Computer Systems and Networks
23W

Report:
Security in Container Environments

David Unterholzner, Jakob Hofer, Jakob Khom, Leo Lach

2024-01-05

Contents

1 Introduction - What Are Containers? 1

2 Containers Are Just Processes 1

3 Container Isolation Layers 1
3.1 Namespaces . . . 1
3.2 Capabilities . . . 2
3.3 Control Groups . . . 2
3.4 AppArmor . . . 3
3.5 SELinux . . . 3
3.6 Seccomp . . . 3

4 Past Exploits 3
4.1 Container Malware: . . . 4
4.2 Privilege Escalation Exploit: . . . 4
4.3 Access Control Exploit: . . . 4
4.4 Network Security Exploit: . . . 4
4.5 Container Orchestrator Exploit: . . . 4

5 References

1 Introduction - What Are Containers?

Containers are a form of operating system virtualization and isolation. A single container can be used to run anything from a small microservice or software process to a larger application. Inside a container are all the necessary executables, binary code, libraries, and configuration files. There are different isolation techniques:

- **Linux-Containers** use kernel-level features like cgroups or namespaces to isolate applications. All Linux containers share the same kernel.
- **Sandboxed Containers** still effectively share the same kernel but use a dedicated sandbox program to run the contained processes. Language-level sandboxes are, for example, the Java VM or the Python interpreter.
- **VM-Based Containers** use a hypervisor to provide isolation. The hypervisor virtualizes entire guest operating systems, each with its own kernel.

2 Containers Are Just Processes

The first thing that we need to understand about Linux containers is, that they are essentially just normal Linux processes launched in dedicated control groups and namespaces.
If we start a new container, for example, a web server with the `nginx` image from Docker, we can see that we just created a new process. We can interact with this process using standard Linux tools.

3 Container Isolation Layers

Now that we understand that containers are really just Linux processes, we want to explore how containers are isolated from the rest of the machine. How can we make sure that a process running in one container can't interfere with the operation of another container or the underlying host?

3.1 Namespaces

We will start with the initial layer of isolation: Linux namespaces.
Linux namespaces are a Kernel feature that enables the operating system to provide a particular process with an isolated view of one or more system resources. Namespaces serve as fundamental components to achieve isolation within containers, but they are also used outside of containerization. In total, Linux supports the following eight namespaces:

- Mount (`mnt`)
- PID (`pid`)
- Network (`net`)
- IPC (`ipc`)
- UTS (`uts`)
- Cgroups (`cgroup`)
- User (`user`)
- Time (`time`)

Every process is restricted to being present in each namespace just once. But multiple processes can be part of one specific namespace (For example, a number of n processes can be part of one single `mnt` namespace). This means that all processes, part of the same namespace, share the same view over the given resource. When creating a standard process in Linux, it is placed in what is commonly referred to as the *global namespace*. This is a set of default namespaces (e.g. `mnt, pid, net, ...`) which provide processes with a shared view of resources. This implies a lack of isolation, but in turn, allows processes to interact freely with each other's resources.
When starting a new container, Docker automatically creates a *new* set of namespaces for the new process. By default, docker uses the namespaces:

`mnt`, `pid`, `net`, `ipc`, `uts` and `cgroup`.

Let's briefly talk about the most important namespaces in the context of containerization:

The **Mount (`mnt`)** namespace is used to provide the process with an isolated view of the filesystem hierarchy. When a new `mnt` namespace is created, for example when starting a container, a new set of filesystem mounts is provided. When a process performs filesystem operations (e.g. reading a file, creating a directory), the kernel resolves the path based on the process's current namespace. This means that changes to the filesystem made within one namespace do not affect the filesystem seen by processes in other `mnt` namespaces.

The **PID (`pid`)** namespace provides isolation within the set of process IDs. In a specific `pid` namespace, each process has a unique set of process IDs, beginning from 1. For instance, a process with PID 1 in our container may correspond to PID 2741 in the global namespace. Despite the differing IDs, they both represent the same underlying process, resolved within the context of their respective namespaces.

The **Network (`net`)** namespace is responsible for the isolation of network-related resources. Each network namespace has its own set of network interfaces, routing tables, firewall rules, and network-related configuration. This enables containers to provide a completely separate network environment for its contained processes.

Even though the **User (`user`)** namespace is not enabled in Docker by default, it is still very useful in containerization, especially for security reasons. The `user` namespace allows for processes to be root inside the namespace, without actually being root on the host. If a process within a container escapes and gains root access, it may only have privileges within its own user namespace, limiting the potential damage it can cause to the host.

Control groups (`cgroup`) allow us to control a process's resource usage on the host machine. We will cover them in more detail in a later section (3.3).

