Docker Integration With BPFContain

COMP4905 Honours Project

Trent Holmes

December 17, 2021

Supervised by Dr. Anil Somayaji
Carleton University

Abstract

Container technologies have seen widespread use throughout the world. Technologies such as Docker are used by many companies to deploy their services at scale. An important requirement of containers is their security. Docker provides the ability to use systems such as AppArmor, SELinux, and GRSEC to add an extra layer of security [6]. BPFContain demonstrates a security system based on the eBPF technology which can be used to confine processes. This project will discuss the integration of Docker with BPFContain to further confine Docker containers. The project will showcase the low level workings of Docker, and how eBPF probes can be used to hook into Docker container creation. Additionally, it will demonstrate the integration in action and discuss the challenges which occurred throughout integration. Furthermore, it will present performance benchmarks showcasing the amount of overhead caused by BPFContain. Finally, the project will conclude with a discussion about the value BPFContain provides for securing Docker containers and outline potential areas for future work.

Acknowledgements

I'd like to first start by thanking my project supervisor, Anil Somayaji, for their support and guidance. Secondly, I would like to thank BPContain creator, William Findlay, for their willingness to answer questions relating to BPFContain to ensure I was on the right track. Additionally, I would like to thank Magnus Kulke for their examples in the libbpf repository demonstrating how to create basic eBPF programs. Furthermore, Brendan Gregg's and Zain Asgar's writings were a significant help in learning how to attach user probes to Golang programs. Finally, I would like to thank my family and friends for their continued support throughout my undergraduate degree.

Table of Contents

List of Figures

List of Tables

1. Background

1.1. Introduction to eBPF

eBPF is a Linux technology that can be used to safely and efficiently extend the Linux kernel without requiring changes to the Linux source code [1]. Traditionally, the kernel must be extended using kernel modules, which present a high level of risk, as any flaws in the module may expose the entire system to vulnerabilities. To improve this process, eBPF programs are run through a verifier which ensures the eBPF Program is safe to run [1]. Additionally, eBPF runs efficiently by using Just-in-Time (JIT) compilation [1]. Generic bytecode is converted into machine specific instructions to allow the program to run as efficiently as native kernel code [1].

eBPF programs are event-driven meaning that eBPF program code is only run when some specific hook is hit [1]. For the purposes of this project, we are interested in user probes (uprobes), tracepoints and LSM hooks (Table 1.1). Additionally, to persist information across eBPF program invocations, built-in maps are available [1]. Specifically, this project will utilize eBPF hash tables.

Table 1.1: A summary of eBPF hook types referenced throughout the project [1,2]

Hook Type	**Description**	**Relation to Project**
User probes (uprobe)	Hook into user space source code	Used to hook into Docker source code
Tracepoints	Hook into system call entry/exit points	Used to hook into the entry point of sethostname system call
LSM Hooks	Hooks into the LSM framework to perform permissions checks on various system operations	Used by BPFContain to enforce policy decisions

Many eBPF development toolchains exist which make it easier for developers to create eBPF programs. BPFContain utilizes libbpf-rs which is essentially just a Rust wrapper of libbpf . Libbpf is a well supported tool chain which abstracts away the underlying BPF system call with an easier to use API library [1].

1.2. Introduction to BPFContain

Background

BPFContain is a confinement application created by William Findlay which leverages eBPF to enforce rules outlined in a YAML-based policy [3]. The policy consists of an entry point command and the operations it is allowed to perform [3]. In BPFContain, isolation was the main focus rather than virtualization [3]. Containers created are essentially just new processes within the same namespaces and file system as the host machine. Isolation is provided using eBPF LSM hooks to restrict the runtime access of a process.

Many container technologies exist which offer powerful virtualization features. One of the most popular container runtimes is Docker. Docker is an open-source project which has become part of the Cloud Native Computing Foundation [4]. Docker has seen widespread use allowing users to easily develop, ship and deploy applications [5]. Docker allows applications to be deployed in a consistent way across any environment. For companies using Docker for production environments security is a vital requirement. The attack surface of a container is restricted by using Linux namespaces, and cgroups to restrict what a container can do [6]. However, vulnerabilities may still arise which can allow attackers to compromise the host system. Docker allows users to add an extra layer of security by enabling well known security systems such as AppArmor, SELinux, and GRSEC [6].

The goal of this project is to integrate Docker with BPFContain functionality to provide further confinement to Docker containers. The integration will combine the valuable virtualization aspects of Docker with the powerful confinement properties of BPFContain to add another layer of security to Docker containers.

Work required/Objectives

Currently BPFContain consists of a shim which allows users to create a new container. The creation request is caught by an uprobe which spawns a new process confined by a specified policy file. BPFContain uses several eBPF maps to track all confined processes, containers and policy rules. Every containerized process corresponds to a container struct and a process struct. The container struct specifies which policy should be enforced, and tracks the containers namespace ids. The process struct tracks the pid and gid of all containerized processes. When a new container is created through the BPFContain shim, an entry is made

into both the processes map and the containers map. The namespaces assigned to the container struct are the same used by the host machine. Using the information contained in the process and container structures BPFContain enforces policy decisions through LSM hooks.

In order to enforce the same policy rules on a Docker container, an entry is required in both the processes map and containers map. However, in our case the namespaces will have been created by Docker. Thus, we will need to hook into the Docker container creation process when the namespaces are created in order to add entries into BPFContain with the Docker container information. This process will involve digging into the low level internal workings of Docker containers and expermeting with different eBPF probes to hook into the Docker. In addition, BPFContain must be updated to allow users to create Docker containers confined by a policy file specified by the user. In order to do this, we will either need to create a Docker container from within BPFContain, or allow users to assign policy files to an already running Docker container. Finally, after the container is being tracked, we will need to ensure BPFContain policy rules work for Docker containers.

At a high level, overall the project can be broken down into the following stages:

1. Investigating how Docker creates containers at a low level
2. Experimenting with eBPF probes attached to Docker components to retrieve the information required by BPFContain
3. Integrating the eBPF probes from #2 into BPFContain to allow the Docker container to be tracked
4. Updating BPFContain to optionally allow the shim to not be used, instead allowing Docker to create the container
5. Ensuring BPFContain policies work a docker container

1.3. Environment Setup

The Ubuntu 21.04 LTS Linux distribution was used for development throughout this Project. The operating system was installed using the default normal installation settings. The specific configuration steps taken to install all dependencies are listed on Table 1.3. BPFContain dependencies are listed within the project readme. One notably important step is enabling BPF LSM hooks. By default on Ubuntu 21.04 LSM hooks will not work for eBPF programs. We can use the grub configuration file to update this setting. This step is vital to allow BPFContain to actually enforce policy decisions. Additionally, we will install Docker for creating containers. The software versions of all relevant software can be seen in Table 1.2.

