and RHEL/CentOS.

In a large environment, you may want to set up your own server of the vuln-list, then use trivy in client mode and pass the address and port of the server. This will not download the database, but reference the common one on the server. This also helps in an air-gapped environment where you can download the list on an external system, check the contents, and manually install the list on a protected server.

Another reason Trivy may be more accurate at finding issues is it checks the middle layers of an image to find the version of the static linking library. Several continuous integration tools (CI) have easy-to-use integration files to use Trivy in a CI/CD pipeline.

Dynamic Analysis

While static analysis is a necessary first step, there is always the chance that something was missed. As the code will be running, and perhaps compromising the system, there is greater risk to dynamic analysis. If the images are from trusted repositories, and static analysis has not found any issues, it is still important to analyze the program as it runs. This task would typically be done on a development or non-essential cluster, not in production.

The **perf** and **ftrace** Linux commands are helpful in tracing and profiling a process as it runs. The Tracee program can be used to perform a similar trace, using eBPF programs to observe system calls and events. It can also be used to view the use of memory by a process, and extracting binaries to fully understand what a program is working with.

The Falco project is also useful for analyzing containers as they run, among other tasks. More will be covered on the following pages. Falco was created by Sysdig, which shares much of their libraries with Falco. Sysdig licenses software for monitoring and scanning, among other features.

Use Admission Controller

There are several places one can cause scanning of a workload to take place. The optimal situation is when scanning is done prior to the container or software being allowed into the production environment. In more agile CI/CD environments, this may not be possible. One way to handle this is by leveraging dynamic admission controllers. Every API call to create an image would then reference an outside tool in order to scan and approve an image prior to that image being accepted.

While this setup is beneficial for checking all images, it also adds a lot of latency and resource usage to the environment. Every time the same, previously-cleared, image is requested again, it will be checked. The amount of latency depends on access to the external source and the size and complexity of the image.

A common option is to leverage a dynamic admission controller to insert an *initContainer*: containing a scan or verification tool into the pod spec. Then the load is distributed across all the workers. Only if the initContainer: scans and has a zero exit code, is the rest of the pod spec passed to the container engine for execution. This is the way that TrendMicro offers some of their cloud security tools.

You can read more in the Kubernetes documentation, on the Using Admission Controllers page.

Tracee

Tracee is a tool which allows for real-time monitoring of system calls and kernel events. While all actions will be traced, you could grep through the output to narrow it down to a particular pod. The information shown has a precise time stamp, uts_name, UID, PID, return code, event, and arguments.

In order for tracee to run, you will need to provide at least three volume locations with the -v option for the kernel information to be pulled, such as **/lib/modules/** and **/usr/src**, as well as ephemeral information such as **/tmp/tracee**.

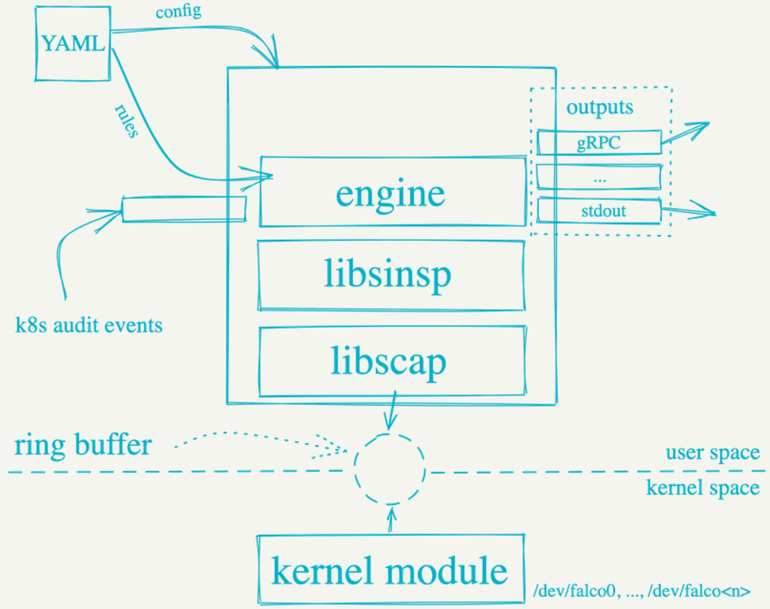
There are quite a few options for working with the traced information, such as capturing data a container writes to disk or memory for further investigation, as well as extracting binaries from a container. All of these features allow Tracee to provide in-depth tracing of an entire container or pod.

The tool is maturing quickly. So far, new versions are quite a bit different than previous versions. You may want to use a particular version and update to new versions in a planned manner.

Falco

Falco is a tool consisting of multiple components working together in order to evaluate system calls against rules, and generate alerts when a rule is broken:

- It is an open source runtime security tool
- Parses the Linux system calls from the kernel at runtime
- Asserting the stream against a powerful rules engine
- Alerting when a rule is violated
- Falco is composed of three main components: a userspace program, configuration, and a driver.



