linode

HackerSploit: Docker Security Essentials

A guide to **auditing and securing** the Docker platform and containers

linode

SECURITY

DOCKER

Table of Contents

Table of Contents

Table of Contents

Prerequisites & Requirements

PREREQUISITES

This guide only focuses on securing the Docker platform on Linux as it is the most widely utilized and deployed version of the technology.

In order to to follow along with the techniques demonstrated in this guide, you need to have a Linux server with the following services installed and running:

- Docker

> **Note:** The demonstrations illustrated in this guide have been performed on an Ubuntu 20.04 server running Docker CE. The commands are distribution agnostic with the exception of package names, package managers, and the respective init systems.

TECHNICAL REQUIREMENTS

- Fundamental knowledge of Docker and Docker CLI commands.
- Functional knowledge of Linux terminal commands.
- Fundamental knowledge of systemd and Linux init systems.

Introduction

Given the increased adoption of Docker by individuals and companies for the containerization, deployment, and hosting of web applications, databases, and other business critical applications, it comes as no surprise that the process of securing the Docker platform is paramount to the implementation and the successful long term application of the platform. The Docker platform is deployed widely across various spectrums of use, and this increased adoption brings up various security issues and pitfalls that plague every other technology with widespread adoption. When a technology is widely used and implemented, the security of the platform is usually put under a microscope as attackers constantly try to identify misconfigurations and vulnerabilities in the technology and its implementation. Failure to configure and secure the Docker platform can result in massive data breaches and exploitation of systems and networks.

It is for this reason that the security of the Docker platform needs to be taken seriously. This should also necessitate the formation of a functional security policy that addresses the security issues and misconfigurations of the platform.

The most common mistake made by individuals and companies is the assumption that the Docker platform is secure out of the box. As with many platforms, this is not the case, and implementations of the Docker platform need to be secured from the ground up.

Another impediment that prevents the adoption and implementation of the technology is the abstraction and complexity of the component technologies that make up the platform. Until recently, containers were not considered a mainstream alternative to virtual machines, primarily because of the technical and idiosyncratic nature of containerization technologies like LXC. Docker was developed to simplify the adoption of containerization technologies and make them available to a wider demographic of users. To its credit, it has achieved this objective and is constantly being improved to make the process more efficient. However, the process of securing Docker can still be unintuitive for organizations.

This ebook aims to provide a clear and concise guide to securing the Docker platform and consequently Docker containers at runtime. This process needs to be approached systematically and requires a functional knowledge of the components that make up the platform, and of the two primary Linux kernel primitives that make containerization possible: namespaces and cgroups.

Introduction

The content in this guide is structured and organized as follows:

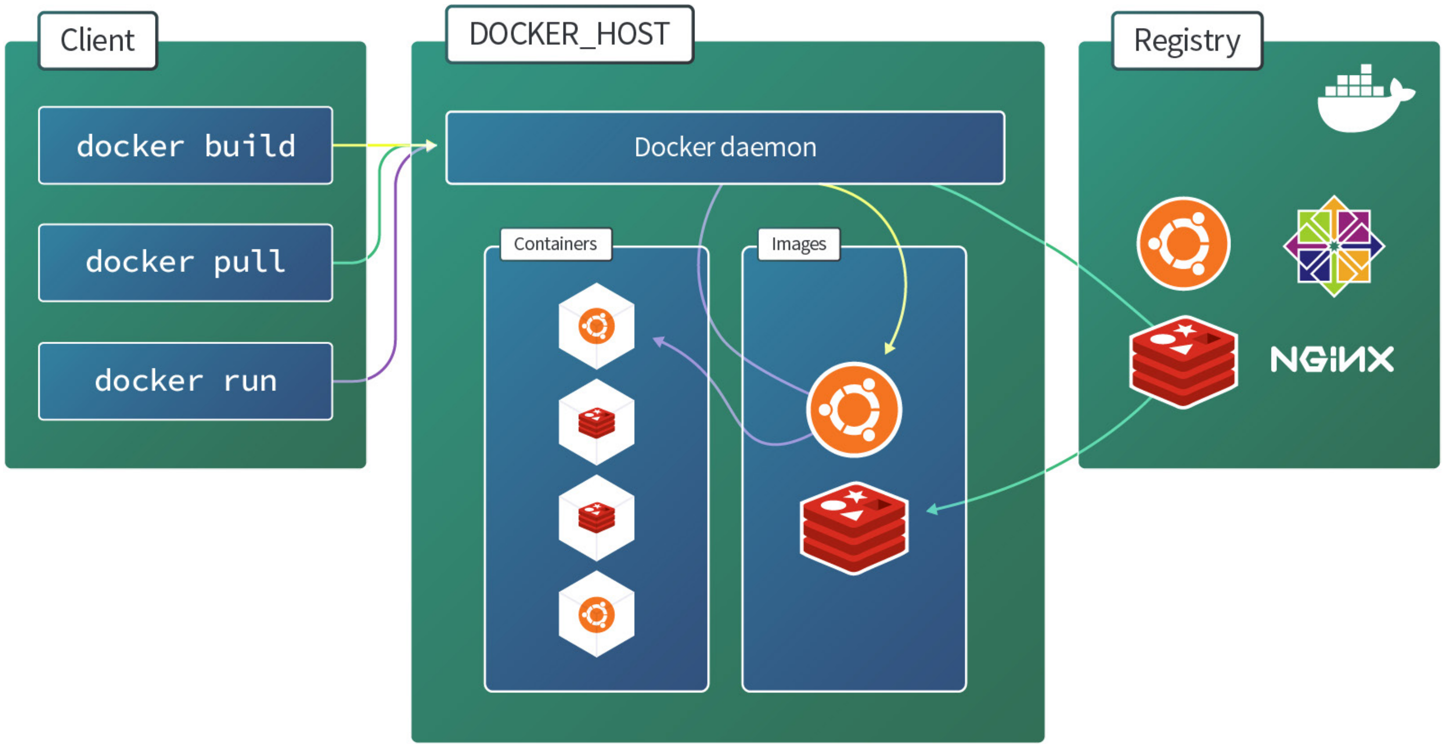
- In **The Docker Platform** section, we will begin the process by explaining the various components that make up the Docker platform.
- In the **Auditing Docker Security** section, we will explore the process of performing a security audit of the Docker platform. An audit identifies vulnerabilities in the configuration of the components that make up the platform.
- In the next two sections, we will begin the process of securing the Docker host and the Docker daemon to ensure that we have a secure base to operate from:
  - **Securing the Docker Host**
  - **Securing the Docker Daemon**
- The remaining sections of the guide will conclude by taking a look at the various ways of securing containers and the process of building secure Docker images:
  - **Container Security Best Practices**
  - **Controlling Container Resource Consumption with Control Groups (cgroups)**
  - **Implementing Access Control with AppArmor**
  - **Limiting Container System Calls with seccomp**
  - **Vulnerability Scanning for Docker Containers**
  - **Building Secure Docker Images**

Let’s begin the process by taking a look at how the Docker platform is designed and organized.

The Docker Platform

Docker is a PaaS (platform as a service) containerization technology. It utilizes OS-level virtualization that allows users to package, distribute, and deploy software, web apps, and any other type of data that can be containerized. Docker distinguishes itself from classic level 2 hypervisors by utilizing the host OS kernel as opposed to virtualizing an operating system for each container.

The following diagram outlines the various components that make up the platform and their inter-connectivity.



In order to understand the process of securing the Docker daemon, we need to take a closer look at how communication between these components is facilitated:

- Communication between the components that make up the Docker platform is facilitated through the use of several APIs.
- The Docker client communicates with the Docker daemon through a Unix domain socket or remotely through a TCP socket.
- Commands sent from the Docker client are sent to the Docker daemon.
- Collectively, the Docker APIs, Docker CLI, and Docker daemon are referred to together as the Docker Engine.

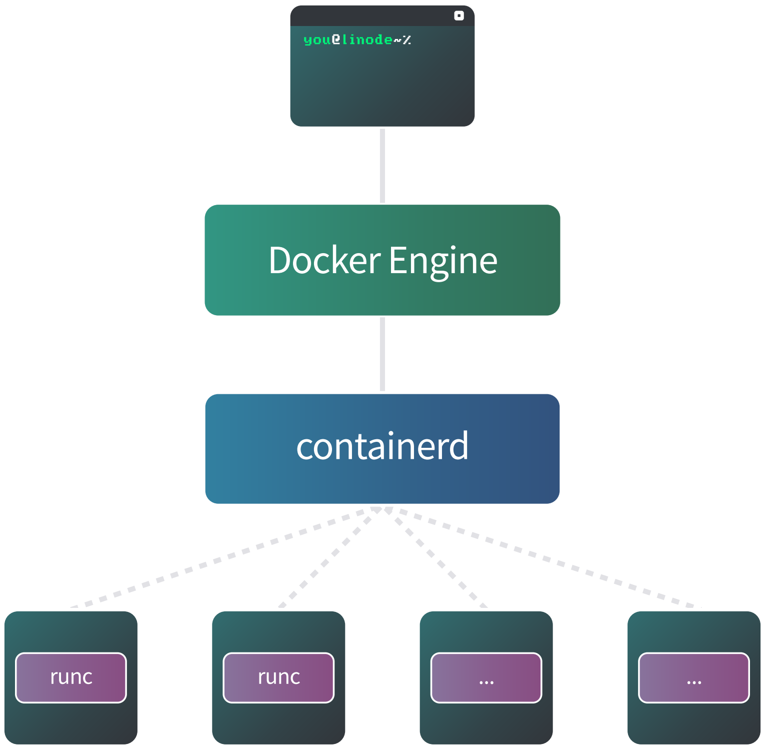
The Docker Platform

However, Figure 1.0 is also a simplified representation of how Docker works:

- The Docker daemon, in turn, forwards commands to Containerd, which is another daemon that manages the containers and performs related functions, like pushing and pulling images and container storage.
- Containerd is an industry-standard container management solution that's also used by other platforms, like Kubernetes.
- Communication between the Docker daemon and Containerd is facilitated through the gRPC (open source remote procedure call).
- Furthermore, Containerd itself utilizes a runtime specification, typically runc, to create and manage the actual containers.

This modularization of components is not random. Docker initially bundled all functionality into the Docker daemon, which centralized most of the functionality, consequently making it bloated and leading to a reduction in performance. This centralized structure was later overhauled in favour of a modularized structure, and containerd was created as part of this modularization effort. The modularization of components also makes it much simpler to secure as each component can be handled and secured individually.

Now that you have an understanding of how the Docker platform is structured and organized, we can begin the process of auditing the security of the Docker host.



Auditing Docker Security

The first step in the process of securing a system is to perform a security audit. An audit establishes a baseline of the security of a system. This initial baseline will be used to guide us in accordance with what needs to be secured.

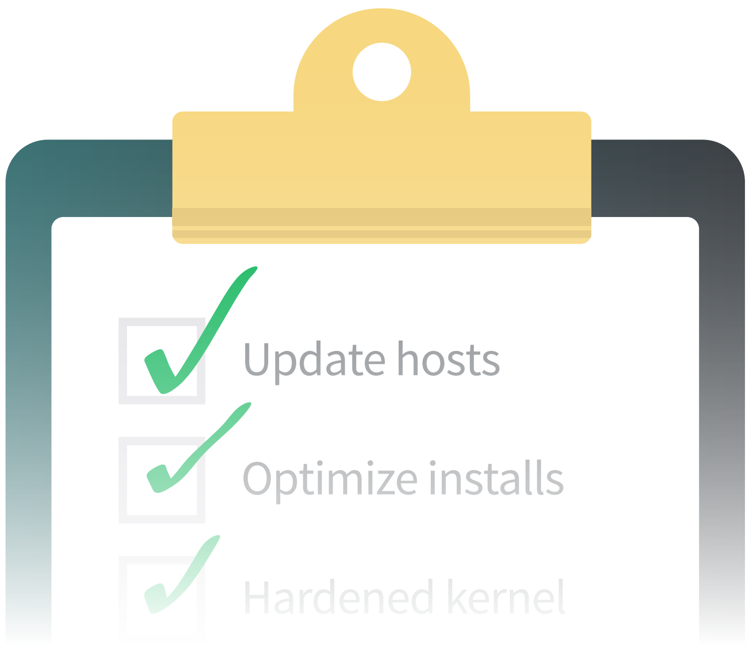
Before we get started with the security auditing process, we need to understand what a security audit is and why it is important in securing a system.

WHAT IS A SECURITY AUDIT?

A security audit is a systematic evaluation of the security and configuration of a particular information system. Security audits are used to measure the security performance of a system against a list of checks, best practices, and standards.

In the case of Docker, we will be using the CIS Docker Benchmark, which is a consensus driven security guideline for the Docker platform. The CIS Docker Benchmark provides us with a solid set of guidelines and checks that can be used to test the security of the Docker platform and establish a baseline security level. More information about the CIS Docker Benchmark can be found here: **https://www.cisecurity.org/benchmark/docker/**

The process of auditing the security of Docker can be automated using various tools. In this guide, we will be using the Docker Bench for Security utility developed by Docker, Inc.



Auditing Docker Security

DOCKER BENCH FOR SECURITY

Docker Bench for Security is an open source Bash script that checks for various common security best practices of deploying Docker in production environments. The tests are all automated and are based on the CIS Docker Benchmark. More information about Docker Bench for Security can be found on GitHub: https://github.com/docker/docker-bench-security

Now that you have an understanding of security audit concepts and the tools and benchmarks we will be using, we can begin the process of performing a security audit on our Docker host.

AUDITING DOCKER SECURITY WITH DOCKER BENCH FOR SECURITY

The auditing process can be performed by following the procedures outlined below:

1. You first need to clone the docker/docker-bench-security GitHub repository on your Docker host. This can be done by running the following command:

   ```
   git clone https://github.com/docker/docker-bench-security.git
   ```

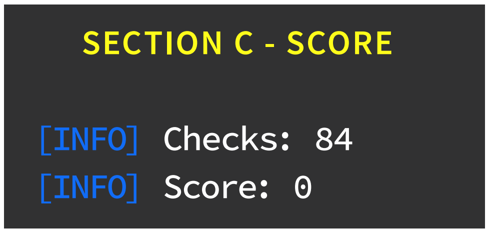
2. After cloning the repository, you will need to navigate into the docker-bench-security repository that you just cloned:

   ```
   cd docker-bench-security
   ```

3. The cloned directory will contain a Bash script named docker-bench-security.sh. We can run this script to perform the Docker security audit by running the following command:

   ```
   sudo ./docker-bench-security.sh
   ```

4. When the script is executed, it will perform all the necessary security checks. Once completed, it will provide you with a baseline security score as highlighted in the image below.



```
SECTION C - SCORE

[INFO] Checks: 84
[INFO] Score: 0
```

The initial baseline security score will be valued at zero, indicating that all checks failed. In this case, we can identify what needs to be secured by analyzing the results produced by the script, as highlighted in the image below.

Auditing Docker Security

```
Section A  - Check results

[INFO] 1 - Host Configuration
[INFO] 1.1 Linux Hosts Specific Configuration
WARNING: No swap limit support
WARNING: No blkio weight support
WARNING: No blkio weight_device support
[WARN] 1.1.1 Ensure a separate partition for containers has been created (Automated)
[INFO] 1.1.2 Ensure only trusted users are allowed to control Docker daemon (Automated)
[INFO]     * Users: alexis
[WARN] 1.1.3 Ensure auditing is configured for the Docker daemon (Automated)
[WARN] 1.1.4 Ensure auditing is configured for Docker files and directories /run/containerd (Automated)
[WARN] 1.1.5 Ensure auditing is configured for Docker files and directories /var/lib/docker (Automated)
[WARN] 1.1.6 Ensure auditing is configured for Docker files and directories /etc/docker (Automated)
```

Each check performed by the script is numbered and is flagged with the corresponding color code based on whether the check was successful:

`WARN:` The corresponding check failed, indicating its need to be secured.
`INFO:` The check was run with no warning.
`PASS:` The corresponding check was run successfully.