Table 1.2: A list of the versions used for relevant dependencies during development

Base Operating System	Ubuntu 21.04 LTS
Kernel Release	5.11.0-40-generic
Docker version	20.10.8 build 3967b7d
Runc version	1.0.1 Commit v1.0.1-g4144b63
Rust version	1.54.0 (a178d0322 2021-07-26)
Cargo version	cargo 1.54.0 (5ae8d74b3 2021-06-22)
Clang/LLVM version	12

Table 1.3: A compressive list of commands used to install the dependencies used seen in Table 3.1

BPFContain Configuration	sudo apt install build-essential
	sudo apt install libelf-dev
	sudo apt-get install gcc-multilib
	sudo apt install linux-tools-common
	sudo apt install linux-tools-5.11.0-40-generic
	sudo apt install curl
	curl https://sh.rustup.rs -sSf | sh
	sudo apt install clang-12 --install-suggests
Enable BPF LSM Hooks	sudo vim /etc/default/grub #Append “lsm=bpf” to the line GRUB_CMDLINE_LINUX_DEFAULT

	#(space separated)
	sudo grub-mkconfig -o /boot/grub/grub.cfg
	systemctl reboot
BPFContain Installation	sudo apt install git
	git clone https://github.com/willfindlay/bpfcontain-rs/
	cd bpfcontain-rs
	cargo install --path .
Docker Installation	curl -fsSL https://get.docker.com -o get-docker.sh
	sudo sh get-docker.sh

2. How Docker Containers Work

The fundamentals of Docker containers are built upon Linux control groups and namespaces [6]. Control groups (Cgroups) facilitate the amount of hardware resources a process may use [6]. Namespaces provide mechanisms to isolate a process [6]. The focus of BPFContain is container isolation, as such we will dive deeper into namespaces below.

2.1. Namespaces

Linux namespaces control the information that a process may see [7]. A process within a given namespace can only see information within its own namespace and any child namespaces [7]. From the perspective of a root process in a new namespace it appears that they are the root process of the host system. There exist eight types of namespaces which each target isolation of a specific part of Linux systems (Table 2.1). Upon system startup, one of each type of namespace is created for the root process (see Fig 2.1).

Table 2.1: List of namespace types as defined in the Linux namespace manual [7]

Namespace	Flag	Isolates
Cgroup	CLONE_NEWCGROUP	Cgroup root directory
IPC	CLONE_NEWIPC	Inter process communications
Network	CLONE_NEWNET	Network devices, stacks, ports
Mount	CLONE_NEWNS	Mount points
PID	CLONE_NEWPID	Process IDs
Time	CLONE_NEWTIME	Boot and monotonic clocks
User	CLONE_NEWUSER	User and group IDs
UTS	CLONE_NEWUTS	Hostname and NIS domain name

```
sudo lsns

        NS TYPE   NPROCS    PID USER             COMMAND
4026531834 time      299      1 root             /sbin/init splash
4026531835 cgroup    299      1 root             /sbin/init splash
4026531836 pid       298      1 root             /sbin/init splash
4026531837 user      298      1 root             /sbin/init splash
4026531838 uts       295      1 root             /sbin/init splash
4026531839 ipc       298      1 root             /sbin/init splash
4026531840 mnt       285      1 root             /sbin/init splash
4026531992 net       297      1 root             /sbin/init splash
```

Figure 2.1: The results of running lsns after startup of a Linux system. One of each type of namespace (Table 4.1) is created for the root process.

When a new Docker container is run, a new set of namespaces are created for the container. A new mount namespace isolates the container into its own root directory. The new uts namespace allows the container to define its own hostname. The IPC namespace restricts inter-process communication to only child processes within the container. The pid namespace remaps the root container process to be labeled with pid 1, with future children receiving pid's incrementing from 1. The network namespace allows the container to use its own network interfaces and routing tables. By default a new user namespace is not created, but can be created by using the --userns-remap Docker command line option[8]. A demonstration showing the new namespaces created for a new container can be seen in Fig 2.2.

Start a new Docker container

```
sudo docker run -it busybox
```

Get container root pid

```
sudo docker ps
```

```
CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS         PORTS     NAMES
3e7c4ea26226   busybox   "sh"      7 minutes ago   Up 7 minutes             cranky_bartik
```

```
sudo docker inspect 3e7c4ea26226 --format '{{.State.Pid}}'
```

```
258829
```

Lookup namespaces relating to the container

```
sudo lsns | grep 258829
```

```
 NS TYPE   NPROCS    PID USER             COMMAND
4026532389 mnt         1 258829 root             sh
4026532390 uts         1 258829 root             sh
4026532391 ipc         1 258829 root             sh
4026532392 pid         1 258829 root             sh
4026532394 net         1 258829 root             sh
```

Figure 2.2: An example of new namespaces being created for a new Docker container. After creating the container, a new mnt, uts, ipc, pid, net container is created for the container's root process.

Shown in Table 2.2, from within the container it appears as if it is the root process on the host. The container root process will appear to have a process id (pid) of 1. Whereas, from the host perspective the container root process will have a pid relative to the host pid namespace. Additionally, the container sees it's own root filesystem. From the host perspective the

container filesystem will appear as an additional mount. It is important to note that it is possible to expose files on the host filesystem to the container, however these will appear in a new mounted filesystem located within the container root filesystem.

Table 2.2 A comparison of the difference in perspective between the host system and from within a Docker container in relation to different namespace types. Listed under each namespace type is the command used to gather the information displayed. The Docker container was created with "docker run -it busybox".

	Host Perspective	**Container Perspective**
Mount (df)	Filesystem **overlay** 1K-blocks **25151748** Used **20530676** Available **3320388** Use% **87%** Mounted on **/var/lib/docker/overlay2/4882b28af2a491f80383e02485d73b20cb9fcd75938e2ccae99ff4951ac618c0/merged**	Filesystem **overlay** 1K-blocks **25151748** Used **20530676** Available **3320388** Use% **87%** Mounted on **/**
PID (ps)	PID USER TIME COMMAND **285713 root 15:01 sh**	PID USER TIME COMMAND **1 root 0:00 sh**

2.2. Docker Architecture

A typical Docker user may run a command such as “docker run image”, resulting in the creation of a container. The user can interact with the container with commands such as “docker exec”. For many user’s this will be as far as their knowledge goes in terms of Docker. In reality the user is only ever interacting with this Docker client [5]. Behind the scenes Docker is composed of many smaller services which facilitate the creation and management of containers. In this section I will provide a brief overview of the chain of internal services which lead to making system calls to create new namespaces to isolate containers.

Docker Daemon

The Docker daemon acts as a management service for Docker objects [5]. This is where the idea of objects such as images, containers, volumes comes from [5]. The daemon abstracts the internals of Docker into these easily understandable objects. When a user creates a container using a command such as “docker run image”, they are interacting with the Docker client [5]. In this context, the user is providing an image object which specifies the configuration used to create a container object. The Docker client forwards this creation request to the Docker daemon service [5]. The daemon manages the state of these Docker objects and updates according to API requests received from the client [5]. The daemon only deals with the high level management of containers. When it wants to create a new container or modify an existing container, it will send a request to the containerd service.