Changing Containers

The tools we have seen up to this point have, for the most part, scanned images prior to execution, or are perhaps part of a container certification process done in development prior to production. Another consideration for security staff is to ensure the container is immutable. In this case, the meaning is that the container or application can elevate or change while running, perhaps long after it is started. It cannot mutate while running.

While there is no set list of characteristics to monitor, consideration should be given to containers with read-write filesystems for source data, the ability to elevate privileged users and other features. While there are several tools to allow or deny ability on a granular basis, it may be a good idea to regularly schedule evaluation of non-immutable containers and reevaluate to ensure they conform to current security restrictions. A security **“spring cleaning”**.

Various Tools

There are several tools you could use to lock down virtually every part of the operating system. While technically they could be used together, they typically are not. SELinux is by far the most commonly used, and now runs on Red Hat, Debian, and SUSE-based distros. AppArmor runs on Debian and SUSE systems, and is often viewed as a less complete and simpler alternative. One could question how long AppArmor will be used, now that SELinux is easily available, and meets Common Criteria and FIPS standards. Smack and TOMOYO are not used in production as commonly, but may fit a particular need.

SELinux settings can be quite granular as well. For example, there are some 14 or so tasks when the ls command is run. Each of those tasks could be allowed or denied using SELinux, to some combination of users, roles, categories, and sensitivity levels. For many, this granularity is daunting and leads to fully disabling SELinux. This is not a suggested setting.

TOMOYO is a pathname-based MAC developed by NTT Data Corporation, which enables state transition by execution. It also has some system analysis tooling.

Smack, which stands for Simplified Mandatory Access Control Kernel, is aimed at an easy-to-use MAC system and often found with YOCTO and Automotive Grade Linux builds.

SELinux Overview

SELinux was originally developed by the United States NSA (National Security Administration) using a company named Tresys. Red Hat was also an early partner, which helped develop and integrate the software for government and military applications. It has been integral to RHEL for a long time, which has brought it a large usage base.

Operationally, SELinux is a set of security rules that are used to determine which processes can access which files, directories, ports, and other items on the system. As all objects in Linux are some sort of "file", and everything that is done is some sort of "process", SELinux can be used to control everything.
SELinux works with these three conceptual quantities:

1. Contexts - These are labels to files, processes and ports. Examples of contexts are SELinux user, role and type.
2. Rules - They describe access control in terms of contexts, processes, files, ports, users, etc.
3. Policies - They are a set of rules that describe what system-wide access control decisions should be made by SELinux.

SELinux Overview

A SELinux context is a label used by a rule to define how users, processes, files and ports interact with each other. As the default policy is to deny any access, rules are used to describe allowed actions on the system. Each of the actions must be allowed via the Access Vector Cache, which we will see more about soon.

SELinux Enforcement Modes

SELinux enforcement modes are selected (and explained) in a file (usually /etc/selinux/config) whose location varies by distribution (it is often either at /etc/sysconfig/selinux or linked from there). The file is well self-documented. There are three modes for SELinux:

- Enforcing - All SELinux code is operative and access is denied according to the policy. All audited violations are logged. There are many violations which are not logged; some versions had almost ten thousand errors which are not logged. They are part of the dontaudit list. This action would not be allowed and there would be no log kept. You can switch back and forth from Enforcing mode to Permissive dynamically.

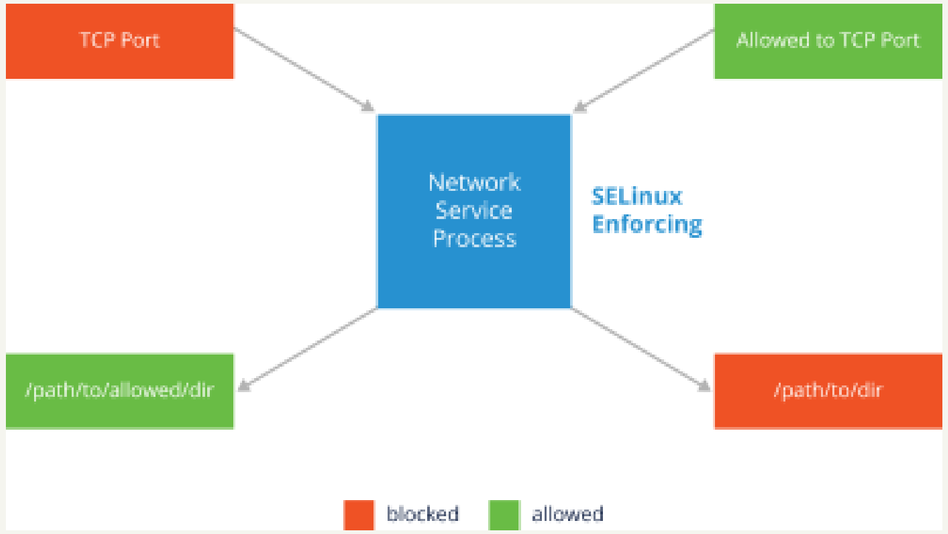
- Permissive - Enables SELinux code, but only warns about operations that would have been denied in enforcing mode. The dontaudit events remain silent. This is a good mode to start with. From here, you will have a list of actions that should not be allowed, but no users are impacted. Over time, each action can be solved, or the policy can be updated to allow the action. After all things that should be allowed are included, or made more secure, you can dynamically switch to Enforcing mode, and grow into a more secure system with minimal impact.