The script also provides a list of recommendations regarding what components need to be secured for every check. For example, as shown in the image below, we need to enable auditing for the Docker daemon:

```
[WARN] 1.1.3 Ensure auditing is configured for the Docker daemon (Automated)
```

> **Note:** In this context, the warning is specifically referring to using the Linux Audit Framework. This topic will be introduced later, in the **Setting Up Audit Rules for Docker Artifacts** section.

The script also sorts the results based on the following categories:

- Host configuration
- General configuration
- Docker daemon configuration
- Docker swarm configuration

This categorization of checks is very useful as it distinguishes the security of components from others, therefore streamlining the process. The first component that we need to secure based on the results is the Docker host. Let's take a look at how to secure the Docker host and implement the security practices recommended by the Docker Bench for Security tool.

Securing The Docker Host

Given the fact that Docker containers utilize the host OS kernel, the Docker platform and its containers are only as secure as the host operating system. In this guide, our host OS is running Linux, but similar principles should be followed for other operating systems.

HOST SECURITY

The security of the host kernel and operating system will have a direct correlation to the security of your containers, given the fact that the containers utilize the host kernel. It is therefore vitaly important to keep your host secure. The following guidelines outline various security best practices you should consider when securing your Docker host:

1. Consider the use of minimal Linux distributions that offer a much smaller attack surface.
2. Secure and harden your host OS.
3. Ensure your host OS is kept up to date.
4. Ensure your kernel is up to date.
5. Ensure you have the latest version of Docker running.
6. Add your host and containers to a robust vulnerability management plan and constantly scan your host and containers for vulnerabilities.
7. Only run the services you need to run.
8. Keep up with the latest vulnerability news for the Linux kernel and the Docker platform.

The process of securing the host OS is multi-faceted and leverages multiple security audit tools in order to establish a baseline security level. This process will result in a Docker host that satisfies the CIS Docker Benchmark.

We will address securing the Docker host in two parts:

1. First, we will run an operating system security audit tool called Lynis. This will help us secure and harden the host OS. We will implement the recommendations made by Lynis.

2. After we harden the host OS, we will return to the Docker Bench for Security to enable and set up auditing for our Docker components and artifacts.

Securing The Docker Host

SECURITY AUDITING WITH LYNIS

Lynis is an extensible security audit tool for computer systems running Linux, FreeBSD, macOS, OpenBSD, Solaris, and other Unix derivatives. It assists system administrators and security professionals with scanning a system and its security defenses, with the final goal being system hardening.

Installing Lynis

Lynis is available as a package for most Linux distributions. We can install it by running the following command on Debian-based systems:

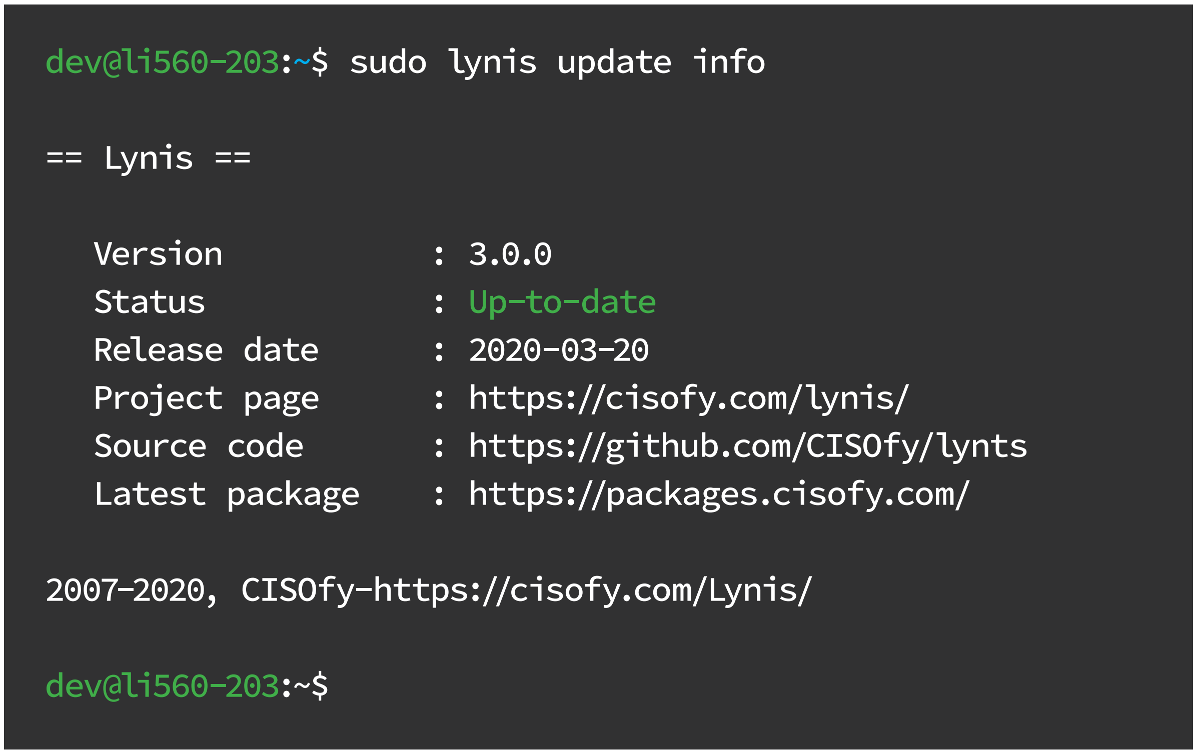
```
sudo apt-get install lynis
```

To display all the options and commands available for Lynis, we can run the following command:

```
lynis show options
```

Before we get started with scanning, we need to ensure that Lynis is up to date. To check if we are running the latest version we can run the following command:

```
sudo lynis update info
```



```
dev@li560-203:~$ sudo lynis update info

== Lynis ==

  Version             : 3.0.0
  Status              : Up-to-date
  Release date        : 2020-03-20
  Project page        : https://cisofy.com/lynis/
  Source code         : https://github.com/CISOfy/lynts
  Latest package      : https://packages.cisofy.com/

2007-2020, CISOfy-https://cisofy.com/Lynis/

dev@li560-203:~$
```

Securing The Docker Host

RUNNING LYNIS

To perform a system audit with Lynis, run the following command:

```
sudo lynis audit system
```

Lynis will output a lot of information that will also be stored under the /var/log/lynis.log file for easier access. The summary of the system audit will reveal important information about your system's security posture and various security misconfigurations and vulnerabilities.

Lynis will also generate output on how these vulnerabilities and misconfigurations can be fixed or tweaked.

```
Lynis security scan details:

Hardening index : 65 [############       ]
Tests performed : 249
Plugins enabled : 0

Components:
- Firewall               [V]
- Malware scanner        [X]

Scan mode:
Normal [V] Forensics [ ] Integration [ ] Pentest [ ]

Lynis modules:
- Compliance status      [?]
- Security audit         [V]
- Vulnerability scan     [V]

Files:
- Test and debug information : /var/log/lynis.log
- Report data                : /var/log/Lynis-report.dat
```

Securing The Docker Host

The output also contains a hardening index score that is rated out of 100. This is used to give you a trackable tangible score of your system's current security posture. Lynis will also display any potential warnings that indicate a severe security vulnerability or misconfiguration that needs to be fixed or patched. In this case, we have no warnings.

```
-[ Lynis 3.0.0 Results ]

Great, no warnings
```

To increase our hardening index score, Lynis provides us with helpful suggestions that detail the various security configurations we need to make.

```
Suggestions (50):
--------------------------
• This release is more than 4 months old. Consider upgrading [LYNIS]
   https://cisofy.com/lynis/controls/LYNIS/

• Set a password on GRUB boot loader to prevent altering boot configuration [BOOT
5122]
   https://cisofy.con/Lynis/controls/B0OT-5122/

• Consider hardening system services [BOOT-5264]
Details : Run '/usr/bin/systend-analyze security SERVICE' for each service
   https://cisofy.com/lynts/controls/800T-5264/

• If not required, consider explicit disabling of core dump in /etc/security/limit
   https://cisofy.con/lynis/controls/KRNL-5820/

• Check PAM configuration, add rounds if applicable and expire passwords to encrypt
   https://ctsofy.com/lynis/controls/AUTH-9229/

• Configure mininum encryption algorithm rounds in /etc/login. defs [AUTH-9230]
   https://cisofy.com/lynis/controls/AUTH-9230/

• Configure maximum encryption algorithm rounds in /etc/login.defs [AUTH-9230]
   https://cisofy.com/Lynts/controts/AUTH-9230/
```

We can now follow the recommendations provided by Lynis to secure and harden our Docker host.

Securing The Docker Host

CREATING A USER ACCOUNT

We are now ready to begin securing our host OS:

- The first step is to add and configure the necessary user accounts on the system.
- We then need to set up the various groups that will be used to assign permissions to particular users with specific roles.
- After, we will begin specifying file permissions and assigning ownership of particular files and directories. This will help us set up a system of accountability and defense in depth.

Linux has multi-user support and, as a result, multiple users can access the system simultaneously. This can be seen as both an advantage and disadvantage from a security perspective in that multiple accounts offer multiple access vectors for attackers and therefore increase the overall risk of the server. To counter this concern, we must ensure that user accounts are set up and sorted accordingly in terms of their privileges and roles. For example: Having multiple users on a Linux server with root privileges is extremely dangerous as an attacker will only need to compromise one account to get root access on the system. We can easily solve this issue by segregating permissions for users based on their roles.

Securing The Docker Host

Creating a user account on Linux can be done by following the steps outlined below:

1. The `useradd` command creates users on your system and has this general syntax:

   ```
   useradd <arguments> username
   ```

2. The arguments that can be included are used to specify particular information and configurations for the user account. Some of these options are described in the table below:

Argument	Function
`-c`	A text string that is used to include comments about the account, like the user's first and last name.
`-m`	When included, this option tells the `useradd` command to create a home directory for the new user.
`-s`	Used to specify the user's login shell (e.g. `/bin/bash, /bin/zsh`, etc).

3. Now that we understand the arguments we can specify or use when creating a user, let us create the user account:

   ```
   useradd -c "First Name Last Name" -m -s /bin/bash <username>
   ```

4. We have used the `-c` argument to specify the full name of the user, and we have used the `-s` argument to specify that Bash should be the default shell for the new user. The `-m` argument will create the home directory for the user. We finally end the statement with the username of the account.

5. We now need to specify the password for the user account. We can do this with the following command:

   ```
   passwd <username>
   ```

6. We will then be prompted to enter a password for the user. Make sure to use a strong password that follows the specification in your organization's security policy, if applicable.

Securing The Docker Host

SETTING UP SUDO ACCESS

When setting up access on a Linux server, some users may require sudo access to perform administrative tasks like updating packages and installing software. By default, users do not have sudo access, which means they are unable to perform these administrative tasks.

Giving a user sudo access involves adding the user to a sudo-enabled group. By default, this group is just called sudo on Debian-based systems, and on Fedora and RedHat-based systems this group is called wheel. One way we can add the user we have just created to the sudo group by running the following command:

```
usermod -aG sudo <username>
```

ADDING THE USER TO THE DOCKER GROUP

Docker implements access control for the Docker daemon through a Linux group with specific permissions. Members of this group will have the privileges required to interact with the Docker daemon. As a result, only authorized users that require access should be added to this group.

We can add our custom user to this group by running the following command:

```
usermod -aG docker <username>
```

DISABLING ROOT LOGINS

The first step in setting up local authentication security is to disable root logins . Following this step prevents any authorized or unauthorized user from gaining access to the root user account and consequently the server because the root user has complete power over the system.

The root user’s privileges can be abused to run any commands provided (malicious or otherwise), including modifying the passwords of other users on the system, consequently locking them out. Common Linux security practices recommend disabling root logins and creating a separate administrative account, which can be assigned sudo privileges to run certain commands with root privileges. Following this step will help mitigate the threats to the root account and will reduce the overall attack surface of the host.

Securing The Docker Host

We can disable root logins in a few different ways. The first method of disabling root logins is by changing the default shell of the root user from `/bin/bash` or `/bin/sh` to `/usr/sbin/nologin`. This can be done by using the `chsh` (Change Shell) utility on Linux:

1. Run the following command:

   ```
   sudo chsh root
   ```

2. After running the command, we will be prompted to enter the absolute path of the shell we want to switch to. Specify `/usr/sbin/nologin` as the shell at the prompt.

3. After you have entered the absolute path to the nologin shell, we can try logging in to the root account. When attempting to log in, the message

   ```
   This account is currently not available
   ```

   appears, and we are unable to log into the root account:

4. These changes will prevent unauthorized users from using the root account, because we have not specified a valid shell. However, users with sudo privileges will still be able to run all administrative commands unless the privileges are constrained to certain commands.

   > **Note:** Aside from using the chsh utility, another way to update the user's shell is to modify the `/etc/passwd` file.

Securing The Docker Host

The second method of preventing root logins is by locking the password of the root account with the `passwd` utility. This will add an additional layer of security. Locking the password of an account on Linux will not disable the account; it will simply disable local password authentication for the account.

> **Note:** Users will still be able to login to the account remotely via SSH keys, if they have been set up. The process of securing SSH is introduced in the next section.

We can lock the password of the root account by running the `passwd` command with the `-l` option:

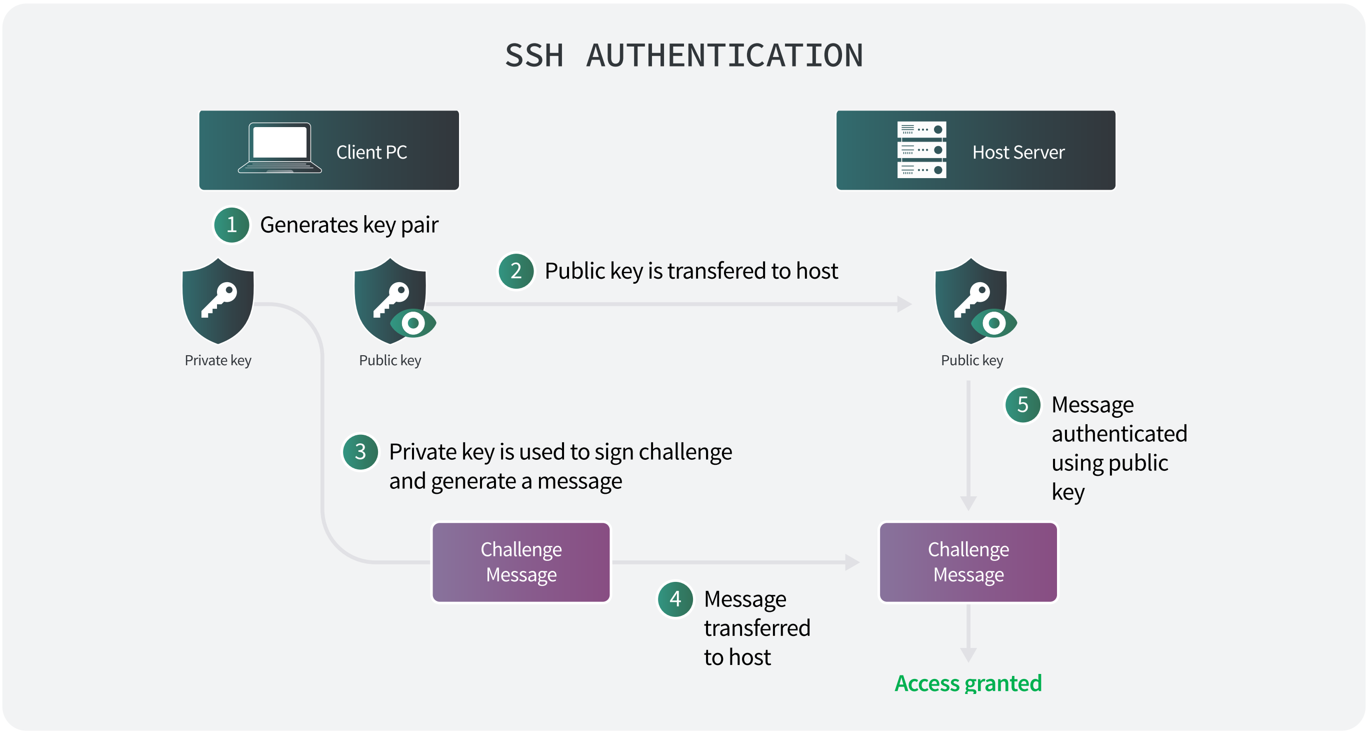
```
sudo passwd -l root
```

If you want to unlock the password for a specific account, you can use the `-u` unlock option for the passwd command:

```
sudo passwd -u root
```

This will unlock the password for the root account and you will be able to access the account via password authentication.