Containerd

We know that Docker containers are built upon the concept of Linux namespaces [6]. As such, when creating a new container there are many system calls that need to be made. Additionally, Docker offers many new features which add to the complexity of creating a new container. Containerd is a container daemon that abstracts away this complexity [9]. It was designed to be used by systems such as Docker and Kubernetes rather than actual end users [10]. Containerd acts as a manager of containers and images [10]. Images can be managed using commands such as push and pull [10]. The life cycle of containers can be managed using commands such as start and stop [10]. However, Containerd does not actually perform the low level system calls required for creating the container. Instead, Containerd sends requests to the Runc service.

Runc/Libcontainer

With the growth of container technologies came the Open Container Initiative (OCI) which defines standard formats for containers [11]. Runc is a service created for spawning and running containers according to the OCI specification [12]. Runc contains all of the code which interacts with the host system to actually create a new container[13]. Runc implements the lowest level of containers and as such is not designed for the end user [12]. Instead, it is expected that a higher level service such as Containerd will use it [12].

Diving deeper into the source code of runc exposes the libcontainer library. This library is especially relevant for the integration done in this project. Libcontainer is the library which makes the system calls to create new namespaces for a container [14]. Going even deeper there exists the nsenter package which houses the C code used to make the system calls. The overall source code is implemented in go, however it utilizes the cgo package to integrate with C code [15]. The cgo packages works in a way such that anytime the nsenter package is imported in a file the nsexec() function is called [15]. The package is only imported once inside an initialization function [15]. The nsexec.c file contains the low level implementation of how to create a new container. In Section 3, I will show how I was able to hook into the nsexec package to catch the point of namespace creation.

Overlay2 filesystem

Another vital part of Docker containers is the filesystem. Docker uses the Overlay union file system [16]. Specifically, Docker by default will use the overlay2 storage driver [16]. The overlay filesystem merges two directories together to be represented as one directory [17]. Specifically, the overlay filesystem takes an ‘upper’ filesystem and a ‘lower’ filesystem and unifies them into a ‘merged’ filesystem [17]. If the same filename exists in both directories, the file from the ‘upper’ filesystem is used and the file from the ‘lower’ filesystem is hidden [17]. Docker images are composed of many layers [16]. Upon container creation these layers are merged into one directory which is mounted to be used as the root filesystem of the container (see Fig 2.3). By default all Docker overlay filesystem directories are saved to “/var/lib/docker/overlay” [16].

Start a new Docker container

```
sudo docker run -it busybox
```

Get container id

```
sudo docker ps
```

```
CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS         PORTS     NAMES

161cfbe4919c   busybox   "sh"      5 minutes ago   Up 5 minutes             eager_mclaren
```

Lookup Docker filesystem information

```
sudo docker inspect 161cfbe4919c
```

```
"Data": {
  "LowerDir":
"/var/lib/docker/overlay2/32533b7abef03d82ac7cb5c68f7032d07110f8e6765f6570caa2e6
e7d05cdc8b-init/diff:/var/lib/docker/overlay2/9403c2ca142f59e52027c067d49112f483
9113d1315e57510867faf030311c3f/diff",
  "MergedDir":
"/var/lib/docker/overlay2/32533b7abef03d82ac7cb5c68f7032d07110f8e6765f6570caa2e6
e7d05cdc8b/merged",
  "UpperDir":
"/var/lib/docker/overlay2/32533b7abef03d82ac7cb5c68f7032d07110f8e6765f6570caa2e6
e7d05cdc8b/diff",
  "WorkDir":
"/var/lib/docker/overlay2/32533b7abef03d82ac7cb5c68f7032d07110f8e6765f6570caa2e6
e7d05cdc8b/work"
      },
"Name": "overlay2"
```

Check which directory was mounted as the root filesystem for the container

```
df
```

```
Filesystem overlay
Mounted on
/var/lib/docker/overlay2/32533b7abef03d82ac7cb5c68f7032d07110f8e6765f6570caa2e6e
7d05cdc8b/merged
```

Figure 2.3: An example demonstrating how a Docker container consists of multiple directories, which are merged to become the container root filesystem.

3. eBPF uprobe experimentation

In this section, I will demonstrate my attempts to hook into the container creation process using uprobes to retrieve the information required by BPFContain as outlined in Section 1.2.

3.1. Trace Container Creation

To begin the investigation, we will trace the execve system calls made when starting a new Docker container. Observe that many different Docker services are called (Fig 3.1). As mentioned in Section 2.2, dockerd, containerd and runc are all part of the creation process. Additionally, we observe that the container's root process comes from runc. Immediately before “sh 259739” is called, we see “runc:[2:INIT] 259739” both with a pid of 259739. This aligns with our understanding of runc, as it is where the low level system calls are made. At this point we have the ability to retrieve the pid and gid of the container. However, complications arise with determining how to filter through all execve invocations to only catch the new container process. Instead, we will look at hooking directly into a runc function. We want to find a function that hooks directly into the container creation process. From within the uprobe, we should be able to retrieve all of the information required by BPFContain through the current task_struct.

Start tracing the execve system call

```
sudo bpftrace -e 'tracepoint:syscalls:sys_*exec* { printf("%s %d\n", comm, pid); }'
```

```
Attaching 8 probes...
```

Start a new container

```
sudo docker run -it busybox
```

Get containers root pid

```
sudo docker ps
```

```
CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS         PORTS     NAMES
51b673f65371   busybox   "sh"      2 minutes ago   Up 2 minutes             upbeat_euler
```

```
donut@donut-VirtualBox:~$ sudo docker inspect 51b673f65371 --format '{{.State.Pid}}'
```

```
259739
```

Observe traced system calls:

```
bash 259692
sudo 259692
sudo 259693
docker 259693
modprobe 259702
modprobe 259703
(spawn) 259706
systemd-sysctl 259706
(spawn) 259709
systemd-sysctl 259709
containerd 259710
containerd-shim 259710
containerd-shim 259717
containerd-shim 259717
containerd-shim 259729
runc 259729
runc 259736
exe 259736
exe 259736
7 259736
runc 259745
exe 259745
dockerd 259752
exe 259752
containerd-shim 259759
runc 259759
runc:[2:INIT] 259739
sh 259739
```

Figure 3.1: An example of the execve system calls made when starting a new Docker container. The trace was done using the BPFTrace tool.

3.2. Uprobe into runc

```
sudo find / -name runc

/usr/bin/runc
```

Figure 3.2: Using the find command to locate the runc binary

Hooking into nsexec

As described in Section 2.2, the nsexec() function contains the source code which creates new namespaces for a Docker container. As such, we will attempt to attach an uprobe onto it and use it to retrieve the newly created namespace ids. One observation made while experimenting with uprobes in Golang is that many functions are inlined at compile time [18]. As such, we must confirm that the nsexec function is not inlined in order to attach an uprobe to it. Seen in Fig 3.3 we confirmed that nsexec is not inlined and we are able to attach

an uprobe to it. However, after starting a new container we notice that none of the processes hooked were the root container process. It appears this function is called too early, the root container process is created within this function. Instead, we want to find the function which creates the runc:[2:INIT] process seen in Fig 3.1.

