AWS Firecracker MicroVMs to Host Rootless Podman Containers

1. Introduction

1.1 Overview

This report outlines the implementation and benefits of nesting each Secure Internet Surfing (SIS) container within a lightweight virtual machine (VM) using Firecracker microVMs (uVMs). The SIS system currently relies on rootless Podman containers for user session isolation, and this project aims to improve security by leveraging the hardware-level isolation provided by Firecracker uVMs. By doing so, we fortify the segmentation boundary between user sessions and between each session and the host, ultimately strengthening the security of the SIS system.

1.2 Motivation

The existing SIS system utilises rootless Podman containers, primarily relying on namespaces to establish a layer of separation between the user and the host. The objective of this project is to realise enhanced security of the SIS system by further isolating user sessions from the host and from one another while preserving the ability for resources to be shared across many ephemeral workloads and keeping overhead low. Firecracker allows us to do so by allocating each uVM its own custom set of resources such as kernel, root file system (rootfs) etc, and only emulating the essential devices. This is ideal for our use case – Internet surfing which does not require extravagant functionalities.

1.3 Background

Containerization only offers operating system (OS) level isolation. Despite each container running inside its own mount, user, network, ipc and pid namespaces, they are all controlled by a common container engine, which directly interacts with the host OS.

On the other hand, traditional virtualization provides hardware-level isolation, with each VM owning its individual set of virtualized resources and running its own OS. The VM runs its own kernel and is independent of the host's, making it harder for the user to break into the host OS. However, creating its own set of resources also means traditional VMs are typically bulkier and have longer startup time.

1.4 Contributions

This project demonstrates the hosting of rootless Podman containers in custom-built Firecracker uVMs. It also shows the snapshotting feature that allows uVMs to be restored quickly.

1.5 Related Work

1.5.1 Kata Containers

Kata Containers uses a hypervisor to launch a lightweight VM that runs systemd as its kernel init system that in turn launches an agent in the VM to create a Docker container environment. It offers flexibility in terms of the hypervisor used, with support for QEMU, Cloud-hypervisor etc which makes it more suited for VMs that require added guest functionalities such as ACPI Power Management since QEMU supports a larger range of hardware emulation as compared to Firecracker. Yet, due to the higher complexity, startup latency and resource consumption increases, which limits the number of uVMs deployed at any point in time.

1.5.2 Nabla Containers

Nabla containers adopt a strategy of attack surface reduction to the host by minimising the containers' access to system calls. It uses library OS techniques to substitute all but 7 system calls, isolating the container from the host container. However, because Nabla containers utilises unikernels, its image format is not OCI image–spec compliant and therefore requires rebuilding of existing SIS containers.