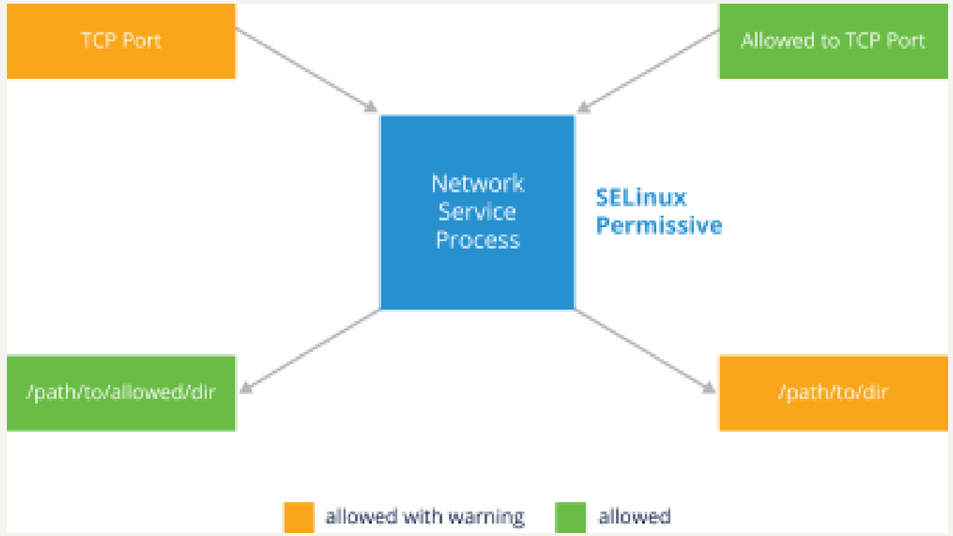
SELinux Enforcement Modes

- Disabled - Completely disables the SELinux kernel and application code leaving the system without any of its protections. The only way to enter or leave this mode is by reboot. Should you enable SELinux again, be aware that the first boot will take much longer than typical, as every file must be relabeled.

You can use **getenforce** and **setenforce** to see or set current mode.

The **sestatus** utility can display the current mode and policy. The **seinfo** command can show a lot of details, if a policy file is available.





getenforce and setenforce

Use getenforce for view the current SELinux state.

While setenforce allows you to switch between Permissive and Enforcing modes, it does not allow disabling SELinux completely. Only via a reboot can SELinux be put into a disabled mode. There are at least two different ways to disable SELinux:

- Configuration file - Edit the SELinux configuration file (usually /etc/selinux/config? and set SELINUX=disabled, then reboot. This is the default method and should be used to permanently disable SELinux.
- Kernel parameter - Add selinux=0 to the Kernel parameter list when rebooting.

However, it is important to note that disabling SELinux on systems in which SELinux will be re-enabled is not recommended. It is preferable to use Permissive mode instead of disabling SELinux, so as to avoid relabeling the entire filesystem which can be time consuming. There is no way to enable SELinux and not relabel files.

Default SELinux Policies

The same configuration file that sets the mode, usually **/etc/sysconfig/selinux**, also sets the SELinux policy. Multiple policies are allowed, but only one can be active at a time.

Changing the policy may require a reboot of the system and a time-consuming re-labeling of filesystem contents. Each policy has files which must be installed under /etc/selinux/[SELINUXTYPE].

targeted - The default policy in which SELinux is more restrictive to targeted processes. User processes and init processes are not targeted, while network service processes are targeted. SELinux enforces memory restrictions for all processes, which reduces the vulnerability to buffer overflow attacks.

minimum - A modification of the targeted policy where only selected processes are protected.

Multi-Level Security (MLS) - The Multi-Level Security policy is much more restrictive; all processes are placed in fine-grained security domains with particular policies.

Context Utilities

As mentioned earlier, contexts are labels applied to files, directories, ports, and processes. Those labels are used to describe access rules. There are four SELinux contexts: User, Role, Type, and Level.

We will focus on type, the most commonly utilized context. The label naming convention determines that type context labels should end with _t, as in kernel_t.