Now that we have disabled root user logins, we will be using the custom user account that we have created going forward. The next step in authentication security involves securing the remote access protocol, which in most cases will be SSH.


SSH AUTHENTICATION
Client PC
Host Server
1 Generates key pair
Private key
Public key
2 Public key is transfered to host
Public key
5 Message authenticated using public key
3 Private key is used to sign challenge and generate a message
Challenge Message
4 Message transferred to host
Challenge Message
Access granted

Securing The Docker Host

SECURING SSH

If your system did not have root password logins disabled, then any attacker could attempt to gain root access by performing password brute-force attacks on the SSH protocol. So, it's important to disable root login via SSH as well.

It's also important to do this even if you do have root password logins disabled, because it adds an extra layer of security. Furthermore, it prevents root logins with alternative authentication methods, like key-based authentication, which will be explored in the next section.

1. We can disable root login via SSH by modifying the OpenSSH server configuration file found in `/etc/ssh/sshd_config`.
2. After opening the file with a text editor like nano or vim, we will be greeted with extensive configuration options that we can use to modify how the SSH server will function.

```
# $OpenBSD: sshd config. v 1.103 2018/04/09 20:41:22 ti Exp $

This is the sshd server system-wide configuration file. See
sshd_config(5) for more information.

This sshd was compiled with PATH=/usr/bin:/bin:/usr/sbin:/sbin

The strategy used for options in the default sshd config shipped with
OpenSSH is to specify options with their default value where
# possible, but leave them commented. Uncommented options override the
default value.

#Port 22
#AddressFamily any
#ListenAddress 0.0.0.0
#ListenAddress :

#HostKey /etc/ssh/ssh host_rsa_key
#HostKey /etc/ssh/ssh_host_ecdsa_key
#HostKey /etc/ssh/ssh_host_ed25519_key

#Ciphers and keying
#RekeyLimit default none

#Logging
#SyslogFacility AUTH
#LogLevel INFO
```

Securing The Docker Host

3. To disable root login with SSH, we need to change the `PermitRootLogin` configuration from `yes` to `no`. The authentication configurations can be found under the `#Authentication` section. Ensure that you also uncomment the configuration to activate it by removing the # symbol at the beginning of the `PermitRootLogin` line.

```
#Authentication:

#LoginGraceTime 2m
#PermitRootLogin no
#StrictModes yes
#MaxAuthTries 6
#MaxSessions 10
```

4. As you can see in the image above, we have set the option from yes to no. This will prevent users from authenticating via SSH as the root user.

5. After saving the file, we now need to restart the SSH service. This can be done by running the following command:

```
sudo systemctl restart sshd
```

6. After restarting the SSH daemon on the server, we can try logging in to the root account remotely via SSH. As you can see in the image below we get a `Permission Denied` error even after entering the correct root password. This confirms that we have successfully disabled root logins via SSH.

```
alexis@lenovo:~$ ssh root@192.168.1.107
root@192.168.1.107's password:
Permission denied, please try again.
root@192.168.1.107's password:
Permission denied, please try again.
root@192.168.1.107's password:
```

Securing The Docker Host

SETTING UP KEY-BASED AUTHENTICATION WITH SSH

Key-based authentication utilizes asymmetric encryption to generate two keys that are used for the encryption and decryption of data. These two keys are called the public key and the private key, and together they are called a public-private key pair.

The public key is used to encrypt data and only the corresponding private key can decrypt the data. As a result, the private key must be kept private and secure, whereas the public key can be shared.

1. SSH key pairs can be generated on the client by using the `ssh-keygen` utility. We can generate the key pair by running the following command:

   ```
   ssh-keygen -t rsa
   ```

2. This will generate the public and private RSA key pair, and you will be prompted to specify the directory to which you want to save the keys. You will also be prompted to specify a passphrase for the key pair. This is an additional level of security that you can use to secure your key pair.

```
alexis@lenovo:~$ ssh-keygen-t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/alexis/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/alexis/.ssh/id_rsa.
Your public key has been saved in /home/alexis/.ssh/id_rsa.pub.
The key fingerprint is:
SHA256: pT4MXs8bTIr4v/GYOZanrDdbklhTDhvx8NMMFLIVQKg alexis@lenovo
The key’s randomart image is:
+---[RSA 2048]----+
|E   ..o=*o.+o    |
|.   .oo+oo...    |
|....  o=..       |
| o+. o  =        |
|o .oo ooS.       |
|* ...+o oo       |
|oo.. o+o+o       |
| .   o+o+o       |
|      .o..       |
+----[SHA256]-----+
alexis@lenovo:~$
```

Securing The Docker Host

SETTING UP KEY-BASED AUTHENTICATION WITH SSH

3. The key pair will be generated and saved in your `~/.ssh/ directory`. In this directory, you will find your public key with the `.pub` extension (e.g. `id_rsa.pub`), and your private key with no file extension (e.g. `id_rsa`).

4. Your public key now needs to be uploaded to your server. We can do this with the `ssh-copy-id` utility:

   ```
   ssh-copy-id <username@SERVER-IP>
   ```

5. We are now able to log in directly without entering a user password. Note that if you previously supplied a passphrase to the `ssh-keygen` utility, you will be prompted to enter that passphrase when logging in.

DISABLE PASSWORD AUTHENTICATION WITH SSH

We can now login with our private key. The next step is to disable password authentication completely, which will ensure that no user will be able to authenticate remotely with SSH without their respective key pair.

1. This can be done by modifying the `/etc/ssh/sshd_config` OpenSSH configuration file and setting the `PasswordAuthentication` option to no:

   ```
   # To disable tunneled clear text passwords, change to no here!
   PasswordAuthentication no
   #PermitEmptyPasswords no
   ```

2. After saving the new changes to the OpenSSH configuration file, restart the SSH daemon:

   ```
   sudo systemctl restart sshd
   ```

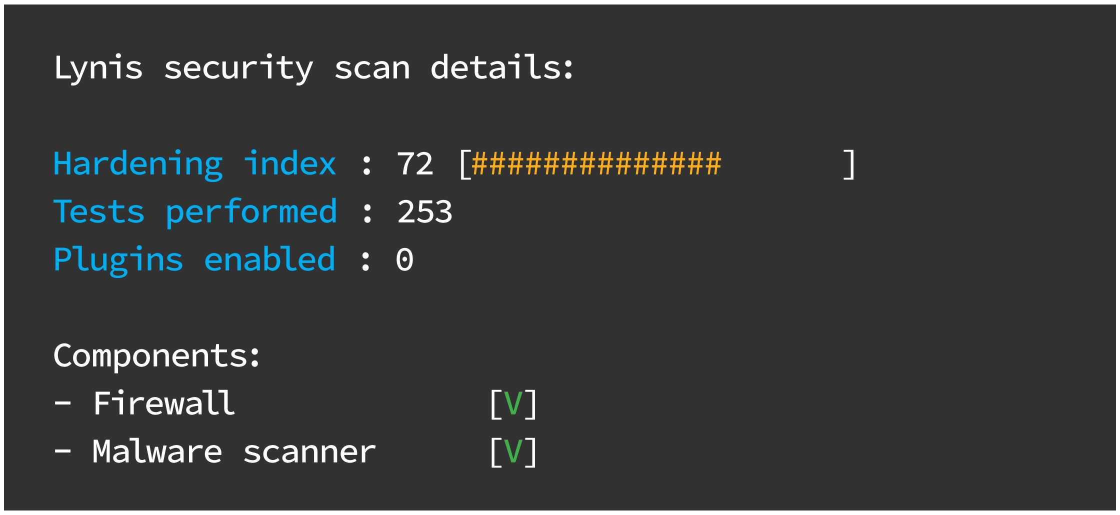
3. The SSH server will restart with the new changes applied.

We have now secured the new user account and root account from unauthorized remote access. We are only able to login to the user account with the unique private key from the key pair we generated.

Securing The Docker Host

RUNNING LYNIS AFTER IMPLEMENTING RECOMMENDATIONS

After implementing the recommendations provided by Lynis, we can run a system audit with Lynis again to verify the application of the changes we have made.



```
Lynis security scan details:

Hardening index : 72 [##############       ]
Tests performed : 253
Plugins enabled : 0

Components:
- Firewall               [V]
- Malware scanner        [V]
```

As highlighted in the preceding image, our hardening index should have increased as a direct consequence of following the security recommendations.

We can now move on the next step in securing the host, which involves setting up auditing for Docker artifacts.

SETTING UP AUDIT RULES FOR DOCKER ARTIFACTS

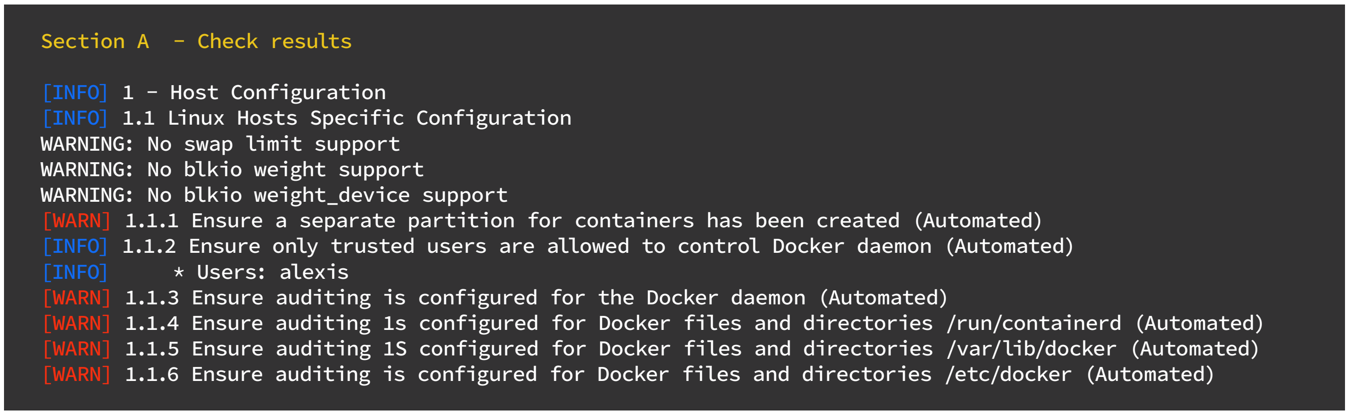
During the initial Docker security audit we performed with the Docker Bench for Security utility, we were able to identify several host configuration warnings that required us to set up audit rules for specific Docker artifacts. Examples of these artifacts include configuration files, binaries, and systemd service files.

We can perform the Docker Bench for Security utility again. This time, we can limit our results to the host configuration to focus on just those checks. This can be done by running the following command:

```
sudo ./docker-bench-security.sh -c host_configuration
```

Securing The Docker Host

This will only run the host configuration checks and benchmarks as highlighted in the image below.



```
Section A  - Check results

[INFO] 1 - Host Configuration
[INFO] 1.1 Linux Hosts Specific Configuration
WARNING: No swap limit support
WARNING: No blkio weight support
WARNING: No blkio weight_device support
[WARN] 1.1.1 Ensure a separate partition for containers has been created (Automated)
[INFO] 1.1.2 Ensure only trusted users are allowed to control Docker daemon (Automated)
[INFO]      * Users: alexis
[WARN] 1.1.3 Ensure auditing is configured for the Docker daemon (Automated)
[WARN] 1.1.4 Ensure auditing 1s configured for Docker files and directories /run/containerd (Automated)
[WARN] 1.1.5 Ensure auditing 1S configured for Docker files and directories /var/lib/docker (Automated)
[WARN] 1.1.6 Ensure auditing is configured for Docker files and directories /etc/docker (Automated)
```

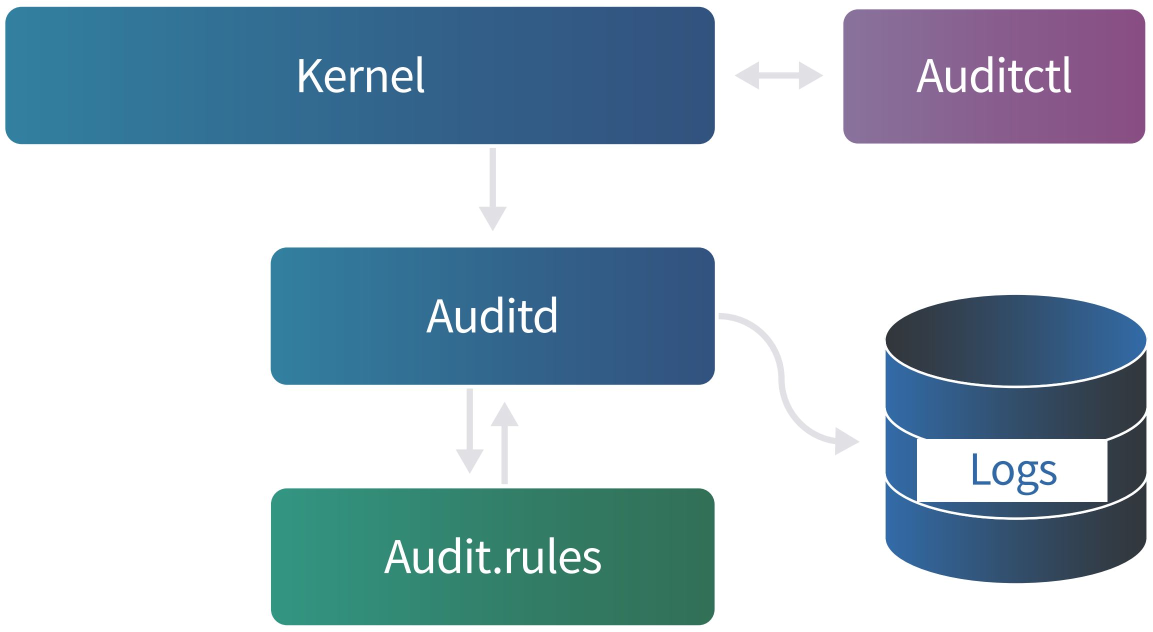
The script should output a list of Docker artifacts that require audit rules. Before we can enable auditing of these artifacts, we need to further explore the concept of auditing files and objects on Linux systems.