Look for nsexec symbol in runc

```
objdump -tT /usr/bin/runc | grep nsexec
```

```
000000000073c460 g    DF .text   00000000000016e0  Base        nsexec
```

Attach a uprobe to nsexec

```
sudo bpftrace -e 'uprobe:/usr/bin/runc:nsexec { printf("%d, %s %s\n", pid, comm, probe); }'
```

```
Attaching 1 probe...
```

Start a new container

```
sudo docker run -it busybox
```

Get container root pid

```
sudo docker ps
```

```
CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS         PORTS     NAMES
cb14913a784f   busybox   "sh"      6 minutes ago   Up 6 minutes             condescending_robinson
```

```
sudo docker inspect cb14913a784f --format '{{.State.Pid}}'
```

```
260288
```

Observe traced output

```
260277, runc uprobe:/usr/bin/runc:nsexec
260285, exe uprobe:/usr/bin/runc:nsexec
260285, 7 uprobe:/usr/bin/runc:nsexec
260310, runc uprobe:/usr/bin/runc:nsexec
```

Figure 3.3: Attaching an uprobe onto the nsexec function in runc using BPFTrace to trace the function invocations when starting a new Docker container.

Hooking int x_cgo_init

As a next step we can search for any init functions within runc. Seen in Fig 3.4, we attached an uprobe to all init functions within runc. After starting a new Docker container, we observe that many of the init functions are invoked. Additionally, we see the runc:[2:INIT] process correlated to the container root process pid. The corresponding uprobe comes from the x_cgo_init function. Now we want to confirm that at this point, we are able to retrieve all of the new namespace ids relating to the container. Seen in Fig 3.5, we can attach a uprobe to

the x_cgo_init function and retrieve the pid namespace out of the current task struct for the process on each invocation. After starting a new Docker container, we observe that x_cgo_init is called multiple times. Looking back at Fig 3.4, we see that x_cgo_init is called three times. However, only one invocation is for the runc:[2:INIT] process. Looking at the process namespace ids for each, we see the process namespace id for the runc:[2:INIT] is infact the new namespace created for the Docker container. The other two invocations have the same process namespace as the host system. As a result, we should be able to use this uprobe within BPFContain to hook into the Docker container creation process. Within this uprobe we can retrieve the information required by BPFContain and add it into the inteneral maps. After BPFContain has the container information it will be able to begin enforcing policy rules.

Look for init symbol in runc

```
objdump -tT /usr/bin/runc | grep init
```

```
0000000000000000      DF *UND*   0000000000000000              seccomp_init
0000000000000000      DF *UND*   0000000000000000  GLIBC_2.2.5 pthread_attr_init
000000000073ec60 g    DF .text   0000000000000065  Base        __libc_csu_init
000000000073df30 g    DF .text   00000000000000cb  Base        x_cgo_init
000000000073dd10 g    DF .text   000000000000003a  Base        x_cgo_notify_runtime_init_done
0000000000c4ccd8 g    DO .bss    0000000000000008  Base        x_cgo_inittls
000000000026b770 g    DF .text   000000000000000b  Base        init
000000000073dc70 g    DF .text   000000000000009f  Base        _cgo_wait_runtime_init_done
000000000073e5b0 g    DF .text   000000000000002f  Base        _cgo_e3ef74848b91_Cfunc_seccomp_init
```

Attach a uprobe to init functions

```
sudo bpftrace -e 'uprobe:/usr/bin/runc:*init* { printf("%d, %s %s\n", pid, comm, probe); }'
```

```
Attaching 6 probes...
```

Start a new container

```
sudo docker run -it busybox
```

Get container root pid

```
sudo docker ps
```

```
CONTAINER ID   IMAGE     COMMAND   CREATED         STATUS         PORTS     NAMES

8a66b9464f59   busybox   "sh"      8 minutes ago   Up 8 minutes             magical_hermann
```

```
sudo docker inspect 8a66b9464f59  --format '{{.State.Pid}}'
```

```
260485
```

Observe traced output

```
260470, runc uprobe:/usr/bin/runc:__libc_csu_init
260470, runc uprobe:/usr/bin/runc:init
260470, runc uprobe:/usr/bin/runc:x_cgo_init
260470, runc uprobe:/usr/bin/runc:x_cgo_notify_runtime_init_done
260481, exe uprobe:/usr/bin/runc:__libc_csu_init
260481, exe uprobe:/usr/bin/runc:init
260481, 7 uprobe:/usr/bin/runc:__libc_csu_init
260481, 7 uprobe:/usr/bin/runc:init
260485, runc:[2:INIT] uprobe:/usr/bin/runc:x_cgo_init
260485, runc:[2:INIT] uprobe:/usr/bin/runc:x_cgo_notify_runtime_init_done
260485, runc:[2:INIT] uprobe:/usr/bin/runc:_cgo_e3ef74848b91_Cfunc_seccomp_init
260504, runc uprobe:/usr/bin/runc:__libc_csu_init
260504, runc uprobe:/usr/bin/runc:init
260504, runc uprobe:/usr/bin/runc:x_cgo_init
260504, runc uprobe:/usr/bin/runc:x_cgo_notify_runtime_init_done
```

Figure 3.4: Attaching an uprobe onto all init functions in runc using BPFTrace to trace the function invocations when starting a new Docker container.

Attach a uprobe to x_cgo_init

```
sudo bpftrace -e 'uprobe:/usr/bin/runc:x_cgo_init {
                      printf("%d, %s %u\n", pid, comm,
                      curtask->nsproxy->pid_ns_for_children->ns.inum);
                }'
```

```
Attaching 1 probe...
```

Start a new container

```
sudo docker run -it busybox
```

Lookup the id of the pid namespace created for the container

```
sudo lsns
```

```
4026532319 pid         1 260786 root             sh
```

Observe traced output

```
260774, runc 4026531836
260786, runc:[2:INIT] 4026532319
260811, runc 4026531836
```

Figure 3.5: Attaching an uprobe onto the x_cgo_init function in runc using BPFtrace to view the process namespace id related to each function invocation made when starting a new Docker container.

