Hardening Docker: A Security Toolkit

Container technology has radically changed the way that applications are being developed and deployed. Notably, containers dramatically ease dependency management, so shipping new features or code is faster than ever before. While Docker containers and Kubernetes are great for DevOps, they also present new security challenges that both security practitioners and developers must understand and address with diligence. Docker's team of security experts has built some valuable security features into the Docker platform over the last several years. Use these valuable tips and practical suggestions on how to harden containers and hosts as a first step toward a more secure container environment.

A Brief Look at Containers from a Security Perspective

In essence, Docker containers are a wrapper around Linux control groups (`cgroups`) and namespaces. Cgroups are used in the Linux kernel for monitoring and restricting resources among a group of processes. Namespaces determine what a process can see. For example, the `PID` namespace restricts which processes can be seen within a container.

Each container running on a host shares a common underlying kernel. Containers are isolated from one another, which – from a security standpoint – is advantageous. However, if the host OS is compromised, all containers running on it are at risk. Similarly, if a container is using a vulnerable library, it could be exploited to gain access to the underlying host.

Securing the Host OS

The underlying host OS needs to be secured in order to prevent container breaches from affecting the host. To address this challenge, Linux provides several out-of-the-box security modules. Some of the popular ones are **SELinux**, **AppArmor** and **seccomp**. One can also develop custom security modules using **Linux Security Modules** (`LSMs`).

SELinux

SELinux is a type of Mandatory Access Control (`MAC`) security module based on type enforcement. Type enforcement revolves around defining a type and assigning privileges to those types. Here's a simple example of using an SELinux policy:

```
policy_module(localpolicy, 1.0)

gen_require('
  type user_t;
  type var_log_t;
)

allow user_t var_log_t:dir { getattr search open read };
```

This configuration allows any Linux process to execute file operations (such as `getattr`, `search`, `open`, `read`) for which `user_t` and `var_log_t` are applied.

Even though SELinux is comprehensive, you can see the policy language is pretty hard to get right. Often users will end up creating different SELinux policies over time to address different scenarios.

AppArmor

AppArmor is another MAC solution. It is based on file system paths rather than defining types. Users can specify a file path to a binary and the permissions they have. Here's a simple example of confining nginx:

```
#include <tunables/global>
/usr/sbin/nginx {
  #include <abstractions/apache2-common>
  #include <abstractions/base>
  #include <abstractions/nis>
  capability dac_override,
  capability dac_read_search,
  capability net_bind_service,
  capability setgid,
  capability setuid,
  /data/www/safe/* r,
  deny /data/www/unsafe/* r,
  /etc/group r,
```

```
  /etc/nginx/conf.d/ r,
  /etc/nginx/mime.types r,
  /etc/nginx/nginx.conf r,
  /etc/nsswitch.conf r,
  /etc/passwd r,
  /etc/ssl/openssl.cnf r,
  /run/nginx.pid rw,
  /usr/sbin/nginx mr,
  /var/log/nginx/access.log w,
  /var/log/nginx/error.log w,
}
```

This AppArmor profile blocks reading from unsafe directories. It may not be suitable for all situations; users may have to customize it. AppArmor is good for restricting application access. However, it requires a learning curve to be able to write good enforcement profiles.

Use Namespaces to Establish Security Boundaries

`Seccomp` (short for 'Secure Computing') is another security module included in many Linux distributions that allows users to restrict system calls. Users can specify custom actions to be taken when a certain system call is executed. The actions are `allow`, `kill`, `err`, and `trap`. Seccomp can be used to sandbox applications that handle untrusted user inputs to a subset of system calls.

The first step in using seccomp is to determine all the system calls an application makes when it runs. This process can be a difficult and error-prone exercise that should be conducted when the application is written. Users can use tools like `audit` to profile all the system calls that it makes by exercising it in different ways.

Seccomp policies are defined using JSON files. A sample seccomp policy looks like the following:

```
{
       "defaultAction": "SCMP_ACT_ALLOW",
     "syscalls": [
            {
              "name": "mkdir",
              "action": "SCMP_ACT_ERRNO"
            },
```

```
            {


              "name": "chown",
              "action": "SCMP_ACT_ERRNO"
            }
        ]
}
```

This policy causes an error to be returned when `mkdir` or `chown` are executed.

The drawback with seccomp is that the profile has to be applied during the launch of the application. The granularity of restricting system calls is too narrow and requires extensive working knowledge of Linux to come up with good profiles.

Capabilities

Linux capabilities are groups of permissions that can be given to child processes. Child processes cannot acquire newer capabilities. The idea behind capabilities is that no process should have all privileges, but instead, have only enough privileges to perform their intended service. By bootstrapping processes with limited privileges, we can "contain" the damage that can occur if they are ever compromised.

Some capabilities give excessive privileges to processes such as:

- **SYS_ADMIN**: This capability gives processes many privileges, some of which are given only to the root user.
- **SETUID**: Many of the Linux distributions ship binaries that run with the `setuid` bit set to give root privileges by default. Instead of `setuid` bit, they can be replaced with capabilities to provide more granular privileges.