File and object auditing allows us to log and analyze all the activity of an object. Auditing on Linux is facilitated through the Linux Audit Framework. In the context of auditing, an object is a system resource like a file, directory, application, or service. Docker requires us to have audit rules for core artifacts, like the Docker daemon, in order to ensure that all activity from these artifacts is logged for security purposes.

Securing The Docker Host

THE LINUX AUDIT FRAMEWORK

The Linux Audit Framework is used to set up and configure auditing policies for user-space processes like Docker. The following diagram outlines the various components that make up the Linux Audit Framework and how they interact with each other:



Here's an overview of some of The Linux Audit Framework components:

- **Auditd**: The Audit daemon. This saves audit events to the audit log.
- **Audit Log**: Contains event logs from all configured audit rules.
- **Auditctl:** Client software that is used to manage and control the framework and is also used to create or delete audit rules.
- **Audit.rules:** A configuration file that contains audit rules and is accessed by auditd when the service is restarted.

All auditing is handled by the Linux kernel. Whenever a system call is made by a user-space service like Docker, the kernel will check the audit policy to determine whether the service in question has any audit rules. If it does, it will send the audit event to Auditd, and consequently, Auditd will send the event log to the audit.log for storage and analysis. Tools like aureport can be used to perform the analysis.

When Auditd is started or restarted, it will load the audit rules saved in the audit.rules file. We will take a look at how to create audit rules in the upcoming sections.

Securing The Docker Host

INSTALLING AUDITD

Auditd comes pre-installed on most Linux distributions. However, if you need to install it manually, you can use your distribution package manager. On Ubuntu and Debian based distributions, Auditd can be installed by running the following command:

```
sudo apt-get install auditd
```

CREATING AUDIT RULES FOR DOCKER ARTIFACTS

The host configuration security audit we performed with Docker Bench for Security provided a list of Docker artifacts that require auditing. We can create audit rules for these artifacts by running the following `auditctl` command:

```
sudo auditctl -w <path to artifact> -k docker
```

The arguments in this command have the following functions:

Argument	Function
-w	Used to specify the file or service to watch.
-k	Used to specify the filter key, which is a short string of text. The same key can be applied to several different audit rules. By applying a key, you can group different rules together, which can be useful when analyzing and searching through your audit logs.

We need to create audit rules for all the artifacts listed in the audit results from the Docker Bench for Security utility:

1. In the previous Docker Bench for Security report, warnings like the following appeared:

```
Ensure auditing is configured for Docker files and directories
- /etc/docker
```

2. For this warning, create a corresponding audit rule with a command like this:

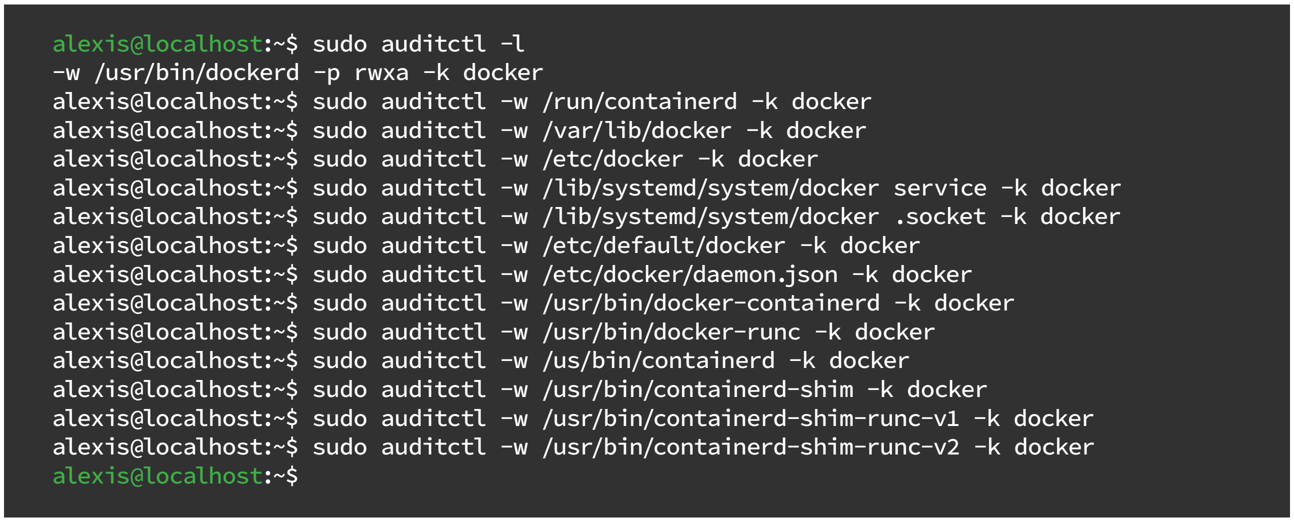
```
sudo auditctl -w /etc/docker -k docker
```

Securing The Docker Host

3. Create audit rules for each such warning. After creating the audit rules, we can list them by running the following command:

```
sudo auditctl -l
```

4. This will list out all the created audit rules for the Docker artifacts. You should have similar rules to the ones highlighted in the image below:



```
alexis@localhost:~$ sudo auditctl -l
-w /usr/bin/dockerd -p rwxa -k docker
alexis@localhost:~$ sudo auditctl -w /run/containerd -k docker
alexis@localhost:~$ sudo auditctl -w /var/lib/docker -k docker
alexis@localhost:~$ sudo auditctl -w /etc/docker -k docker
alexis@localhost:~$ sudo auditctl -w /lib/systemd/system/docker service -k docker
alexis@localhost:~$ sudo auditctl -w /lib/systemd/system/docker .socket -k docker
alexis@localhost:~$ sudo auditctl -w /etc/default/docker -k docker
alexis@localhost:~$ sudo auditctl -w /etc/docker/daemon.json -k docker
alexis@localhost:~$ sudo auditctl -w /usr/bin/docker-containerd -k docker
alexis@localhost:~$ sudo auditctl -w /usr/bin/docker-runc -k docker
alexis@localhost:~$ sudo auditctl -w /us/bin/containerd -k docker
alexis@localhost:~$ sudo auditctl -w /usr/bin/containerd-shim -k docker
alexis@localhost:~$ sudo auditctl -w /usr/bin/containerd-shim-runc-v1 -k docker
alexis@localhost:~$ sudo auditctl -w /usr/bin/containerd-shim-runc-v2 -k docker
alexis@localhost:~$
```

5. After creating the rules, we need to save them to the audit.rules file to make them permanent. This can be done by copying and pasting the audit rules from the output of the `sudo auditctl -l` command to the `audit.rules` file located in:

```
/etc/audit/rules.d/audit.rules
```

6. After adding the rules to the `audit.rules` file, you will need to restart the auditd service. This can be done by running the following command:

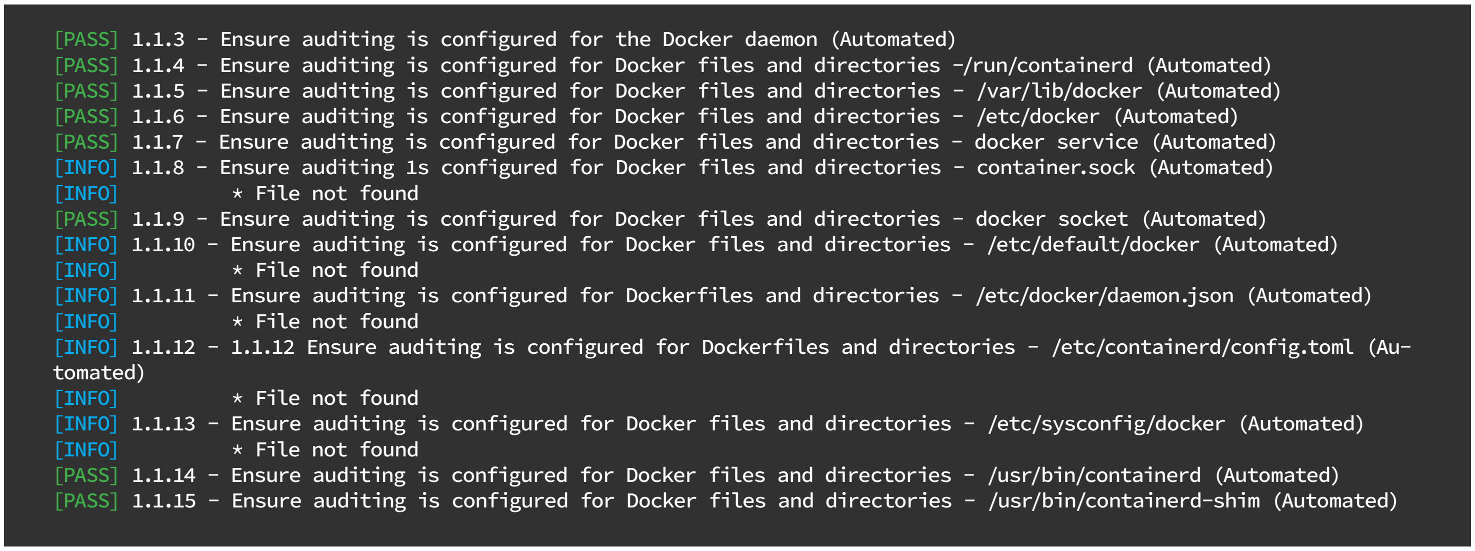
```
sudo systemctl restart auditd
```

7. After restarting auditd, we can re-run the Docker Bench for Security tool to confirm that the audit rules have been enabled and are active:

```
cd ~/docker-bench-security/
sudo ./docker-bench-security.sh -c host_configuration
```

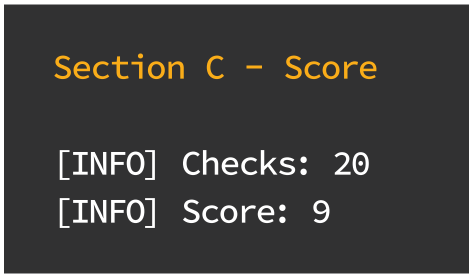
Securing The Docker Host

8. As illustrated in the image below, the host configuration checks related to the auditing of Docker artifacts should all be successful:



```
[PASS] 1.1.3 - Ensure auditing is configured for the Docker daemon (Automated)
[PASS] 1.1.4 - Ensure auditing is configured for Docker files and directories -/run/containerd (Automated)
[PASS] 1.1.5 - Ensure auditing is configured for Docker files and directories - /var/lib/docker (Automated)
[PASS] 1.1.6 - Ensure auditing is configured for Docker files and directories - /etc/docker (Automated)
[PASS] 1.1.7 - Ensure auditing is configured for Docker files and directories - docker service (Automated)
[INFO] 1.1.8 - Ensure auditing 1s configured for Docker files and directories - container.sock (Automated)
[INFO]         * File not found
[PASS] 1.1.9 - Ensure auditing is configured for Docker files and directories - docker socket (Automated)
[INFO] 1.1.10 - Ensure auditing is configured for Docker files and directories - /etc/default/docker (Automated)
[INFO]         * File not found
[INFO] 1.1.11 - Ensure auditing is configured for Dockerfiles and directories - /etc/docker/daemon.json (Automated)
[INFO]         * File not found
[INFO] 1.1.12 - 1.1.12 Ensure auditing is configured for Dockerfiles and directories - /etc/containerd/config.toml (Au-
tomated)
[INFO]         * File not found
[INFO] 1.1.13 - Ensure auditing is configured for Docker files and directories - /etc/sysconfig/docker (Automated)
[INFO]         * File not found
[PASS] 1.1.14 - Ensure auditing is configured for Docker files and directories - /usr/bin/containerd (Automated)
[PASS] 1.1.15 - Ensure auditing is configured for Docker files and directories - /usr/bin/containerd-shim (Automated)
```

We should also have a new audit score that reflects the audit rules we have created as shown in the image below:



```
Section C - Score

[INFO] Checks: 20
[INFO] Score: 9
```

Now that we have been able to successfully secure our Docker host, we can begin the process of securing the Docker daemon.

Securing the Docker Daemon

Now that we have a secure Docker host to work with, we can begin the process of securing the Docker daemon from the recommendations provided by the Docker Bench for Security tool.

The components that we need to implement are:

- TLS encryption between the Docker client and daemon
- User namespaces

We will begin the process by taking a look at how to implement TLS encryption between the Docker client and daemon.

IMPLEMENTING TLS ENCRYPTION

As mentioned earlier in this guide, communication between the Docker client and daemon can be performed locally through a unix domain socket or remotely through the use of a TCP socket. This communication is not encrypted by default and, as a result, an attacker can perform a man in the middle (MITM) attack and can intercept the commands being sent remotely from the Docker client to the Docker daemon.

We can remedy this situation by implementing TLS encryption for remote connections. This process is twofold and involves generating the TLS certificates for the server and the remote clients. We will begin by taking a look at how to generate the TLS certificates for both the Docker client and server, and then we will update the Docker daemon configuration to use the certificates.

GENERATING TLS CERTIFICATES

The process of generating TLS certificates manually can be slightly complicated. As a result, we will be using an automated Bash script to generate the certificates for us. This can be done by following the procedures outlined below:

1. The first step in the process involves downloading the Bash script. This can be done by running the following commands:

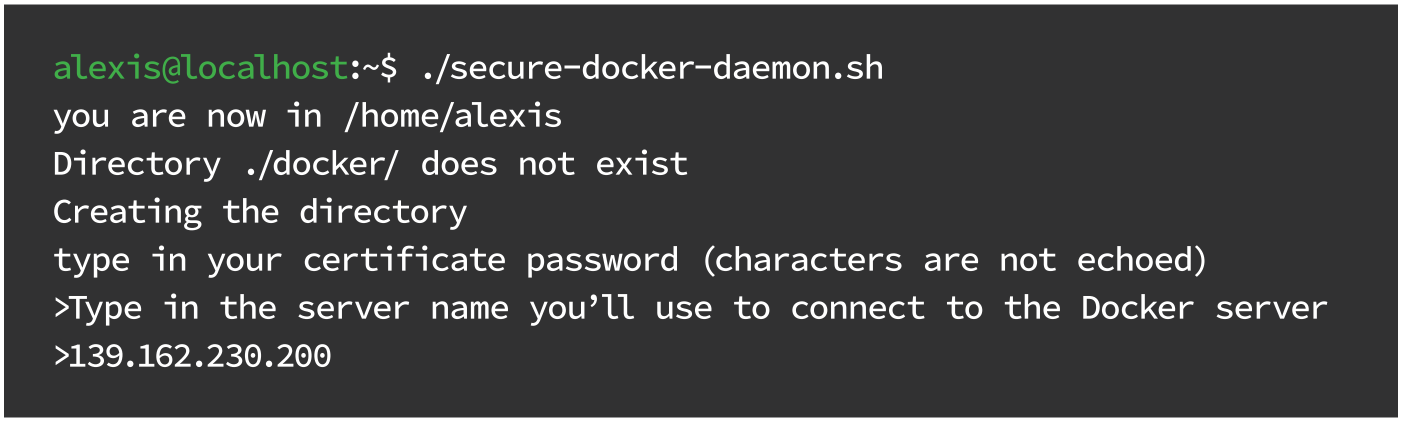
```
cd ~
wget https://raw.githubusercontent.com/AlexisAhmed/
DockerSecurityEssentials/main/Docker-TLS-Authentication/secure-
docker-daemon.sh
chmod u+x secure-docker-daemon.sh
```