4. Integration

4.1. Architectural Overview

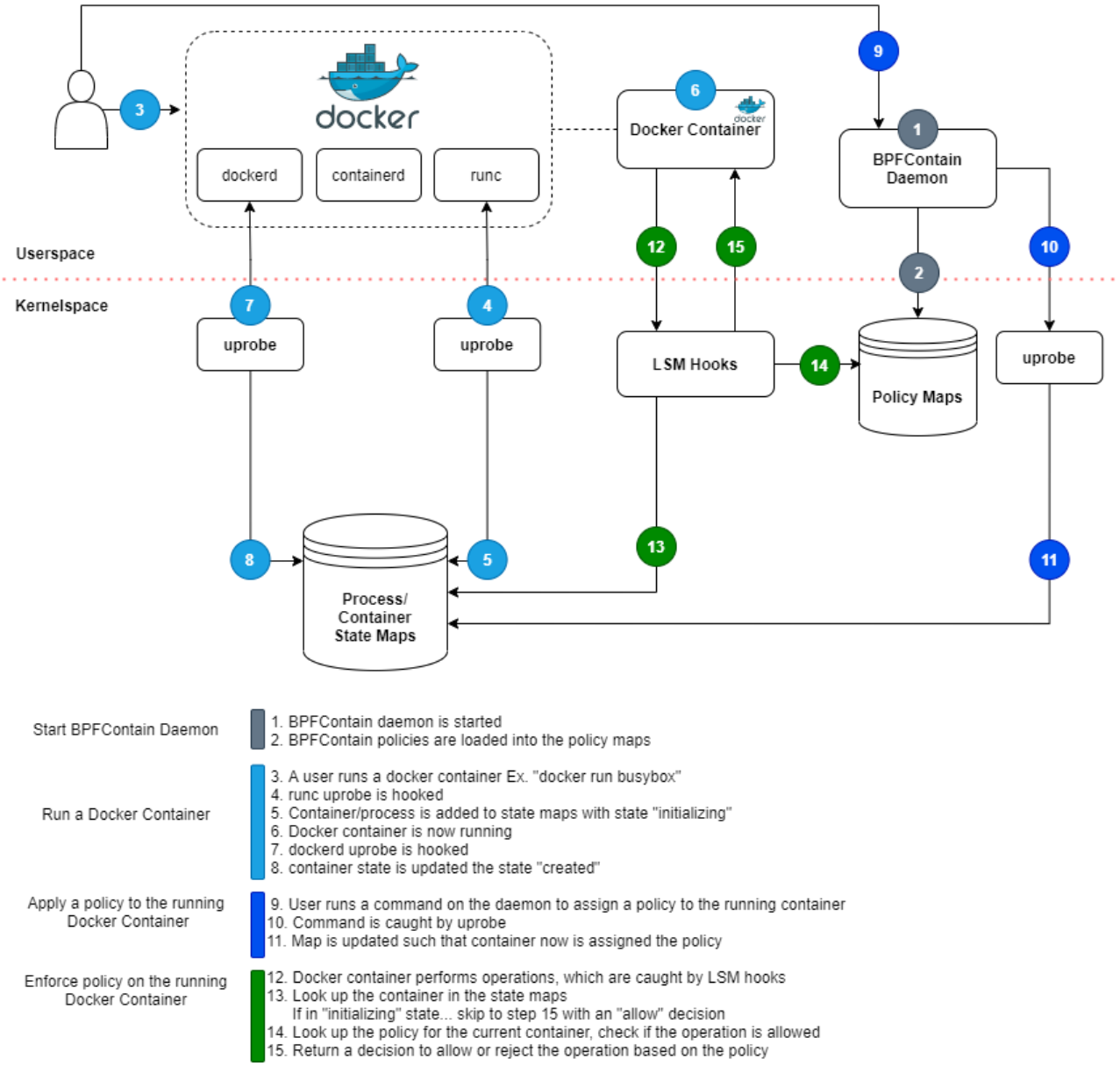

Figure 4.1: An overview of how the Docker integration works within BPFContain

Overall, the integration architecture as shown in Fig 4.1 consists of four stages:

1. Start BPFContain Daemon

First, the BPFContain Daemon must be started to load our eBPF programs and yaml policies.

2. Run a Docker Container

The second stage starts by a user running a new Docker container using the Docker CLI. The request is sent to the Docker daemon, followed by containerd which reaches out to Runc as described in Section 2.2. The runc service makes the low level system calls to create a new container. At this point, the runc uprobe is invoked and the container information such as it's process id, and namespace ids are added to the state maps. The container is assigned to an empty policy.

Now BPFContain has the information required to start enforcing policy rules on the Docker container. However, at this point the container is not fully created. There are many more system calls that will be made to fully set up the container. In order to allow these system calls to be made we set the container to be in an "initialization" state. In this state, all operations made by the Docker container will be allowed by the LSM hooks. Additionally, in this stage any mount requests made for the Docker container will be added to the implicit filesystem policy assigned to the container.

To know when we want to begin enforcing the policy we attach an uprobe onto the Docker daemon service. The uprobe is attached to a function which is invoked after the container is fully created. In the uprobe, we look up the corresponding container in the state maps, and update it to a "created" state. In this state, the LSM hooks will begin querying the policy maps to determine if operations are allowed. The policy assigned to the container at this point only allows access to the mounted filesystem through the implicit filesystem rules. Any other operations will be denied, resulting in the container being severely restricted for now.

3. Assign a policy to the Docker container

In the third stage a user will assign a new policy to the running container. The default policy is quite restrictive, so it is likely that the user will want to assign a new policy to the container so that it can perform as intended. The user will make a request to the BPFContain CLI providing the process id of the container, along with the policy name. The request will be caught by an uprobe, which uses the arguments to update the container in the state maps to use the new policy. Note that the implicit file system rules will transfer through policy changes.

4. Enforce policy rules on the Docker container

Now the Docker container has its policy assigned and is being tracked by BPFContain's internal maps. The LSM hooks now have the necessary information to enforce the policy rules. For example, when the Docker container makes system call requests, the LSM hooks will be invoked at which point it will lookup the corresponding container in the state maps. Using that information it will look in the policy maps to determine if the operation is allowed for the container. The LSM hook returns a response either allowing or denying the request.

4.2. Hooking into runc

As described in Section 3, the runc:x_cgo_init uprobe hooks into the container creation process at a point where the new namespaces have been created. In the context of this uprobe, we can retrieve the namespace information required through the current task struct.

In order to attach an uprobe in eBPF, we must specify the function symbol address. In the experimentation done with bpftrace, the user space symbol lookup is done automatically for us. However, using libbpf we need to implement the logic required for this ourselves. Luckily, the libbpf-bootstrap github repository contains an example created by Magnus Kulke doing exactly this [19]. The example contains a function, "get_symbol_address" which will lookup the symbol address for a given function name and binary path. We can attach our uprobe using the symbol address returned. For our purposes, this function was copied into the BPFcontain source code.

One complexity that came up in experimentation was that this uprobe is called multiple times. Looking at the runc source code directly we can look into why this happens [20]. However, for our purposes we don't need to worry about the specifics, we simply just want to ignore the extra function calls. To achieve this, we can look up the pid namespace and pid for children namespace in each invocation. The call we are interested in will have a newly created pid for children namespace. Whereas the other calls will have the same pid for children namespace as their parent pid namespace id. Thus, we will ignore any calls to this uprobe if their pid namespace id is the same as their pid for children namespace id. Another approach considered was to check if the process common name was "runc:[2:INIT]" and ignore any other invocations. However, it was decided to not use this option as if the runc source code changed this name, then our code will not work.

To add our entries into the container and process eBPF hashmaps, the majority of the code was reused from the container shim start function. However, there are some key differences which led me to create a new start_docker_container function. The first major difference is the policy_id is set to the unique container_id created for the container. Thus, when the container starts it will have an empty policy. Additionally, we set the Docker containers to be tainted by default because there are no taint rules in the default policy they are assigned.