To look up or change a content, we can use the -Z option to see the context:

```
$ ls -Z
$ ps -auZ
```

we can use the chcon command to change the context:

```
$ chcon -t etc_t somefile
$ chcon --reference somefile someotherfile
$ chcon --reference somefile somefile1
```

Context Inheritance

Newly created files inherit the context from their parent directory, but when moving or copying files, it is the context of the source directory which may be preserved, which can cause problems.

Copy will make file to inherit the context of the destination directory.

Move will make file to retain its contexts.

restorecon

restorecon resets file contexts, based on parent directory settings.

In the following example, restorecon resets the default label recursively for all files at the home directory

```
restorecon -Rv /home/alok
```

semanage

Another issue is how to configure the default context for a newly-created directory. semanage fcontext (provided by the policycoreutils-python package) can change and display the default context of files and directories. Note that semanage fcontext only changes the default settings; it does not apply them to existing objects. This requires calling restorecon afterwards.

For example:

```
mkdir /virtualHosts
semanage fcontext -a -t httpd_sys_content_t /virtualHosts
restorecon -RFv /virtualHosts
```

SELinux Booleans

A boolean is a parameter which has a value of either true or false. A SELinux boolean is an easy way of enabling or disabling functionality without needing to edit or fully understand the underlying code. For example, the allow_ftpd_anon_write boolean may allow several different processes to act on files that were otherwise disallowed.

You can use SELinux booleans to change the behavior of the SELinux policy, to enable or disable it:

- getsebool: to see booleans
- setsebool: to set booleans
- semanage boolean -l: to see persistent boolean settings

Monitoring SELinux Access

The **setroubleshoot-server** package provides tools to collect issues at runtime, log them, and propose solutions to prevent the issue in the future. After installation of the package, the auditd daemon needs to be restarted. Raw errors will be tagged as AVC errors and appended to the audit.log.

More human-readable output will be appended to the system log location, such as /var/log/messages.

AppArmor

AppArmor is an alternative LSM (Linux Security Module) to SELinux. Support for it has been incorporated in the Linux kernel since 2006. It has been used by SUSE, Ubuntu, and other distributions.

AppArmor supplements the traditional UNIX Discretionary Access Control (DAC) model by providing Mandatory Access Control (MAC).

In addition to manually specifying profiles, AppArmor includes a learning mode, in which violations of the profile are logged, but not prevented. This log can then be turned into a profile, based on the program's typical behavior.

- Allows administrators to associate a security profile to a program which restricts its capabilities.
- Is considered easier (by some, but not all) to use than SELinux.
- Is considered filesystem-neutral (no security labels required).

AppArmor in Kubernetes

In order for an AppArmor profile to be used by a pod, it must be available on the node where assigned. There is not a native process for Kubernetes to load policies. As a result, you need to ensure policies are loaded on every node where AppArmor-required pods are scheduled, and the scheduler is unaware of which nodes have profiles.

Adding profiles could be done during node installation with a tool like Ansible or Puppet, at least for some of the nodes. If only some nodes will have profiles installed, you could use a NodeSelector or a taint to ensure the scheduler chooses the appropriate node. Another solution would be to deploy a DaemonSet and allow the pod the ability to modify the host and add profiles. This would put the responsibility into the cluster administrators, if different from those responsible for operating system configuration and security.

Should you need to disable AppArmor on the entire cluster (perhaps to troubleshoot), pass the **--feature-gates=AppArmor=false** option. Profiles can also be managed via the PodSecurityPolicy extension admission controller.

Checking Status

Distributions that come with AppArmor, such as Ubuntu, tend to enable it and load it by default; note that the Linux kernel has to have it turned on as well, and in most cases, only one LSM can run at a time.

Assuming you have the AppArmor kernel module available, on a systemd-equipped system you can do:

$ sudo systemctl [start|stop|restart|status] apparmor
to change or inquire about the current state of operation, or do:
$ sudo systemctl [enable|disable] apparmor
to cause to be loaded or not loaded at boot.
In order to see the current status:
$ sudo apparmor_status

Modes & Profiles

Profiles restrict how executable programs, which have pathnames on your system, such as /usr/bin/evince, can be used. Processes can be run in either of the following two modes:

- Enforce Mode - Applications are prevented from acting in ways which are restricted. Attempted violations are reported to the system logging files. This is the default mode. A profile can be set to this mode with aa-enforce.
- Complain - Polices are not enforced, but attempted policy violations are reported. This is also called learning mode. A profile can be set to this mode with aa-complain.

Linux distributions come with pre-packaged profiles, typically installed either when a given package is installed, or with an AppArmor package, such as apparmor-profiles. These profiles are stored in /etc/apparmor.d.

When installing new software, new profiles can be created specific to any executables in the package.