Securing the Docker Daemon

2. After downloading the script, we can execute it by running the following command:

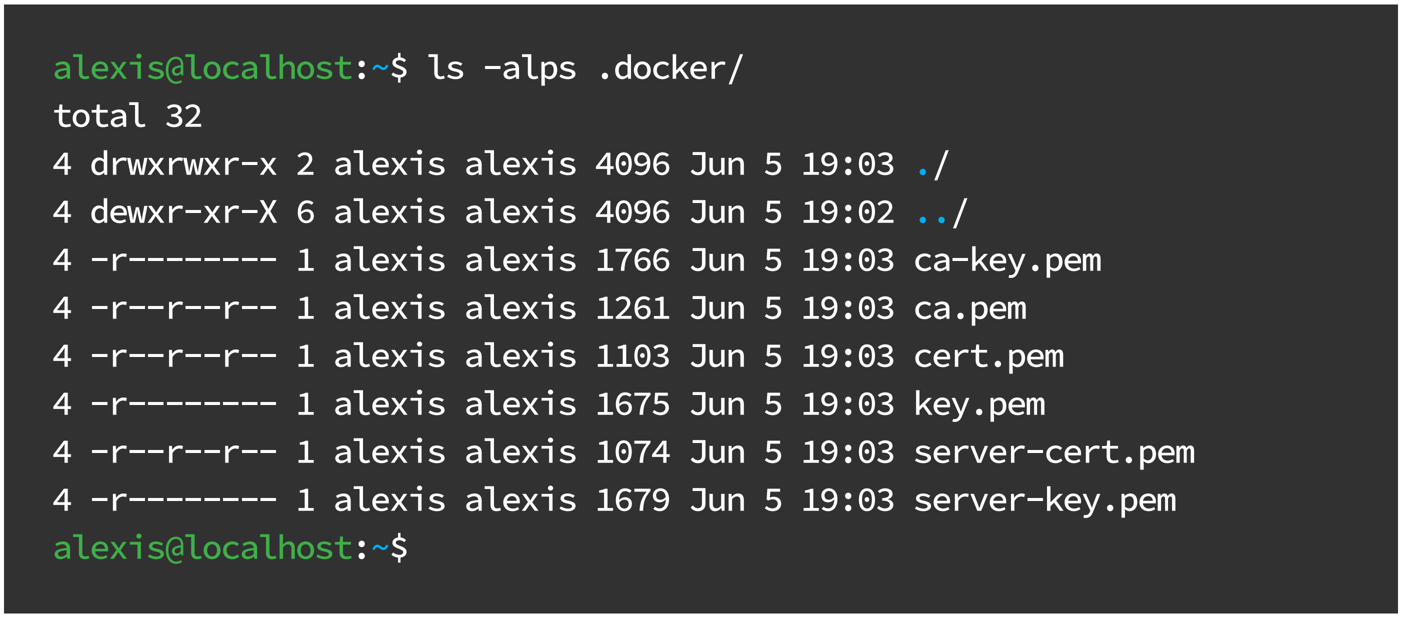
   ```
   ./secure-docker-daemon.sh
   ```

   The script will create a `.docker/` directory in your user's home directory as illustrated in the image below. This is where the certificates will be stored.



```
alexis@localhost:~$ ./secure-docker-daemon.sh
you are now in /home/alexis
Directory ./docker/ does not exist
Creating the directory
type in your certificate password (characters are not echoed)
>Type in the server name you'll use to connect to the Docker server
>139.162.230.200
```

3. The script will prompt you to enter the Docker server IP. After providing the IP address, the script will automatically create the client and server certificates in the `.docker/` directory as highlighted in the image below:



```
alexis@localhost:~$ ls -alps .docker/
total 32
4 drwxrwxr-x 2 alexis alexis 4096 Jun 5 19:03 ./
4 dewxr-xr-X 6 alexis alexis 4096 Jun 5 19:02 ../
4 -r-------- 1 alexis alexis 1766 Jun 5 19:03 ca-key.pem
4 -r--r--r-- 1 alexis alexis 1261 Jun 5 19:03 ca.pem
4 -r--r--r-- 1 alexis alexis 1103 Jun 5 19:03 cert.pem
4 -r-------- 1 alexis alexis 1675 Jun 5 19:03 key.pem
4 -r--r--r-- 1 alexis alexis 1074 Jun 5 19:03 server-cert.pem
4 -r-------- 1 alexis alexis 1679 Jun 5 19:03 server-key.pem
alexis@localhost:~$
```

Securing the Docker Daemon

DOCKER DAEMON CONFIGURATION

After generating the TLS certificates, we now need to create a custom systemd configuration file for the Docker daemon. This configuration file will be used to enable TLS and specify the TLS certificates.

1. We can create the custom systemd file with a text editor and add the TLS configuration to it. Create the systemd file in your preferred text editor with a command like the following:

   ```
   sudo mkdir /etc/systemd/system/docker.service.d/
   sudo vim /etc/systemd/system/docker.service.d/override.conf
   ```

2. After creating and opening the file in your editor, add the following configuration to it. When pasting this snippet into your file, be sure to replace the `<user>` string with the username on your system:

   ```
   [Service]
   ExecStart=
   ExecStart=/usr/bin/dockerd -D -H unix:///var/run/docker.sock
   --tlsverify  --tlscert=/home/<user>/.docker/server-cert.pem
   --tlscacert=/home/<user>/.docker/ca.pem --tlskey=/home/<user>/.
   docker/server-key.pem  -H tcp://0.0.0.0:2376
   ```

3. After adding the configuration to the file, we need to save it and restart the Docker service. This can be done by running the following command:

   ```
   sudo systemctl restart docker
   ```

4. If the configuration file has been created and set up correctly, the Docker service should restart with no issues.

5. You can now copy over the client TLS certificates to the remote Docker client for authentication. We will not be covering this process in detail as it is beyond the scope of this guide. More information regarding TLS authentication can be found here: https://docs.docker.com/engine/security/protect-access/

Now that we have configured TLS encryption between the Docker client and daemon, we can move on to implementing user namespaces for containers.

Securing the Docker Daemon

IMPLEMENTING USER NAMESPACES

After generating the TLS certificates, we need to create a custom systemd configuration file for the Docker daemon. This configuration file will be used to enable TLS and specify the TLS certificates.

When we run a Docker container, the process is run from the default namespace. As a result, the process is run under the root user as highlighted in the image below:

```
alexis@localhost:~$ docker container top test
UID           PID            PPID
D
root          17782          17762
in/bash
alexis@localhost:~$
```

This can be dangerous in the event of a container breakout. Because the process is being run as the root user, an attacker would be able to get root privileges for the host. As a result, we need to run containers as an unprivileged user.

We need to reconfigure the Docker daemon to use user namespaces. Docker generates a default dockremap user that you can use, or you can specify your own non-privileged user.

1. We can implement user namespaces by adding the following option to the `ExecStart` line in the /`etc/systemd/system/docker.service.d/override.conf` file we created in the previous section:

   ```
   --userns-remap="default"
   ```

2. Your new configuration should be structured as follows::

   ```
   [Service]
   ExecStart=
   ExecStart=/usr/bin/dockerd -D -H unix:///var/run/docker.sock
   --tlsverify  --tlscert=/home/<user>/.docker/server-cert.pem
   --tlscacert=/home/<user>/.docker/ca.pem --tlskey=/
   home/&lt;user&gt;/.docker/server-key.pem --userns-remap="default"
   -H tcp://0.0.0.0:2376
   ```

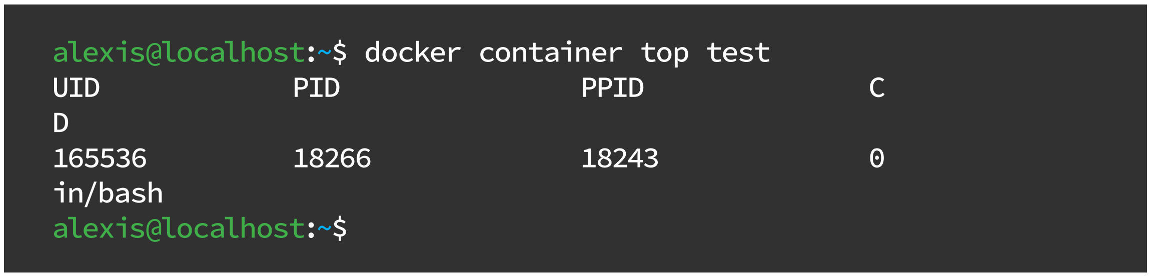
3. After saving the configuration, you will need to restart the Docker service. This can be done by running the following commands:

```
sudo systemctl daemon-reload
sudo systemctl restart docker
```

4. We can now confirm that containers will run under the default `dockremap` UID. Run a container and give it the name `test`. Then inspect the container process with this command:

```
docker container top test
```

5. The output from this command will resemble the following:



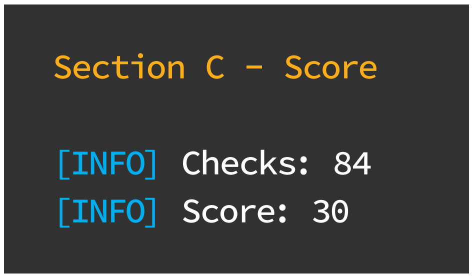
```
alexis@localhost:~$ docker container top test
UID             PID               PPID              C
D
165536          18266             18243             0
in/bash
alexis@localhost:~$
```

RUNNING DOCKER BENCH FOR SECURITY AFTER SECURING THE DOCKER DAEMON

Now that we have implemented TLS encryption and user namespaces for containers, we can re-run our security audit with Docker Bench for Security:

```
cd ~/docker-bench-security/
sudo ./docker-bench-security.sh -c host_configuration
```

As highlighted in the image below, we should now have an improved security score which highlights the changes we have been making.



```
Section C - Score

[INFO] Checks: 84
[INFO] Score: 30
```

We have been able to successfully secure the Docker daemon and can begin exploring the various ways of securing Docker containers.

Container Security Best Practices

Now that we have a secure Docker host and daemon, we can shift our attention to running containers securely. The process of running containers securely is quite robust and will depend on your own use cases. As a result, we have structured this section as a list of security best practices that you can use when running containers based on your own security requirements.

USING AN UNPRIVILEGED USER

Running containers with an unprivileged user will prevent privilege escalation attacks. This can be done by following the outline below:

1. Always reconfigure and build your own Docker images so you can customize the various security parameters to your specification.

2. To run a Docker container as an unprivileged user, you will need to update the Dockerfile before building the image. This can be done by adding a command like the following example to the Dockerfile:

```
RUN groupadd -r <user> && useradd -r -g <group> <user>
```

```
FROM ubuntu: 18.04

LABEL maintainer=”Alexis Ahmed”

RUN groupadd -r alexis && useradd -r -g alexis alexis

# Environment Variables
ENV HOME /home/alexis
ENV DEBIAN_FRONTEND=noninteractive
```

3. This will add the user to the Docker image, and you can now run the container with the unprivileged user instead of running it with the default root user. You can specify the user for a container with the `-u` option for the `docker run` command:

```
docker run -u <user> <IMAGE-ID>
```

```
root@localhost:~# docker run -u alexis -it --rm cbalba6cb267 /bin/bash
alexis@3485dfe797b4:/$
```

Container Security Best Practices

DISABLING THE ROOT USER

As an added security measure, we can disable the root user of a container by modifying the Dockerfile. Specifically, we can change the default shell from `/bin/bash` to `/usr/sbin/nologin`. This can be done by adding the following command to the Dockerfile:

```
RUN chsh -s /usr/sbin/nologin root
```

This will prevent any user on the container from accessing the root account regardless of whether they have the root password. This configuration is only applicable if you want to disable the root account completely.

PREVENTING PRIVILEGE ESCALATION ATTACKS

It is recommended to run your containers with specific permissions and ensure that users cannot escalate their privileges. To do this, use the following flag when running containers:

```
docker run --security-opt=no-new-privileges <IMAGE-ID>
```

The `no-new-privileges` option will stop container processes from gaining any additional privileges. This will prevent commands like `su` and `sudo` from working in your container, and it can prevent attacks that exploit SETUID binaries.

LIMITING CONTAINER CAPABILITIES

When you run a container, you can specify a set of kernel capabilities that are available to the container. For example, a container can be given the capability of binding to a low-number port on the host (e.g. a web server container that binds to ports 80 and 443). You also can run a container with the `--privileged` flag, which gives it all of the kernel capability options. However, this is never recommended as giving full privileges to a container will usurp any other user permission and security restrictions you have set and open new vulnerabilities.

The recommended method for assigning privileges to a container is to first remove all of the capabilities (also referred to as *dropping* the capabilities) and then only add the ones required for your container to function. If your container does not need kernel capabilities to run, then they should all be dropped.

Container Security Best Practices

1. We can remove all kernel capabilities when running a container with the following options:

```
docker run --cap-drop all <IMAGE-ID>
```

2. You can also add the specific kernel capabilities required by your containers by running the following command:

```
docker run --cap-drop all --cap-add <CAPABILITY> <IMAGE-ID>
```

```
root@localhost:~# docker run --cap-drop all --cap-add NET_ADMIN -it
--rm -u alexis c51c05657e9f /bin/bash
alexis@b377978845d0:/$
```

FILESYSTEM PERMISSIONS AND ACCESS

You also have the ability to specify filesystem permissions and access, allowing you to set up containers with a read only file system or with a temporary file system. This option is useful if you would like to control whether your containers can store data or make changes to the filesystem.

1. We can run containers with a read-only file system by running the following command:

```
docker run --read-only <DOCKER-ID>
```

```
alexis@a7e5ee193740:/home$ touch test
touch: cannot touch 'test': Read-only file system
alexis@a7e5ee193740:/home$
```

2. If your container has a service or application that requires the storage of data, you can specify a temporary file system by running the following command:

```
docker run --read-only --tmpfs /tmp <DOCKER-ID>
```

Container Security Best Practices

DISABLING INTER-CONTAINER COMMUNICATION

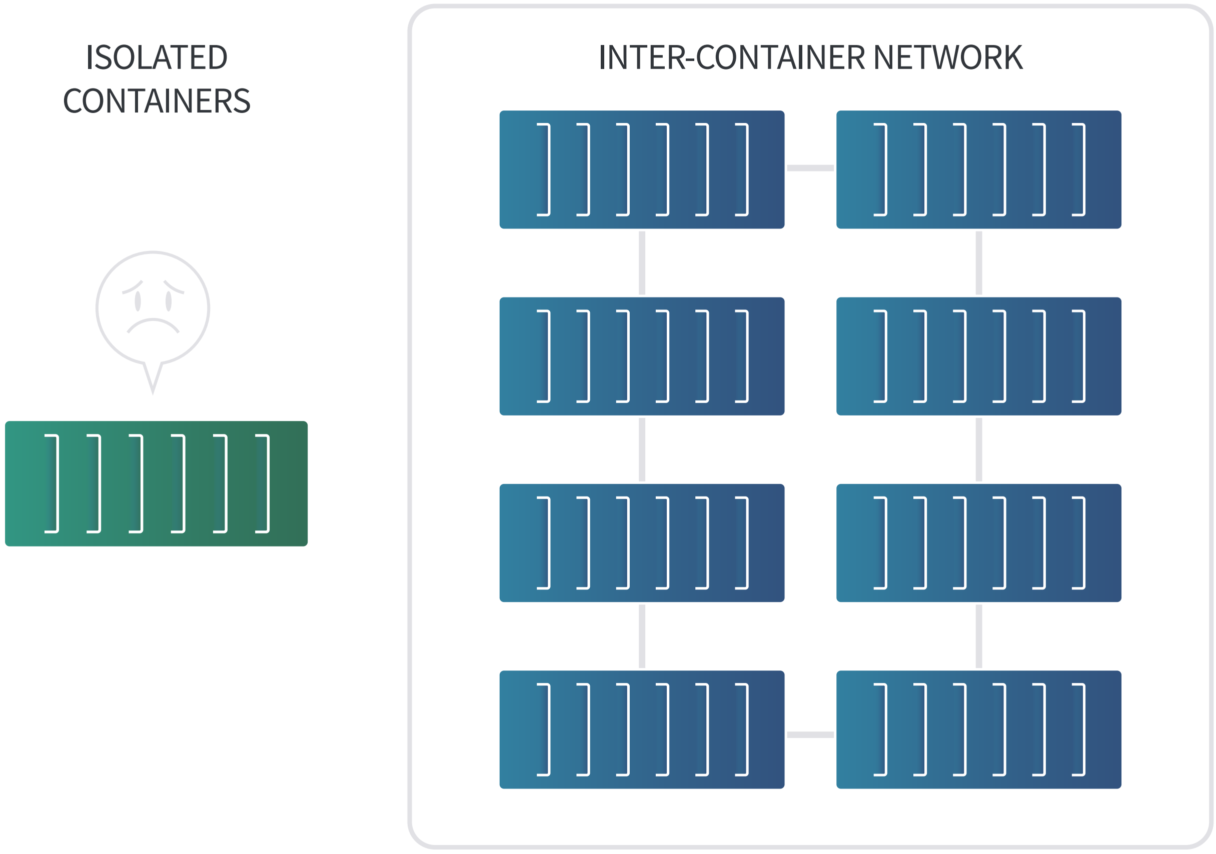
Docker creates a default bridge network, and containers are created on this network by default. All containers on this default network can communicate with each other. However, we can also choose to isolate Docker containers from communicating with one another. For example, this can be helpful if you want to isolate a particular Docker container away from another connected group of containers you're running.