As mentioned in the overview, there will still be many more system calls made after this uprobe to finish the initialization of the container. Any system calls made during the initialization stage should be allowed. However, after the container is fully created we want to confine it using the policy assigned. As a result, we add an attribute to the container struct which tracks the state of a container (Fig 4.2). If the container is in the initialization state, then the LSM hooks will allow any operation requested, and if in the started state it will enforce BPFContain policy decisions. It is important to note that this does place trust in the underlying Docker services. However, in practice any company using Docker already places an implicit trust that Docker itself is non malicious.

```
typedef enum {
    DEFAULT_SHIM = 0,
    DOCKER_INIT = 1000,
    DOCKER_STARTED = 1001,
} container_status_t;
```

Figure 4.2: The container status enum created to track Docker container creation stages.

4.3. Hooking into Docker Daemon

In order to update the container status to created, we need to know when the container has finished initializing. To find a suitable function to attach an uprobe, we need to perform similar experimental steps as what was done to find the runc uprobe. In this case we can look at the Docker source code to see what the Docker daemon does when starting a new container. In the source code we find a function called “containerStart” which makes a request to containerd. After this creation step, we see a simple call to a function called “SetRunning” [21]. The function appears to simply update the container related to the provided pid to a status of “running” [22]. This appears to be a perfect place to hook into to know when the container is created. At this point the container is fully created, and we can use the pid argument to lookup the container within BPFContain. Searching for the function

symbol in the dockerd binary we find:
“github.com/docker/docker/container.(*State).SetRunning”.
We attach an uprobe to this function the same way it was done in Section 4.2. When this uprobe is invoked we need to grab the pid from the functions arguments. It turns out this is more complex than expected because Docker daemon is written in Golang. Golang stores arguments on the stack [18]. To retrieve the argument off the stack, we can access it from the context struct in the uprobe. Searching for how to do this we find the input argument is stored in the ‘ax’ attribute [23]. Thus, we can successfully retrieve the pid argument on uprobe invocation. Using the pid, we can lookup the container from the eBPF hash maps and update it’s state to “created”.

4.4. Setting the hostname

One thing that was learned during the creation of the runc uprobe, was that the hostname within the container was still the same as the host machine at the point in which our uprobe is invoked. It turns out that the hostname is updated to match the Docker container id later on in the initialization processes. To allow us to track the updated hostname within BPFContain, we can add a tracepoint hook on the sethostname system call, specifically “tp/syscalls/sys_enter_sethostname”. To know which argument contained the name, we can look up the tracepoint information on the system at: “/sys/kernel/debug/tracing/events /syscalls/sys_enter_sethostname/format”. Within the eBPF code, we can lookup the container associated with the current pid to update it’s hostname to what is provided in the arguments. Now, after the Docker container is created we are able to retrieve it’s hostname easily.

4.5. Implicit Filesystem Rules

A problem faced while testing BPFContain policies with Docker was the file system rules. In BPFContain file system rules are defined in the policy using filenames. When the policies are loaded by the daemon, each file system rule is parsed. The inode and device number of the file/directory is found and stored in the file system policy map. Within the LSM hooks, when a file is accessed by a confined process, it’s inode and device number are compared to the policy map to determine if access should be granted. All BPFContain policies are loaded upon the daemon startup. This presents a problem for Docker containers because at the time of daemon startup our Docker containers do not exist. As such, their file systems do not exist. Thus, when loading the policy we are unable to retrieve the inode numbers for files inside the Docker container.

In an attempt to resolve this issue, we can try some experimental changes to BPFContain behaviour to periodically poll the policy folder, each time loading the policies into eBPF maps. This behaviour would allow us to create a Docker container, and then create a policy for it using it's newly created file system. One caveat is that the file path in the policy may be confusing because it must be relative to the host machine rather than the Docker container. For example the file path might be "/var/lib/docker/overlay2/<containerid>/merged" instead of just "/". Additionally, we would be required to create a custom policy for every single Docker container we create which is impractical for real world use. Importantly there are also inherent security concerns with continuously loading policies into eBPF maps because if an existing policy is changed, then it will be added to the eBPF maps. However, the existing rules will remain, resulting in legacy rules which should not exist. It is possible for us to resolve this behaviour by only loading policies which don't already exist inside the maps. However, overall this solution proves to be complicated and does not satisfy usability requirements.

Instead we can proceed with another direction in which we implicitly add the file system rules for Docker containers. Whenever a new Docker container is created, we will add implicit access to any file system mounted during creation. Using the container status flag we can tell if the container is in it's initialization state. By hooking into the mount system call, we can catch any mount requests made by the container during initialization. Any mount system calls made after initialization will be rejected. To track the implicit rules we use a new filesystem map. It's implementation is similar to the existing filesystem map, however instead of using the policy id as it's key, it uses the container id. This allows the rules to follow containers across new policy assignments. This solution aligns closer to how users will use the system. If a user specifies mounts when creating the container, then it is expected that they would want access to such mounts. Note that we are still providing additional security, as we prevent break out attacks from accessing file systems external to the container. For example, an attacker inside a basic Docker container would be unable to access the host filesystem. As a result, we will proceed with this solution. The file system rules will be created implicitly, and the remaining rule types will be defined with a policy yaml file.

4.6. Applying a new policy to a Docker container

When a new Docker container is created, it is captured by BPFContain. The only permissions it receives are the implicit filesystem rules from it's mounts. As such, the container will not be able to do much. After creation, a user will want to assign a new policy to the container which defines it's intended usage. To implement this behaviour we will follow the implementation of how users can start a new container using the BPFContain shim. The user will make a request such as: "bpfcontain confine process id policy_name", which will be caught by an uprobe. The uprobe will lookup the container related to the process id provided. We then update the container's policy to be the new policy provided. Any actions performed by the container in the future will use the new policy id for looking up policy rules.

5. Demonstration

Fig 5.1 is a basic example demonstrating how to confine a Docker container using BPFContain. First the BPFContain daemon must be started at which point the device.yml policy file is loaded. Now we create a new Docker container using the familiar Docker client. BPFContain catches the new container and begins tracking it. At this point when running basic commands in the container, we are met with “Permission denied” errors. The implicit policy assigned to the container is extremely restrictive, only providing access to the container mounts. We can now apply the docker-device policy (Table 5.1) which adds permission to use the terminal device. Now when running the same command again, we see access was granted and we see the container root filesystem.

Figure 5.2 demonstrates how we can use BPFContain to confine a basic web server. In this case we start a basic http web server using a python inside a Docker container. After creation we apply the docker-server (Table 5.1) policy to allow the web server to receive and respond to requests. After applying the policy, we are now able to request files from the server successfully.