The remaining namespaces (`ipc` and `uts`) are not relevant to many use cases, as they serve a very specific purpose, like providing isolation for resources like POSIX message queues (`ipc`), or setting the hostname used by a process (`uts`). We will not cover them in more detail here.

3.2 Capabilities

As our next layer of isolation, we will take a look at Linux capabilities. Traditionally in Linux, if a process needed to do anything only the root user could do, it needed to be executed as root, granting it *all* the privileges, even if it only needed a specific one. This "all-or-nothing" approach posed a lot of security concerns due to the lack of granularity and increased vulnerability risks associated with root programs. As a result, Linux introduced capabilities, which split the root privilege into over 40 privileges that can be individually granted to processes or files.
When creating a container, Docker assigns the process a given set of capabilities. This set was designed so that most workloads inside containers would be able to run successfully, while also restricting any unnecessary capabilities that could lead to privilege escalation attacks or pose other security risks.
Presented below are a couple of examples:

- `CAP_CHOWN`: Capability that allows a process to change file ownership.
- `CAP_NET_RAW`: Allows the creation of ICMP packages.
- `CAP_DAC_OVERRIDE`: Bypass file read, write, and execute permission checks.
- `CAP_SETUID`: Allows us to set the UID of a process.

To help increase the security of containers and make them less vulnerable to attacks, it is important to minimize a container's set of capabilities. Depending on the application, you may be able to drop some or all of the default capabilities assigned by Docker.

3.3 Control Groups

Control groups (cgroups) are a Linux kernel feature that manages, monitors, and limits system resources such as memory, CPU cores, and network bandwidth.
You handle them through a virtual file system at "/sys/fs/cgroup".

cgroups are organized hierarchically and appear as directories in the file system, and their settings are managed through special files.
To keep tabs on resources, the respective files can be read or written.

Example: To restrict the memory usage of untrusted programs to 500mb of physical memory, we create a cgroup and set the memory limit to 500mb.
Whenever we run an untrusted program, we start its process under the said cgroup, and it only sees the specified amount of system memory. Attempting to allocate more than that would cause the system's OOM-killer to kill the program.

3.4 AppArmor

AppArmor is a set of policies that can restrict the access to files, capabilities, network access and more for individual programs. In comparison, isolation through namespaces and cgroups does not offer such a fain-grained control as they are applied to the whole system. The purpose is to limit the damage that a compromised program can do to the system. Docker loads an AppArmor profile by default, which provides a good starting point for creating a secure container.

3.5 SELinux

SeLinux is another similar Linux security module but has a different architecture and more sophisticated ways of creating policies. It can operate in either the *enforcing* or *permissive* mode, where in both cases violations are logged but only in the enforcing mode the policies are enforced. SeLinux can also be integrated with Docker and is used to ensure the separation of containers by running them in their own security context. This means that a compromised container cannot affect another container or the host system as easily.

3.6 Seccomp

This is one of the most crucial security layers for containers, as containers share the kernel with the host system, unlike virtual machines. Programs in the host or container interact with the kernel via syscalls. Many of them are safe like "write" for printing text to the terminal, but others can be dangerous. Many syscalls are blocked by default in Docker, but when the user inside the container has root permissions and seccomp is disabled (unconfined seccomp), it may be possible that the container can take over the host system. For example, the command "insmod" invokes the "finit_module" syscall, which loads a kernel module. A kernel module is a C program that runs with the highest privileges in ring 0. Once loaded, it could create a reverse shell and give an attacker access to the host system remotely.

4 Past Exploits

CVE	Description	Affected Systems
CVE-2017-1002101	subPath Volume Mount Vulnerability	Docker
CVE-2017-16995	eBPF Vulnerability	Linux
CVE-2018-1002105	Severe Privilege Escalation Vulnerability	Kubernetes
CVE-2019-1002100	API Server Patch Permission DoS Vulnerability	Kubernetes
CVE-2019-5736	High Severity RunC Vulnerability	Docker
CVE-2020-35467	Blank Password could allow root access	Docker
CVE-2018-15514	User in the "docker-users" group could gain Admin privileges	Docker/Windows
CVE-2014-6407	Remote attacker write to arbitrary files/execute arbitrary code	Docker
CVE-2016-5195	Dirty COW	Linux

Security attacks on container environments often aim to breach the contained space and infiltrate the underlying system. In this Segment, we try to showcase different past security vulnerabilities. Security attacks on Container primarily focus on 5 exploits sorted by difficulty, they are:

- `Container Image Malware`: Container Images with Malware being deployed.
- `Privilege Escalation Exploits`: Processes gaining unauthorized access to resources beyond their permission scope.
- `Access Control Exploits`: Containers granted more privileges than necessary, potentially leading to unauthorized access.
- `Network Security Exploits`: Exploits arising from improperly configured networking rules allowing unauthorized communication.
- `Container Orchestrator Exploits`: Vulnerabilities within the orchestration tool allow various forms of attacks.