1. In order to disable inter-container communication, we will need to create a new Docker network. This can be done by running the following command with the "icc" option set to false.

```
docker network create --driver bridge -o
"com.docker.network.bridge.enable_icc"="false" <NETWORK-NAME>
```

2. You can also add the specific kernel capabilities required by your containers by running the following command:

```
docker run --network <NETWORK-NAME> <IMAGE-ID>
```



Controlling Container Resource Consumption with Control Groups (cgroups)

Control groups (cgroups) are a feature of the Linux kernel that are used to isolate, limit, and account for resource usage on a system for a set of processes. Control groups are used to isolate CPU, memory, network, and disk usage. As with namespaces, cgroups are a kernel feature that is essential to the Docker platform and container technologies generally. For your applications, controlling resource consumption is important for a number of reasons:

- Some applications consume a high amount of resources and need to be managed with respect to the host's performance.
- Containers need to operate in a manner conducive to, and respective of, the performance of other containers.
- Together, these controls help you optimize container performance generally.
- If a container is compromised, an attacker could utilize the resources of the container for CPU intensive processes like cryptocurrency mining. Control groups can limit the impact of these exploits.

By default, control groups are managed and maintained by the host's init system, which in most cases will be systemd. Docker utilizes cgroupfs (cgroup file system) to manage and maintain the control groups associated with containers. Docker provides you with the ability to allocate resources for all containers system-wide or on a container-by-container basis.

Now that we have an understanding of what control groups are and what they are used for, we can explore the various control group *subsystems* utilized by Docker.

Controlling Container Resource Consumption with Control Groups (cgroups)

CONTROL GROUP SUBSYSTEMS

Subsystems, also known as resource controllers, are used to manage and limit the usage of a specific resource on a system. The following is a list of subsystems utilized by Docker to control resource consumption:

- cpu: Manages and controls access to CPU usage.
- cpuset: Assigns processes to CPU cores.
- memory: Controls and monitors memory usage.
- pids: Limits the number of processes in a control group.
- blkio: Controls block I/O operations.
- devices: Controls access to devices.

We can use these subsystems to limit and control container resource consumption at runtime.

1. To limit container CPU usage, add the `--cpus` argument when running a container:

   ```
   docker run -it --rm --cpus 0.25 <image name> /bin/bash
   ```

2. To limit containers to use only certain CPU cores, add the --*cpuset-cpu* argument when running a container:

   ```
   docker run -it --rm --cpuset-cpus=0 <image name> /bin/bash
   ```

   In the above example, we are limiting the container to use only the first CPU core. If your Docker host has multiple cores, you can specify more than one core to use by including a comma-separated list of core numbers as follows:

   ```
   docker run -it --rm --cpuset-cpus=0,1 <image name> /bin/bash
   ```

   You could also supply a range of core numbers as follows:

   ```
   docker run -it --rm --cpuset-cpus=0-3 <image name> /bin/bash
   ```

Controlling Container Resource Consumption with Control Groups (cgroups)

3. In addition to controlling CPU usage, we can also limit the memory consumption of a Docker container by adding the `-m` argument when running a container::

```
docker run -it --rm -m 128m <image name> /bin/bash
```

In this example, we have limited the container to 128 MB of RAM usage. You can specify a minimum of 4MB of RAM for each container.

4. Docker also provides users with the ability to specify a limit on the number of processes that a container can fork. This can be very helpful in limiting the container to specific services and can prevent fork bomb denial-of-service attacks. To limit the number of processes, use the `--pids-limit` argument when running a container:

```
docker run -it --rm --pids-limit 5 <image name> /bin/bash
```

In this example, we have limited the container to a maximum of 5 PIDs.

5. You also have the ability to impose resource restrictions on containers that are already running. This can be done by running the docker update command, as in this example:

```
docker update --cpus 0.25 <CONTAINER-ID>
```

In the above example, we have limited the CPU usage of a container to 25% of available CPU processing power.

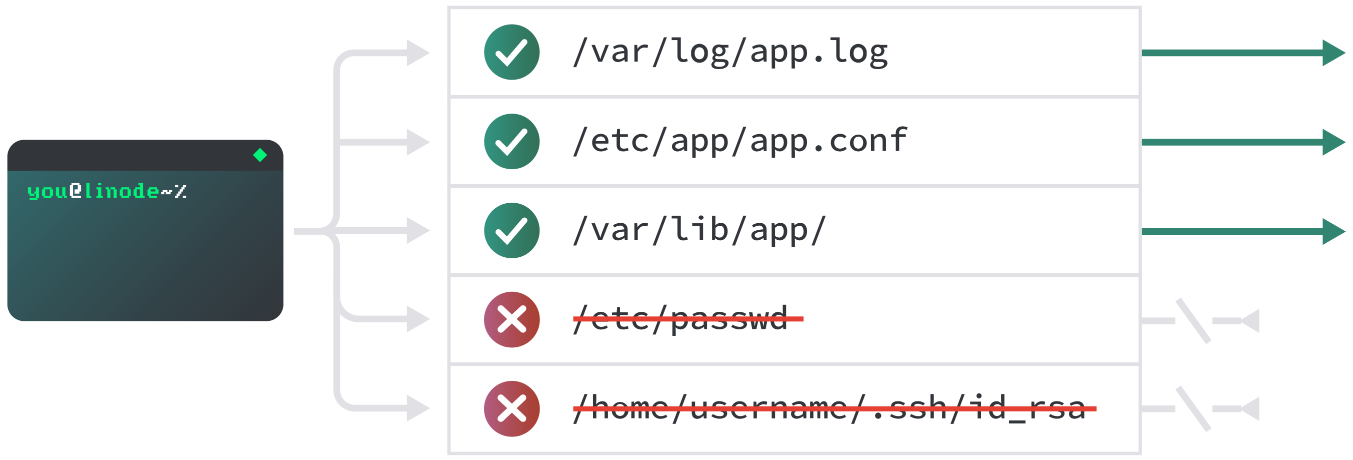
Now that we have an idea of how to control container resource consumption, we can take a look at how to implement access control for Docker containers with AppArmor.

Implementing Access Control with AppArmor

Access control is the process of managing and controlling access to a system resource. In the context of containers, we need to configure what system resources and functionality the containers can access.

Linux implements access control in two forms:

1. **Discretionary Access Control:** Access to resources is specified by the resource owner. An example of this is the implementation of file and directory permissions. This type of access control does not offer much in regard to the types of resources we can restrict access to.
2. **Mandatory Access Control:** Access to resources is dependent on predefined access policies. Examples of solutions that provide mandatory access control are SELinux and AppArmor.



We can manage the system resources that containers have access to with access control systems like AppArmor.

WHAT IS APPARMOR

AppArmor (Application Armor) is a Linux security module that is used to manage access to OS resources by utilizing custom profiles for applications and containers. AppArmor is quite extensive and can be used to restrict access to networking and specified file paths. Mandatory Access Control solutions like AppArmor are implemented into the Linux kernel as security modules.

Implementing Access Control with AppArmor

AppArmor is the default Mandatory Access Control system implemented in Debian-based distributions like Ubuntu, whereas SELinux is implemented on Fedora and RedHat based distributions.

In the context of Docker, we can use AppArmor to secure containers by restricting the resources and functionality they have access to. Docker runs containers with a default AppArmor profile that provides a good level of protection for most cases. However, it is recommended to create your own AppArmor profile based on your requirements and constraints.

If AppArmor is enabled for your host, Docker will utilize the default profile. You can also opt to run containers with no AppArmor profile, but running containers with no AppArmor profile is considered dangerous and should not be done in a production environment.

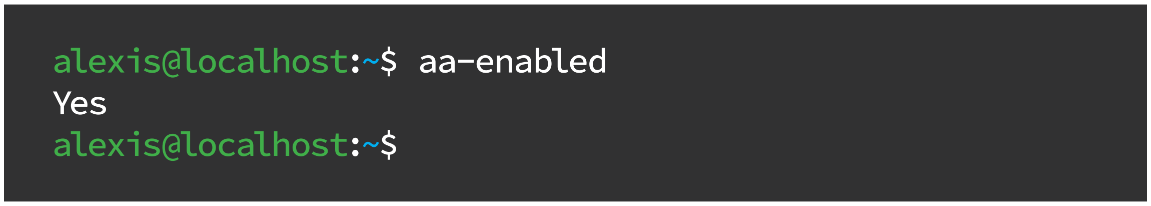
Generating a custom AppArmor profile can be tedious and time-consuming and will require a good understanding of the requirements of the container. For this reason, we will be utilizing Bane, an open source tool that automates the process of generating custom AppArmor profiles. More information regarding Bane can be found here: https://github.com/genuinetools/bane

CONFIRMING APPARMOR IS ENABLED

Before we can start generating and using custom AppArmor profiles, we need to ensure that AppArmor is installed and enabled. This can be done by running the following command:

```
aa-enabled
```

If AppArmor is enabled, you should receive the response text Yes, as illustrated in the figure below.



```
alexis@localhost:~$ aa-enabled
Yes
alexis@localhost:~$
```

After confirming that AppArmor is installed and enabled, you can explore the contents of the AppArmor configuration directory found under `/etc/apparmor.d/`. This is the recommended directory for storage of custom AppArmor profiles and other configurations. We will be creating our AppArmor configuration files in this directory.

Implementing Access Control with AppArmor

INSTALLING BANE

Bane can be installed by following the procedures outlined below:

1. In order to install Bane, we can use the automated installer for Linux. This can be done by running the following commands:

   ```
   export
   BANE_SHA256="69df3447cc79b028d4a435e151428bd85a816b3e26199cd010c74b7
   a17807a05"
   ```

   ```
   sudo curl -fSL "https://github.com/genuinetools/bane/releases/
   download/v0.4.4/bane-linux-amd64" -o "/usr/local/bin/bane" \
   && echo "${BANE_SHA256}  /usr/local/bin/bane" | sha256sum -c - \
   && sudo chmod a+x "/usr/local/bin/bane"
   ```

   NOTE: The above commands reference Bane's x86 executable for Linux, version v0.4.4. Check the releases page for Bane on GitHub for other architectures, operating systems, or any newer releases that are available: https://github.com/genuinetools/bane/releases

2. After running the installer script, we can confirm that Bane is installed by running the following command:

   ```
   bane -h
   ```

Now that we have confirmed that Bane is installed and enabled, we can take a look at how to create custom AppArmor profiles.

CREATING A CUSTOM APPARMOR PROFILE WITH BANE

Creating a custom AppArmor profile requires a good understanding of the resources that your containers need to access. The scope of this guide is limited to exploring the structure of an AppArmor profile and how to use the profile when running containers. The exact details of your profiles will require further investigation.

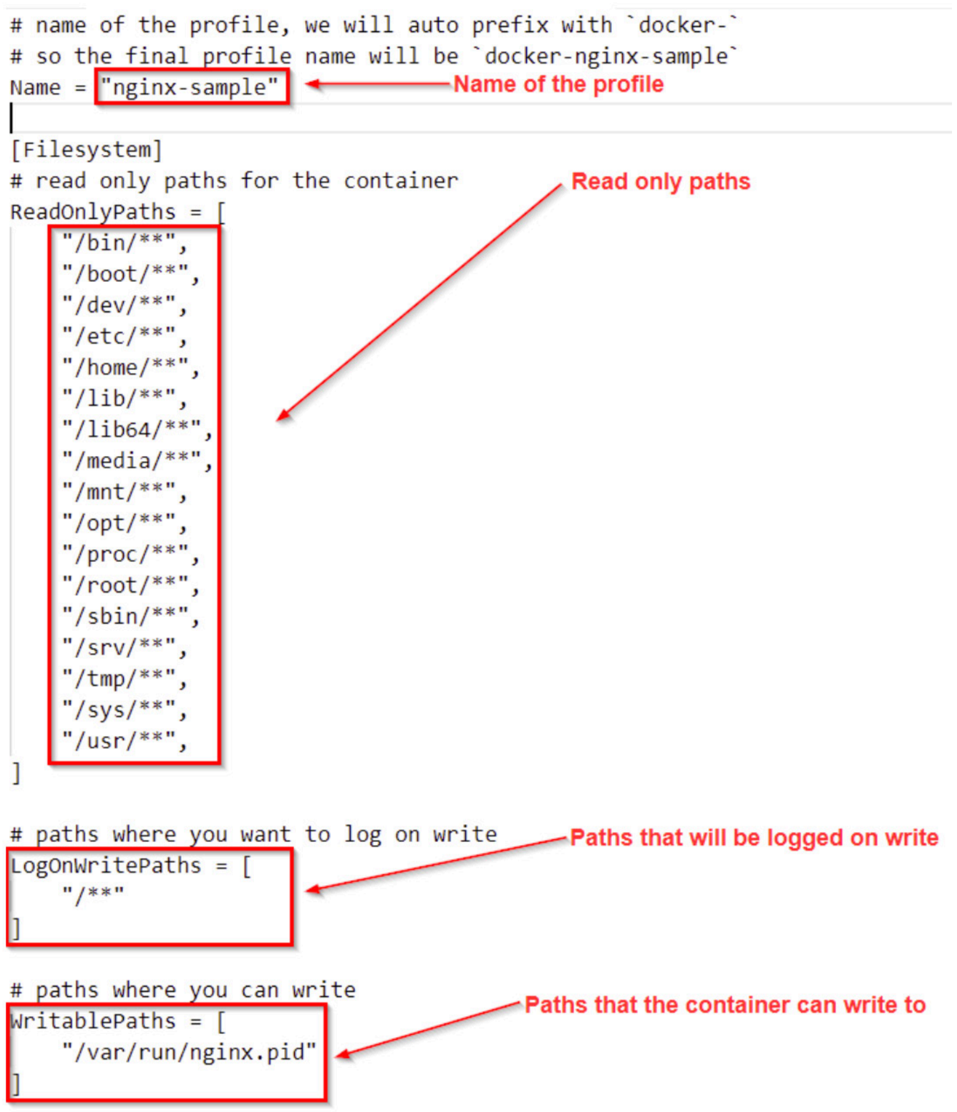
Implementing Access Control with AppArmor

In this section, we will start off with an AppArmor template available on the Bane GitHub repository. You can then customize and modify this profile as needed to meet the requirements of your container.