Table 5.1: A list of example policies demonstrating various rule types

Device permissions (device.yml)	`name: docker-device`
	
	`allow:`
	`  - dev: terminal`

Capabilities (capability.yml)	`name: docker-cap`
	
	`allow:`
	`  - dev: terminal`
	`  - capability: [setUID, setGID]`

Network (server.yml)	`name: docker-server`
	
	`allow:`
	`  - dev: terminal`
	`  - net: [server, send, recv]`

Start the BPFContain daemon

```
sudo bpfcontain daemon fg
```

```
[INFO]: Running in the foreground...
[INFO]: Loading policy recursively from /var/lib/bpfcontain/policy...
[INFO]: Loading policy from file /var/lib/bpfcontain/policy/device.yml...
[INFO]: Done loading policy!
```

Start a Docker container

```
docker run -it busybox
```

Trying running ls - Permissions are heavily restricted

```
/ # ls
```

sh: can't set tty process group: Permission denied
sh: can't set tty process group: Permission denied
[1]+ Done(2)

Apply a policy to the container

```
sudo docker ps
```

```
CONTAINER ID   IMAGE     COMMAND   CREATED          STATUS          PORTS     NAMES
7d109859d884   busybox   "sh"      17 seconds ago   Up 16 seconds             cool_hawking
```

```
sudo docker inspect 7d109859d884 --format '{{.State.Pid}}'
```

```
15292
```

```
bpfcontain confine 15292 device.yml
```

```
Applying policy now "15292" "device.yml"
Policy Id 2234808277196645527
```

Trying running ls

```
/ # ls
```

```
/ # bin   dev   etc   home  proc  root  sys   tmp   usr   var
```

Figure 5.1: A base example demonstrating how to confine a Docker container using BPFContain.

Start the BPFContain daemon

```
sudo bpfcontain daemon fg
```

```
[INFO]: Running in the foreground...
[INFO]: Loading policy recursively from /var/lib/bpfcontain/policy...
[INFO]: Loading policy from file /var/lib/bpfcontain/policy/server.yml...
[INFO]: Done loading policy!
```

Start Python Docker container

```
sudo docker run -p 9000:9000 -it python bash
```

Apply server.yml policy to the container

```
sudo docker ps
```

```
CONTAINER ID   IMAGE     COMMAND    PORTS
2034ead7c760   python    "bash"     0.0.0.0:9000->9000/tcp, :::9000->9000/tcp
```

```
sudo docker inspect 2034ead7c760 --format '{{.State.Pid}}'
```

```
21377
```

```
bpfcontain confine 21377 server.yml
```

```
Applying policy now "21377" "server.yml"
Policy Id 638762433925158055
```

Start python server

```
python -m http.server 9000
```

```
Serving HTTP on :: port 9000 (http://[::]:9000/) ...
```

Browse to server

```
curl http://localhost:9000
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<title>Directory listing for /</title>
</head>
<body>
<h1>Directory listing for /</h1>
<hr>
<ul>
<li><a href=".dockerenv">.dockerenv</a></li>
<li><a href="bin/">bin/</a></li>
<li><a href="boot/">boot/</a></li>
<li><a href="dev/">dev/</a></li>
<li><a href="etc/">etc/</a></li>
<li><a href="home/">home/</a></li>
<li><a href="lib/">lib/</a></li>
```

```
<li><a href="lib64/">lib64/</a></li>
<li><a href="media/">media/</a></li>
<li><a href="mnt/">mnt/</a></li>
<li><a href="opt/">opt/</a></li>
<li><a href="proc/">proc/</a></li>
<li><a href="root/">root/</a></li>
<li><a href="run/">run/</a></li>
<li><a href="sbin/">sbin/</a></li>
<li><a href="srv/">srv/</a></li>
<li><a href="sys/">sys/</a></li>
<li><a href="tmp/">tmp/</a></li>
<li><a href="usr/">usr/</a></li>
<li><a href="var/">var/</a></li>
</ul>
<hr>
</body>
</html>
```

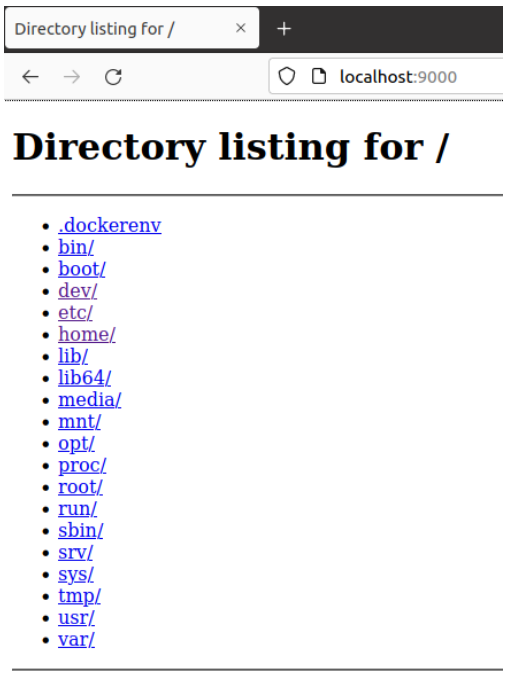

Figure 5.2: An example demonstrating how to use BPFContain to confine a python web server running with Docker

6. Performance benchmarking

In this section, we will perform some basic performance benchmark tests to determine the performance impact of BPFContain on Docker containers. First we will look at the impact of the changes specific to this project. Specifically, we will look at the impact on Docker container creation times. Secondly, we will analyze the overall runtime impact that BPFContain has on a running Docker container.

To observe the impact of BPFContain on Docker container creation, we will record the time taken to start a number of Docker containers. For each quantity of containers we ran the test 10 times and recorded the average overall. We observed minimal impact on the creation of Docker containers by BPFContain. The tests were run for the creation of 1, 10 and 100 containers with results shown in Table 6.1. The largest percent increase found was 1.70% representing an increase of around 6.96 ms to create 1 container. An increase of 1.60% was observed for the creation of 100 containers, which amounts to around a 7.89ms increase per container. As a result, we see that the overall impact of BPFContain on the creation of Docker containers is minimal.

Another important consideration is the runtime impact of BPFContain on Docker containers. We will utilize the Phoronix Test Suite to run osbench and apache benchmarking on running containers with and without BPFContain running. To prioritize accuracy we will run each suite in strict mode, running as many tests as required. The OSBench suite performs a series of tests to measure performance of operating system functionality [24]. The results of OSBench demonstrate some overhead compared to an unconfined Docker container (Table 6.2). The largest overhead seen is an increase of 37.37% for “Create files” and 16.11% for “Launch Programs”. The difference is likely attributed due to the significant increase in work required by BPFContain for file system rules. The apache tests suite attempts to make as many web requests as possible within a set amount of time [25]. The results of the Apache test suite demonstrate a small overhead compared to an unconfined Docker container (Table 6.3). This test demonstrated a small impact of performance caused by the networking rules in BPFContian. To provide some context into how these results may compare with competitors; William Findlay has analyzed the overall overhead of eBPF policies which found bpf probe overhead to be lower when compared to AppArmor [26].

Table 6.1: The time taken to start containers with and without BPFContain running (lower is better). The results given represent the average out of 10 runs for each. Percent differences are given in parentheses.