4.1 Container Malware:

In the years 2019 and 2020 Container images hosted on the Docker Hub repository, were revealed to contain Malware. Malware such as Cryptocurrency miners made up 44% of the malicious images. Malware Exploits are the most common exploits, however, they can be easily prevented by an image scan of the Container before deployment.

4.2 Privilege Escalation Exploit:

In the year 2014, a certain race condition was present during docker engine version 1.3.2 and created a vulnerability to extract files of arbitrary paths on the host machine. The exploit used symlink and hard link traversals during the docker pull and docker load operations in the Docker's image extraction. Enabling remote code execution and privilege escalation.

4.3 Access Control Exploit:

The Kubernetes dashboard had an exploit that allowed users to skip an authentication check. Through this, malicious users could gain access to a server's TLS certificate and private key and exploit the privileges the Kubernetes pod held.

4.4 Network Security Exploit:

The Docker Daemon API allows users to communicate with the docker daemon using an HTTP client. If his Configuration is faulty or sloppily done, it exposes a vulnerability that allows an attacker to start a docker container and attach the host's /etc to the container and read/write files in etc.

4.5 Container Orchestrator Exploit:

Container Orchestrators such as Docker Swarm are crucial in large container environments. They automate deployment, management, scaling, networking and availability. The Role Based Access Control or short RBAC is a tool that manages user access to resources. In the case the RBAC is misconfigured an attacker may have unauthorized privileges, which, depending on these privileges, may allow the attacker to deploy Container or Workloads on other parts of the Network.

5 References

References

[1] Author, A. (2020). Title of the paper. *Journal Name*, *Volume*(Issue), PageRange.

[2] Rory McCune. (2023). *Container security fundamentals: Exploring containers as processes.* Retrieved from https://securitylabs.datadoghq.com/articles/container-security-fundamentals-part-1/

[3] Rory McCune. (2023). *Container security fundamentals: Isolation & namespaces.* Retrieved from https://securitylabs.datadoghq.com/articles/container-security-fundamentals-part-2/

[4] Michael Kerrisk. (2013). *Namespaces in operation, part 1.* Retrieved from https://lwn.net/Articles/531114/

[5] Linux manual page. *namespaces - overview of Linux namespaces.* Retrieved from https://man7.org/linux/man-pages/man7/namespaces.7.html

[6] Rory McCune. (2023). *Container security fundamentals: Capabilities.* Retrieved from https://securitylabs.datadoghq.com/articles/container-security-fundamentals-part-3/

[7] Linux manual page. *capabilities - overview of Linux capabilities.* Retrieved from https://man7.org/linux/man-pages/man7/capabilities.7.html

[8] Carlos Polop. (2023). *Linux Capabilities.* Retrieved from https://book.hacktricks.xyz/linux-hardening/privilege-escalation/linux-capabilities

[9] Ben Goldfarb. (2019). *Vulnerabilities in the Container Ecosystem: A Brief History.* Retrieved from https://blog.aquasec.com/container-security-vulnerabilities

[10] Devdatta Mulgund. (2019). *A Brief History of Containerization: Why Container Security Best Practices Need to Evolve Now.* Retrieved from https://securityintelligence.com/posts/a-brief-history-of-containerization-why-container-security-best-practices-need-to-evolve-now/

[11] Sagie Dulce. (2016). *Dirty COW Vulnerability: Impact on Containers.* Retrieved from https://blog.aquasec.com/dirty-cow-vulnerability-impact-on-containers

[12] Tim Smith. (2023). *Container Image Security: Protecting Against CVEs.* Retrieved from https://blog.mondoo.com/container-image-security-protecting-against-cves

[13] Murray McAllister. (2014). *Bug 1167505 (CVE-2014-6407) - CVE-2014-6407 docker: symbolic and hardlink issues leading to privilege escalation.* Retrieved from https://bugzilla.redhat.com/show_bug.cgi?id=1167505

[14] Chris Tozzi. (2023). *5 Common Container Exploits.* Retrieved from https://www.slim.ai/blog/5-common-container-exploits

[15] Phil Muncaster. (2020). *Docker Users Targeted with Crypto Malware Via Exposed APIs.* Retrieved from https://www.infosecurity-magazine.com/news/docker-crypto-malware/

[16] LORENZO DAVID. (2019). *CVE-2018-18264 Privilege escalation through Kubernetes dashboard..* Retrieved from https://sysdig.com/blog/privilege-escalation-kubernetes-dashboard/