1. You can access and download the AppArmor profile template from GitHub: https://github.com/genuinetools/bane/blob/master/sample.toml

   The above commands reference Bane's x86 executable In order to modify and generate the AppArmor profile with Bane, you need to download the sample AppArmor template to the AppArmor configuration directory found under `/etc/apparmor.d/`.

2. The following image highlights the sections of the AppArmor template that you will likely need to modify:

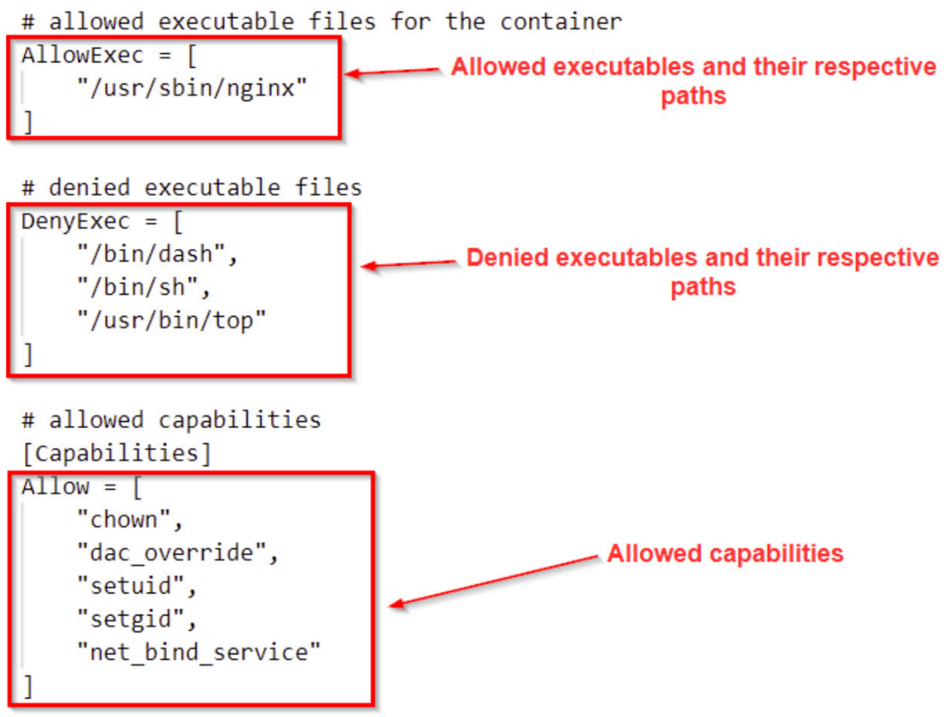


```
# name of the profile, we will auto prefix with `docker-`
# so the final profile name will be `docker-nginx-sample`
Name = "nginx-sample"

[Filesystem]
# read only paths for the container
ReadOnlyPaths = [
    "/bin/**",
    "/boot/**",
    "/dev/**",
    "/etc/**",
    "/home/**",
    "/lib/**",
    "/lib64/**",
    "/media/**",
    "/mnt/**",
    "/opt/**",
    "/proc/**",
    "/root/**",
    "/sbin/**",
    "/srv/**",
    "/tmp/**",
    "/sys/**",
    "/usr/**",
]

# paths where you want to log on write
LogOnWritePaths = [
    "/**"
]

# paths where you can write
WritablePaths = [
    "/var/run/nginx.pid"
]
```

Name of the profile
Read only paths
Paths that will be logged on write
Paths that the container can write to

Implementing Access Control with AppArmor

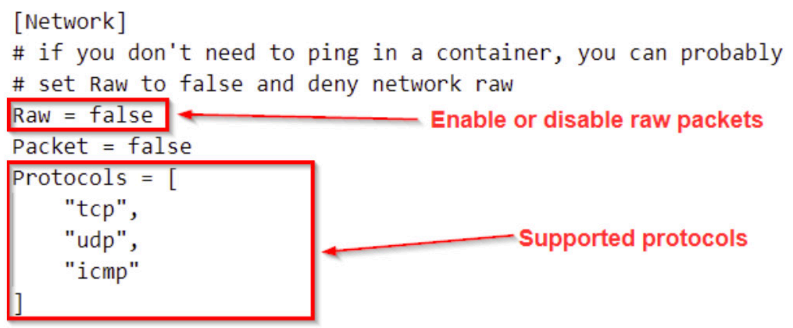
3. In particular, you should provide the profile with a unique name as highlighted in the preceding image. You can also specify the allowed executable binaries and capabilities as shown in the image below.



```
# allowed executable files for the container
AllowExec = [
    "/usr/sbin/nginx"
]

# denied executable files
DenyExec = [
    "/bin/dash",
    "/bin/sh",
    "/usr/bin/top"
]

# allowed capabilities
[Capabilities]
Allow = [
    "chown",
    "dac_override",
    "setuid",
    "setgid",
    "net_bind_service"
]
```

Allowed executables and their respective paths
Denied executables and their respective paths
Allowed capabilities

4. You can also control access to networking by specifying whether you want to enable raw packet connections. You also have the ability to specify the networking protocols that the container can use. These controls are depicted in the image below:



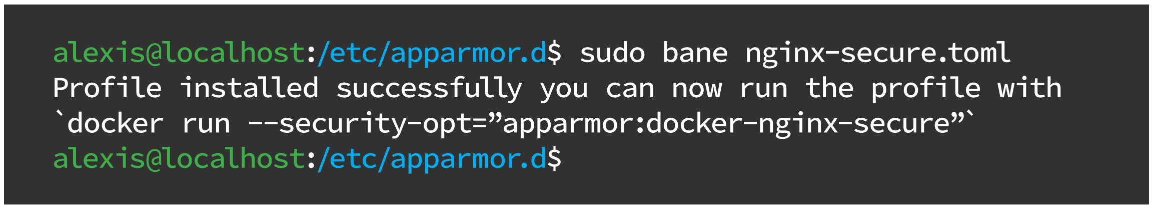
```
[Network]
# if you don't need to ping in a container, you can probably
# set Raw to false and deny network raw
Raw = false
Packet = false
Protocols = [
    "tcp",
    "udp",
    "icmp"
]
```

Implementing Access Control with AppArmor

5. After modifying the AppArmor profile as needed based on your container requirements, you can generate the AppArmor profile for Docker with Bane. This can be done by running the following command:

   ```
   sudo bane <profile-name>.toml
   ```

6. Bane will generate and install the profile for you and will also provide you with the Docker runtime security specification for the AppArmor profile as highlighted in the image below.



```
alexis@localhost:/etc/apparmor.d$ sudo bane nginx-secure.toml
Profile installed successfully you can now run the profile with
`docker run --security-opt=”apparmor:docker-nginx-secure”`
alexis@localhost:/etc/apparmor.d$
```

7. After generating the AppArmor profile with Bane, we can specify the AppArmor profile when running a container with the `--security-opt` argument as follows:

   ```
   docker run -d --security-opt=”apparmor:<profile-name>” <docker
   image>
   ```

   This will run the container with your custom AppArmor profile, and based on your profile, the container will be limited in terms of functionality and the resources it has access to.

RUNNING CONTAINERS WITHOUT AN APPARMOR PROFILE

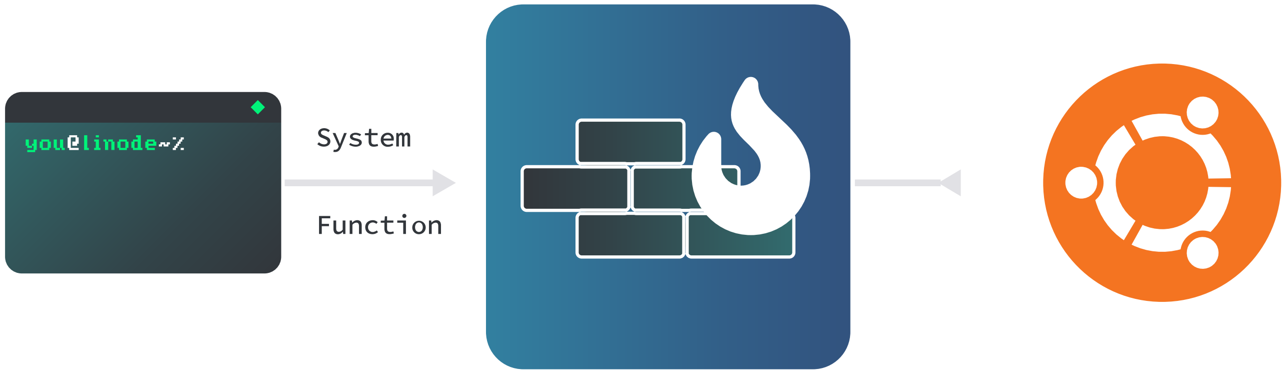
You also have the ability to run containers in an unconfined mode with no AppArmor profile specified. This is **not recommended** as the container will have access to any functionality and resources on the host. This can be done by running the following command:

```
docker run -d --security-opt=”apparmor:<profile-name>”
```

Now that we have an idea of how to generate and use custom AppArmor profiles, we can take a look at how to limit container system calls with *seccomp*.

Limiting Container System Calls with seccomp

Secure computing (seccomp) is a security feature in the Linux kernel that allows you to restrict the system calls that can be made by a process. In the context of containers, seccomp works like a firewall for system calls and can be used to restrict the system calls made by Docker containers.



WHAT IS A SYSTEM CALL?

A system call is the process through which a user-space process communicates with the Linux kernel in order to access resources or functionality. Whenever you want to create a file, change ownership or modify a network configuration, it is facilitated through the use of a system call.

WHY SHOULD YOU USE SECCOMP?

- Containers do not require the ability to make all available system calls in order to function as needed.
- In the event a container is compromised, the attacker can make various system calls that can lead to further exploitation of the Docker host.
- Reducing access to system calls greatly reduces the overall attack surface of a container.

Limiting Container System Calls with seccomp

USING SECCOMP FOR DOCKER CONTAINERS

Docker utilizes the seccomp filters to restrict system calls available to containers. Docker will utilize a default seccomp profile for Docker containers. However, you can also create a custom seccomp profile with your own configurations. Seccomp can be configured to run for all containers or a container-to-container basis. Lastly, you can run containers with no seccomp profile specified, leaving it unsecured, but this is not recommended.

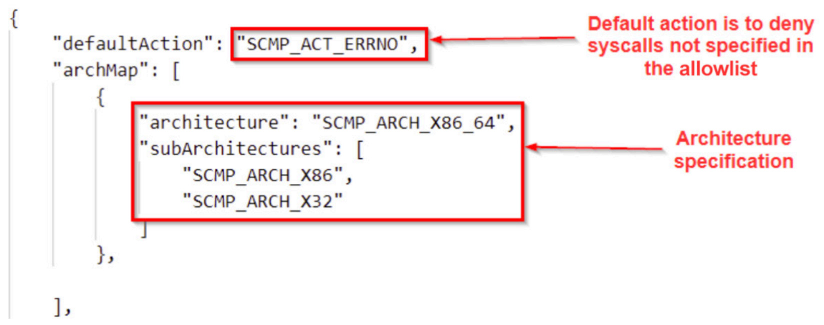
CREATING A CUSTOM SECCOMP PROFILE

Similar to AppArmor, creating your own custom seccomp profile for Docker will require an intimate knowledge of the system calls utilized by your container. You can use the default Docker seccomp profile as a starting point and modify it as needed.

1. Start by downloading the profile from GitHub: https://github.com/moby/moby/blob/master/profiles/seccomp/default.json

   You can store your custom seccomp profiles wherever you want. However, it is recommended to use a standardized directory for all your custom profiles. Custom seccomp profiles are saved in the `.json` format.

2. The following image highlights the architecture specification for the default seccomp profile:

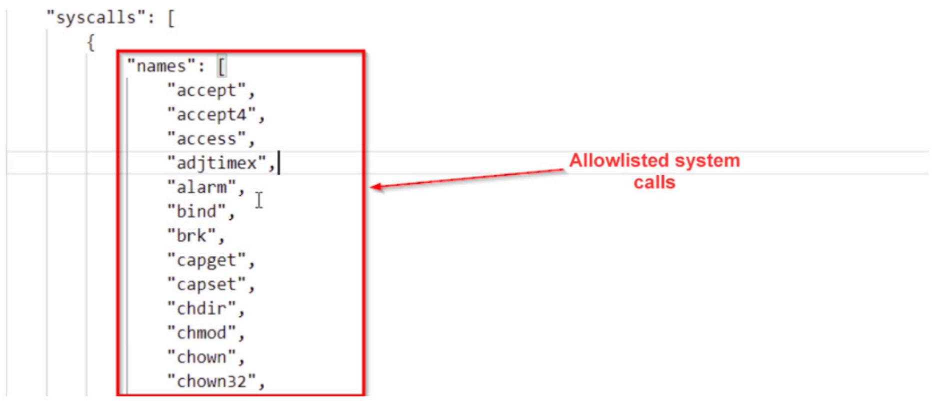


```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "archMap": [
        {
            "architecture": "SCMP_ARCH_X86_64",
            "subArchitectures": [
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
            ]
        },

    ],
```

Default action is to deny syscalls not specified in the allowlist
Architecture specification

In the seccomp profile template, the default action is to deny the container from accessing any system calls not specified in the syscall allowlist. The image below highlights the syscall allowlist, where you can specify what system calls you want your container to have access to. You can modify this profile based on your requirements.

Limiting Container System Calls with seccomp



```
"syscalls": [
    {
        "names": [
            "accept",
            "accept4",
            "access",
            "adjtimex",
            "alarm",
            "bind",
            "brk",
            "capget",
            "capset",
            "chdir",
            "chmod",
            "chown",
            "chown32",
```

3. When saving the profile, you can save it with a file name that pertains to the functionality that it restricts or permits.

4. We can specify the custom seccomp profile with the `--security-opt` option when running a container:

    ```
    docker run -d --security-opt
    seccomp:/path/to/profile/profile.json <docker image>
    ```

    This will run the container with your custom seccomp profile. Based on your profile, the container will be limited to the allowlisted system calls specified in the profile, which can be very useful in limiting the functionality available to users in the container as well as restricting the functionality of applications.

RUNNING CONTAINERS WITHOUT THE DEFAULT SECCOMP PROFILE

You also have the ability to run containers in an unconfined mode with no seccomp profile specified. This is **not recommended**, as the container will have access to all system calls available. This can be done by running the following command:

```
docker run -d --security-opt seccomp:unconfined <docker image>
```

Now that you have an understanding of how to use custom seccomp profiles for your containers, we can explore the process of performing vulnerability scans on your Docker images.

Vulnerability Scanning for Docker Containers

Docker vulnerability scanning is the process of identifying security vulnerabilities for packages utilized in a Docker image. This process will allow you to detect vulnerabilities in images before deploying or running them. These vulnerabilities can then be patched or fixed in order to make the image as secure as possible. This is a very important aspect of Docker security, primarily because all of the security measures we have implemented can be usurped by a vulnerability in an image's packages.

It is to be noted that this guide will only cover the process of identifying vulnerabilities in Docker images. Patching and remediation should only be handled by the respective developer or DevOps team in accordance with the guidelines specified by your organization.