2. Design

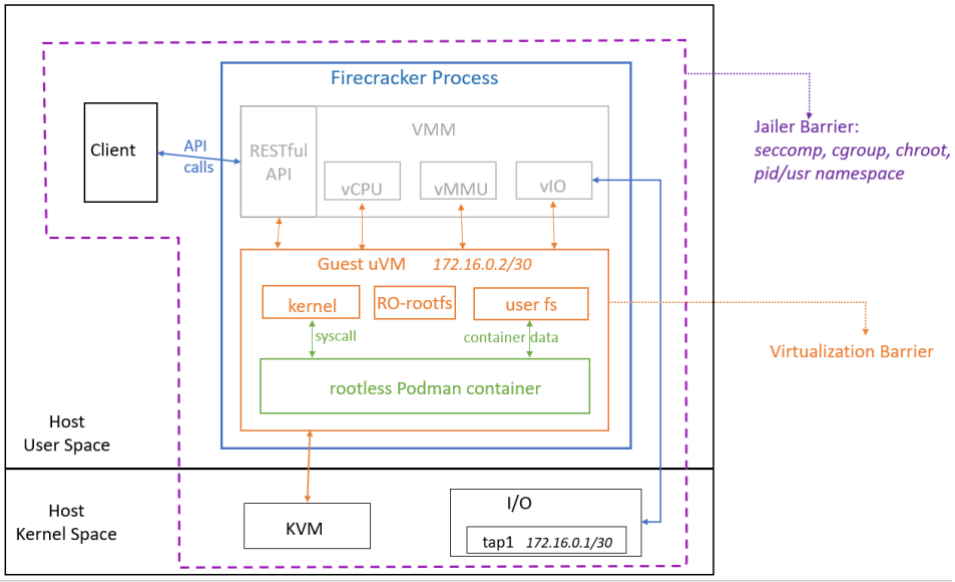

Figure 1(a): Proposed setup -- Firecracker-Podman Architecture

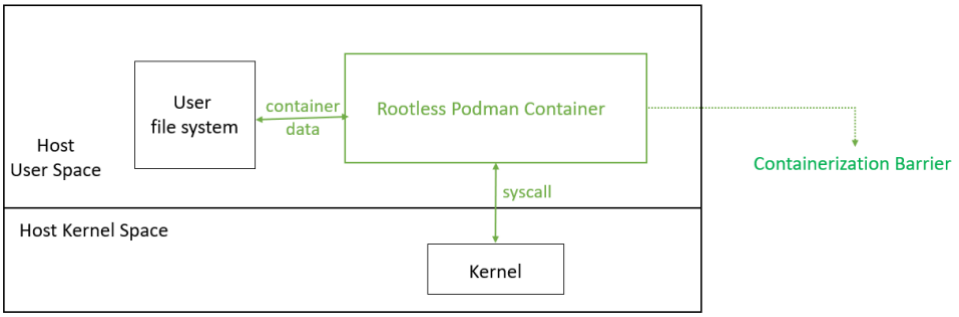

Figure 1(b): Original setup -- Podman Architecture

2.1 Virtualization Barrier

In our implementation, each uVM runs its own kernel, filesystems and interacts with its own set of emulated hardware devices as shown in figure 1(a). All code execution by the user is contained in the uVM, including syscalls that are executed in the guest kernel. Hence, even if the guest kernel is exploited, only the uVM is affected. The guest is not able to influence anything outside of its uVM, namely the host and other neighbouring uVMs, unless it is able to gain control of the Firecracker hypervisor. This contrasts with the original architecture where the user workload is directly interacting with the host's kernel and hardware resources such as the CPU. In that case, syscalls issued by the user workload could affect the host OS and hence other guest containers residing on the host.

We also configure the guest kernel to include only essential modules, reducing boot time, memory footprint, and attack surface area. This is not possible in the original setup defined by underlying containerization architecture where containers run on their host's kernel.

Moreover, we configure Podman to store its data, such as container images, in its own secondary disk that is mounted as the user's home file system and is inaccessible to other guest uVMs. This is unlike the original setup where all container data including the rootfs used by the containers are stored in a common directory on the shared host. Attackers could make use of such a vulnerability to corrupt the host's file system. And if a user were to break out of its own container, it could also access other containers' data.

Besides the enhanced security, Firecracker also offers a snapshotting feature, valuable for backup and recovery purposes and ensuring the system's resilience against unexpected incidents.

2.2 Jailer Barrier

In production, we isolate all Firecracker-related processes in an execution jail as illustrated in purple in Figure 1(a).

2.2.1 Chroot Jail

Before launching a Firecracker uVM, a chroot jail is created to lock down the Firecracker process and the user ID to be executing the Firecracker process. Every Firecracker uVM owns a personal copy of the Firecracker binary, a tuntap device, Linux Kernel-based Virtual Machine (KVM), and urandom device. The custom-built kernel image and file system images to be mounted onto the uVM must also reside within the jail so that it could be accessed by the user id executing the Firecracker process. All these resources are accessed only by corresponding Firecracker process.

2.2.2 Seccomp

To restrict the uVM's access to system calls, Seccomp filters are implemented.

2.2.3 Cgroups

Despite the many differences, cgroups is used in both the setups to manage the CPU and memory available to guest workloads. In the original setup, we limit the resources available to the Podman process, whereas in this new setup, we limit what is available to the Firecracker process.

3. Implementation

3.1 Configuring Firecracker

While Firecracker VMM is built to be processor agnostic and hardware virtualization support for 64-bit Intel CPUs are generally available, we encountered an issue with detecting the frequency of our Intel Xeon Gold 5318Y processor. To resolve this, we could either provide the frequency or edit the Firecracker source code to ignore the frequency. Due to time constraints, we employed the more straightforward method of hardcoding the host CPU's frequency into the Firecracker source code.