These options work well on the host OS, but it can be challenging to adapt them for containers. Let's look at some of the relevant attack surfaces and practical ways to secure the Docker container runtime.

Container Runtime Security Practices

There are various factors to consider when adopting Docker containers for production. When it comes to running Docker container securely, users can follow these recommendations.

Unix socket (/var/run/docker.sock)

By default, the Docker client communicates with the Docker daemon using the unix socket. This socket can also be mounted by any other container unless proper permissions are in place. Once mounted, it is very easy to spin up any container, create new images, or shut down existing containers.

Solution: Set up appropriate SELinux/AppArmor profiles to limit containers mounting this socket.

Volume mounts

Docker allows mounting to sensitive host directories. Also, the contents of the host file system can be changed directly from the container. For application containers with direct Internet exposure, it is important to be extra careful when mounting sensitive host directories (`/etc/`, `/usr/`). Any breach can lead to damaging data loss.

Solution: Mount host-sensitive directories as read-only.

Privileged containers

Privileged containers can do almost anything a host can do. It runs with all capabilities.

Solution: Use capabilities to grant fine-grained privileges instead.

SSH within container

Running ssh service within containers makes managing ssh keys/ access policies difficult. This approach should be avoided if possible.

Solution:

- Do not run ssh services inside a container.
- Instead, run ssh on the host and use docker exec or docker attack to interact with the container.

Binding privileged ports

By default, Docker allows binding privileged ports **(<1024)** to a container. Normal users cannot access these ports. In many cases, mapping **http port 80** and **https port 443** is necessary for running servers in a container.

Solution:

- List all containers and their port mappings using the code below to ensure that the container's ports are not mapped to host ports below port 1024.
- `docker ps --quiet | xargs docker inspect --format '{{ .Id }}: Ports={{ .NetworkSettings.Ports }}'`

Exposing ports

Ports not necessary for the service must not be exposed.

Solution: List all the containers and their exposed ports using the following:

- `docker ps --quiet | xargs docker inspect --format '{{ .Id }}: Ports={{ .NetworkSettings.Ports }}'`
- Ensure that there are no unnecessary ports exposed.

Running without default AppArmor/ SELinux or seccomp

Docker runs containers with default AppArmor/SELinux and seccomp profiles. They can be disabled with the `--unconfined` option.

Solution: Do not disable the default profiles that Docker supplies.

Sharing host namespaces

Sharing namespaces have dangerous consequences if not managed properly. Containers can be started with `-pid` to connect with the host PID namespace or `--net` to share its network namespace. These allow containers to see and kill PIDs running on the host or even connect to privileged ports.

Solution: Avoid sharing host namespaces with containers.

Enabling TLS

If the Docker daemon is running on a TCP endpoint, it is advised to run with TLS enabled.

Solution: Docker offers a *helpful guide on enabling TLS with Docker*.

Do not set mount propagation mode to shared

Mount propagation mode allows mounting volumes in shared, slave or private mode on a container. Do not use shared mount propagation mode until needed.

A shared mount is replicated at all mounts and the changes made at any mount point are propagated to all mounts. Mounting a volume in shared mode does not restrict any other container to mount and make changes to that volume. This configuration might be catastrophic if the mounted volume is sensitive to changes. Do not set mount propagation mode to shared until needed.

Solution: Run the following command to list the propagation mode for mounted volumes:

- `docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}: Propagation={{range $mnt := .Mounts}} {{json $mnt.Propagation}} {{end}}'`
- Ensure that the mode is not set to `shared` unless needed.
- Do not start a container with the following invocation:
- `docker run --volume=/hostPath:/containerPath:shared <image> <command>`

Restrict a container from acquiring new privileges

A process can set the `no_new_priv` bit in the kernel. It persists across `fork`, `clone` and `execve`. The `no_new_priv` bit ensures that the process or its children processes do not gain any additional privileges via setuid or sgid bits.

Solution:

List the security options for all the containers using the following command:

- `docker ps --quiet --all | xargs docker inspect --format '{{ .Id }}: SecurityOpt={{.HostConfig.SecurityOpt }}'`
- The security options should list `no_new_privileges` as one of them.
- One can start a container with `no_new_privileges` as below:
- `docker run <run-options> --security-opt=no-new-privileges <image>`

Conclusion

Naturally, new technologies present new security challenges to organizations that choose to deploy it. As has been the case with the disruptive infrastructure technologies that preceded containers, the first step toward establishing a stronger security posture is to reduce the overall attack surface by hardening it. While the practices presented above are effective in making containers and hosts far less susceptible to exploits, the major container security challenges lie within the runtime phase. This phase of the container life cycle demands a dedicated container security platform.

The *StackRox Container Security Platform* is the industry's first container security platform that adapts detection, prevention, and response to changing threats.

Learn more about how StackRox protects organizations throughout the journey from containers to web-scale microservices.

References

https://www.cisecurity.org/benchmark/docker/