SCANNING FOR VULNERABILITIES WITH TRIVY

In order to perform our vulnerability scans on our Docker image, we will utilize an open source third-party tool called Trivy. Trivy is a simple and comprehensive vulnerability scanner for containers and other artifacts. More information about Trivy can be found on GitHub: https://github.com/aquasecurity/trivy

1. Trivy has a pre-built Docker image that can be used to perform the vulnerability scans for us, as a result, we do not have to install it on our Docker host. We can pull the Trivy Docker image by running the following command:

   ```
   docker pull aquasec/trivy
   ```

2. After pulling the image you will need to create a cache directory for the Trivy image. This directory will be used to store all the cached data:

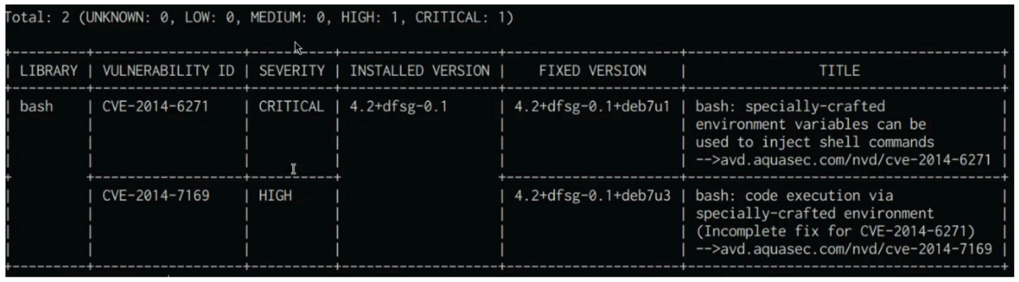
   ```
   mkdir -p trivy/.cache
   ```

3. After you have created the cache directory, we can perform a vulnerability scan on an image by running the Trivy image with the following parameters:

   ```
   docker run --rm -v trivy:/path/to/cache/.cache aquasec/trivy
   <image-name>
   ```

Vulnerability Scanning for Docker Containers

4. This will run a vulnerability scan on the specified image. The Trivy container should output the results of the vulnerability scan as highlighted in the image below:



```
Total: 2 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 1, CRITICAL: 1)

+---------+------------------+----------+-------------------+-----------------------+------------------------------------+
| LIBRARY | VULNERABILITY ID | SEVERITY | INSTALLED VERSION |     FIXED VERSION     |               TITLE                |
+---------+------------------+----------+-------------------+-----------------------+------------------------------------+
| bash    | CVE-2014-6271    | CRITICAL | 4.2+dfsg-0.1      | 4.2+dfsg-0.1+deb7u1   | bash: specially-crafted            |
|         |                  |          |                   |                       | environment variables can be       |
|         |                  |          |                   |                       | used to inject shell commands      |
|         |                  |          |                   |                       | -->avd.aquasec.com/nvd/cve-2014-6271 |
+         +------------------+----------+                   +-----------------------+------------------------------------+
|         | CVE-2014-7169    | HIGH     |                   | 4.2+dfsg-0.1+deb7u3   | bash: code execution via           |
|         |                  |          |                   |                       | specially-crafted environment      |
|         |                  |          |                   |                       | (Incomplete fix for CVE-2014-6271) |
|         |                  |          |                   |                       | -->avd.aquasec.com/nvd/cve-2014-7169 |
+---------+------------------+----------+-------------------+-----------------------+------------------------------------+
```

Trivy will sort the results based on the vulnerability ID, severity, and the installed and patched versions of the software packages affected. This information can then be passed along to the respective teams for patching

5. After the vulnerabilities have been patched, the scan must be re-run to verify that the patches have remediated the vulnerability. It is always recommended to perform regular vulnerability scans on your images before running containers in a production environment.

You should now have an understanding of how to scan Docker images for vulnerabilities. Next, we'll explore the process of building secure Docker images.

Building Secure Docker Images

Docker allows users and organizations to create custom images. However, images are often created without security best practices in mind. It is vitally important to analyze your Docker images and to identify potential misconfigurations in your Dockerfiles as these misconfigurations can be dangerous if left unaddressed. Always follow security best practices when generating Docker images.

SCANNING DOCKER IMAGES WITH DOCKLE

The process of identifying misconfigurations in Docker images can be automated through the use of a third party open source tool called Dockle. More information about Dockle can be found on GitHub: https://github.com/goodwithtech/dockle

1. The Dockle Debian package can be downloaded by running the following script on the Docker host:

```
VERSION=$(
curl --silent "https://api.github.com/repos/goodwithtech/dockle
releases/latest" | \
grep '"tag_name":' | \
sed -E 's/.*"v([^"]+)".*/\1/' \
) && curl -L -o dockle.deb https://github.com/goodwithtech/
dockle/releases/download/v${VERSION}/dockle_${VERSION}_Linux-
64bit.deb
```

2. After the Dockle debian package has been downloaded, we can install it by running the following command:

```
sudo dpkg -i dockle.deb && rm dockle.deb
```

3. After installing Dockle, we can begin scanning Docker images by running the following command:

```
dockle <image-name>
```

Building Secure Docker Images

4. Dockle will scan the image and identify misconfigurations in its Dockerfile as highlighted below:

```
alexis@localhost:~$ dockle polinux/stress
FATAL - DKL-DI-0004: Use “apk add” with --no-cache
      * Use-no-cache option if use ‘apk add’: /bin/sh -c apk add --update bash g++ make curl &&
curl -o /tmp/stress-${RELEASE_VERSION].tgz https://fossies.org/linux/privat/stress-${RELEASE_VER-
SION}.tar.gz 8& cd /tmp && tar
xvf stress-${RELEASE_VERSION}.tgz && rm /tmp/stress-${RELEASE_VERSION}.tgz && cd /tmp/stress-${RE-
LEASE_VERSION}
&& ./configure && make -j$(getconf _NPROCESSORS_ONLN) && make install && apk del g++ make curl &&
rm -rf /tmp
/* /var/tmp/* /var/cache/apk/* /var/cache/distfiles/*

WARN - CIS-DI-0001: Create a user for the container
     * Last user should not be root
WARN - DKL-DI-0006: Avoid latest tag
     * Avoid ‘latest’ tag
INFO - CIS-DI-0005: Enable Content trust for Docker
     * export DOCKER_CONTENT_TRUST=1 before docker pull/build
INFO - CIS-DI-0006: Add HEALTHCHECK instruction to the container image
     * not found HEALTHCHECK statement
```

Dockle will output the list of misconfigurations and recommendations for the changes that need to be made to the Dockerfile.

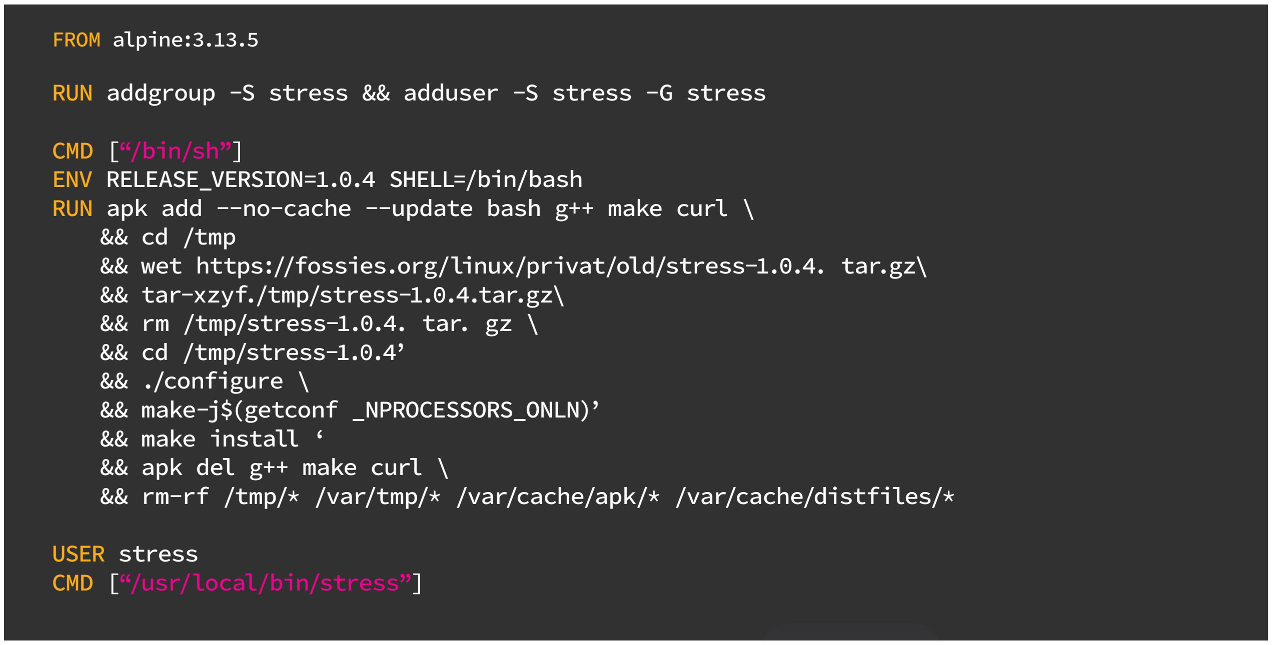
SECURITY BEST PRACTICES FOR BUILDING DOCKER IMAGES

The following is a list of best practices you should take in to consideration when building Docker images:

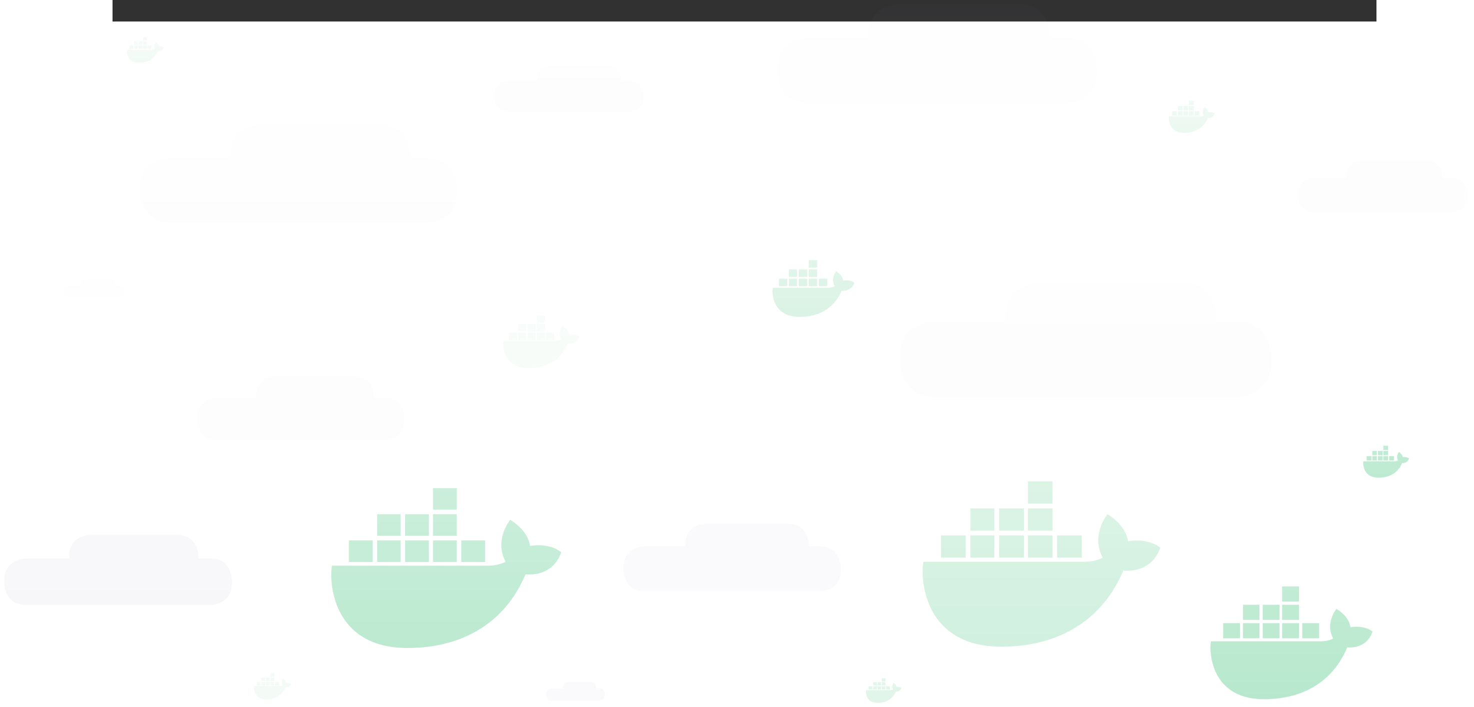
- Use minimal base images like Alpine Linux.
- Specify the exact version of the base image instead of using the “latest” tag.
- Reduce or remove unwanted packages from the image.
- Avoid storing secrets and passwords in the Dockerfile.
- Sign and verify Docker images.
- Add an unprivileged user to the Docker image.
- Avoid using the root user.

Building Secure Docker Images

The image below is an example of a Dockerfile that has been created and configured with security best practices in mind.



```
FROM alpine:3.13.5

RUN addgroup -S stress && adduser -S stress -G stress

CMD ["/bin/sh"]
ENV RELEASE_VERSION=1.0.4 SHELL=/bin/bash
RUN apk add --no-cache --update bash g++ make curl \
    && cd /tmp
    && wet https://fossies.org/linux/privat/old/stress-1.0.4. tar.gz\
    && tar-xzyf./tmp/stress-1.0.4.tar.gz\
    && rm /tmp/stress-1.0.4. tar. gz \
    && cd /tmp/stress-1.0.4'
    && ./configure \
    && make-j$(getconf _NPROCESSORS_ONLN)'
    && make install '
    && apk del g++ make curl \
    && rm-rf /tmp/* /var/tmp/* /var/cache/apk/* /var/cache/distfiles/*

USER stress
CMD ["/usr/local/bin/stress"]
```



Linode's Take

In cloud computing, deployments of Docker and container technologies now rival traditional Linux virtual machines. As their capabilities and uses continue to expand and the overall use of containers increases, it is essential to ensure that you follow these best practices to secure Docker. Having the skills to regularly audit both new projects and existing workloads to get ahead of potential security vulnerabilities will remain a necessity.

Making the cloud simple, affordable, accessible, and secure for developers, partners, and businesses is core to Linode. Ensuring best practices for security creates a shared understanding of responsibility between a cloud provider and a user. All Linode plans include generous transfer allowances, automated Advanced DDoS Protection, and the ability to configure Cloud Firewalls and VLAN via Linode Cloud Manager, our fully-featured API, or CLI at no extra cost.

Encouraging customers to use security best practices begins with our bundled services and the additional educational resources and documentation we make available to raise security awareness.

When vulnerability prevention is integrated into each layer of your infrastructure and development process, you secure application data and reduce potential technical debt while ultimately protecting both your users and yourself.

Next Steps

- Watch our on-demand Docker Security Essentials series to follow HackerSploit's practical demonstrations and more information on the best practices outlined in this ebook.
- Easily deploy Docker with Linode's Marketplace App.
- Keep reading about using Docker and Linode with our extensive Docker guides, including using Docker Images, Containers, and Docker Files in Depth and a Docker Commands Cheat Sheet. Browse all Linode docs on Docker and containers.

About Linode

Our mission is to accelerate innovation by making cloud computing simple, affordable, and accessible to all.

Founded in 2003, Linode helped pioneer the cloud computing industry and is today the largest independent open cloud provider in the world. Headquartered in Philadelphia's Old City, the company empowers more than a million developers, startups, and businesses across its global network of 11 data centers.

linode

The World's Largest
Independent Open Cloud