3.2 Kernel

To host rootless Podman containers in the uVMs, the base kernel configuration recommended by Firecracker is insufficient and we require several additional modules which mainly include:

- Netfilter modules to support nftables in the guest VM for Internet connectivity.
- Fuse-overlayfs modules to support file system in user space for rootless containers. *Note: this could be redundant once kernels >v5.14 are deployed, but proper snapshotting support is currently unavailable for such kernel versions*
- Hardware random number generator modules to support greater entropy to enhance snapshot security which is elaborated in the later section.

We can then use the kernel building tool provided by Firecracker to build the guest kernel based on the updated config file.

3.3 Rootfs

Each uVM is mounted with a read-only rootfs and a read-write secondary disk accessible to the user. As opposed to having a single read-write rootfs for each uVM, this design eliminates the need for multiple copies of the rootfs image. Each uVM only requires a hard link pointing to a common rootfs EXT4 image. Albeit having a copy of the secondary disk for each uVM, there is no RAM wastage since it is mounted as a temporary file system (tmpfs) and thus, consumes only the required amount of RAM at any point in time instead of reserving the full amount of RAM.

To build a custom rootfs image, we first prepare a file and create an empty EXT4 file system on it. To populate it with the init system, /usr filesystem etc. and tools including Slirp4netns, Podman we do the following:

1. Bind-mount it onto an Alpine Docker container
2. Install OpenRC init system (for Alpine) and other tools like Podman, Slirp4netns etc.

3. Add a normal user without CAP_SYS_ADMIN and modify file permissions of the FUSE and KVM devices to assign read-write permissions to the user.
4. Add setuid and setgid capabilities to /usr/bin/newuidmap and /usr/bin/newgidmap respectively. This allows Podman to write user and group ID mappings in a non-root user namespace
5. Edit container engine and storage configuration files to redirect the storage such as container image layer files to the user's home directory ie. the mount point of the RW disk.
6. Copy the newly configured system onto the mounted rootfs image.

To enable networking between the uVM and the host, we create a TAP device on the host and assign the uVM an IP address in the same subnet. Finally, we can launch the Firecracker process with the jailer feature and pass in the kernel image and rootfs image either via a config file or the API socket.

4. Features

4.1 Snapshotting

Hosting each user session in an uVM enables snapshotting, which is done by sending an API request to the socket. We can then quickly resume the original guest workload in a different Firecracker process**.** The snapshots save the full state of the guest memory and emulated hardware state, stored across multiple files as below:

- Guest memory file,
- uVM state file,
- Disk files

This is a significant upgrade from containers where there is a lack of support for checkpointing rootless Podman containers which SIS is currently deploying.

It is important to ensure that these snapshots are loaded properly and do not lead to potential security risks. The greater guest OS entropy pool mentioned above helps to generate higher quality random numbers and tokens to ensure uVM uniqueness. However, there still lacks a strong mechanism that guarantees that the unique identifiers of each uVM remains unique across snapshot restores. Hence, a snapshot should only be loaded once so that resumption of execution from a state happens only once.

4.2 Logger

A Firecracker logger is configured for each uVM on startup to record the uVM's base activity. The log entry specifies the line of Firecracker source code that was executed and the outcome. For instance, when a PUT request is sent to the RESTful API server to create a full snapshot of the uVM, the line of source code triggered to parse the request, the response, and the API call duration are all logged. In the event where we would like to examine malicious activity detected in a uVM, we could complement its logs with the snapshot and reverse-engineer them. Ideally, the snapshot should give us an idea of the events that happened in the user space while the logs detail the events that happened in kernel space.

5. Future work

5.1 Network Filtering

A firewall could be established to block packets with a certain destination address that is deemed to be unsafe from being forwarded from the uVM to the host.

All network traffic in the uVM is routed through eth0 on the guest to its corresponding TAP device on the host, before getting forwarded to enp1s0 on the host. We employ Nftables to set up a firewall between the uVM and the host.

6. Conclusion

The deployment of Firecracker microVMs to host rootless Podman containers in the Secure Internet Surfing (SIS) system offers enhanced security and resilience. With a strong focus on hardware-level isolation and efficient snapshotting, Firecracker uVMs provide a robust solution for secure Internet browsing, complementing the existing containerized architecture. As we continue to refine and optimise our setup, we anticipate further improvements in security, performance, and overall system reliability.