	Docker Baseline	**Docker with BPFContain**
Create 1 container (s)	0.4060841	0.4130439 (+1.70%)
Create 10 containers (s)	4.1914438	4.2053800 (+0.33%)
Create 100 containers (s)	48.9104532	49.6997021 (+1.60%)

Table 6.2: The results of Phoronix OSBench benchmarking with and without BPFContain (lower is better). Percent differences are given in parentheses. The BPFPolicy used can be seen in Fig 8.1.

	Docker Baseline	**Docker with BPFContain**
Create Files (µs)	41.56	60.66 (+37.37%)
Create Threads (µs)	23.05	25.95 (+11.79%)
Launch programs (µs)	218.02	256.22 (+16.11%)
Create Processes (µs)	53.92	55.15 (+2.26%)
Memory Allocations (µs)	104.03	104.27 (+0.73%)

Table 6.3: The results of Phoronix Apache benchmarking with and without BPFContain (higher is better). Percent differences are given in parentheses. The BPFPolicy used can be seen in Fig 8.1.

	Docker Baseline	**Docker with BPFContain**
Requests Per Second (*r*/*s*)	5014.78	4691.96 (-6.65%)

7. Discussion

7.1. Security improvement

Integrating Docker with BPFContain provides an added layer of security to Docker containers. BPFContain denies all actions unless otherwise specified. As such, containers will be heavily restricted by default. This design improves Docker's ability to follow the principle of least privilege. Containers should only have access to the specific operations required for their use case. For example, any Docker container created will be unable to make any networking operations unless otherwise specified in a policy. Additionally, Docker containers will not be able to access any files outside of its own defined mount points. As a result, we provide added protection from Docker container escape vulnerabilities. If there is a use case which requires the host machine filesystem to be mounted then we can use policy rules to restrict the access to only a subset of folders. Overall, the integration provides a valuable option for securing Docker containers.

7.2. Power of eBPF

Furthermore, this project demonstrates the power of eBPF programs. Leveraging eBPF uprobes we are able to integrate directly with third party services without requiring any source code changes on their end. This demonstrates a proof of concept showing that BPFContain has the potential to be integrated with any container technology. The performance benchmarking further demonstrates how eBPF programs can be created with minimal overhead to system performance. Overall, the project serves as encouragement for future use of eBPF in security systems.

7.3. Future work

Apply Policies Using Docker Container ID

Currently we are able to apply a policy to a running container by providing it's root process pid. By using the process id we allow containers using the default shim to utilize the policy change function. However, when working with Docker containers users are familiar with identifying their containers using it's container id rather than pid. Currently a user must lookup the container id and then use it to lookup it's pid. To improve the user experience, an improvement would be to create a command which allows users to use a Docker container id to update it's policy. This improvement would automate the pid lookup. One idea for implementing this feature is to use the Docker Rust SDK [27].

Deny all Access to Policy Files From Within Containers

BPFContain policies are stored in /var/lib/bpfcontain/policy. There shouldn't be any reason for a container to require access to these files. If a container could modify one of these files it opens the door for the container to potentially modify its own policy for privileged escalation. To prevent this completely, we could prevent all access to these files by default for confined containers.

Load Policies after BPFContain Daemon Creation

Currently policies must be created before the BPFContain daemon is created. If a user wants to add a new policy, the daemon must be restarted. In a production system it is likely a user may want to update or add a new policy while the system is in use. Instead of requiring the daemon to be restarted, we can look for a solution to allow policies to be loaded or updated at any time.

Integration with other Container technologies

The integration with Docker lays out a base example demonstrating how BPFContain can be integrated with third party container technologies. To further build on BPFContain, we can look to provide integration with technologies such as Kubernetes, Helm, and Docker Swarm. Any technologies which use runc can benefit from the existing hooks, and only require integration with management layers to know when the container has finished being created.

8. Conclusion

In this project, I have shown the design of integration of Docker with BPFContain. The integration allows Docker containers to be confined by BPFContain policy files. Confinement by BPFContain provides an added layer of security, restricting most operations unless otherwise specified. Benchmarking tests show that the overhead added by the integration is minimal, demonstrating its ability to be used at scale. Overall, this integration showcases a viable option for adding security to Docker containers. Additionally, the project demonstrates the power of the eBPF technology. I was able to integrate with Docker without requiring any source code changes within Docker. Using uprobes presents a powerful way to integrate into any third party technology.

References

[1] "What is eBPF? an introduction and deep dive into the EBPF technology," eBPF Documentation. [Online]. Available: https://ebpf.io/what-is-ebpf/. [Accessed: 29-Nov-2021].

[2] "LSM BPF programs," The Linux Kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/bpf_lsm.html. [Accessed: 29-Nov-2021].

[3] W. Findlay, A. Somayaji, and D. Barrera, "BPFCONTAIN: Fixing the Soft Underbelly of Container Security," Cornell University, 13-Feb-2021. [Online]. Available: arxiv.org. [Accessed: 29-Nov-2021].

[4] S. Hykes, "Docker to donate containerd to the Cloud Native Computing Foundation," Docker Blog, 15-Mar-2017. [Online]. Available: https://www.docker.com/blog/docker-donates-containerd-to-cncf/. [Accessed: 29-Nov-2021].

[5] "Docker Overview," Docker Documentation, 26-Nov-2021. [Online]. Available: https://docs.docker.com/get-started/overview/. [Accessed: 29-Nov-2021].

[6] "Docker Security," Docker Documentation, 26-Nov-2021. [Online]. Available: https://docs.docker.com/engine/security/. [Accessed: 29-Nov-2021].

[7] "Namespaces(7) - linux manual page," Linux Programmer's Manual, 27-Aug-2021. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html. [Accessed: 29-Nov-2021].

[8] "Isolate containers with a user namespace," Docker Documentation, 26-Nov-2021. [Online]. Available: https://docs.docker.com/engine/security/userns-remap/. [Accessed: 29-Nov-2021].

[9] M. Crosby, "What is containerd ?" Docker Blog, 07-Aug-2017. [Online]. Available: https://www.docker.com/blog/what-is-containerd-runtime/. [Accessed: 29-Nov-2021].

[10] Containerd, "Containerd README," containerd, 19-Nov-2021. [Online]. Available: https://github.com/containerd/containerd/blob/main/README.md. [Accessed: 29-Nov-2021].

[11] "About the open container initiative," opencontainers.org. [Online]. Available: https://opencontainers.org/about/overview/. [Accessed: 29-Nov-2021].

[12] Opencontainers, "RUNC Readme," runc, 13-Oct-2021. [Online]. Available: https://github.com/opencontainers/runc/blob/master/README.md. [Accessed: 29-Nov-2021].

[13] S. Hykes, "Introducing runc: A lightweight universal container runtime," Docker Blog, 23-Jun-2015. [Online]. Available: https://www.docker.com/blog/runc/. [Accessed: 29-Nov-2021].

[14] Opencontainers, “libcontainer Readme,” runc - libcontainer, 10-Apr-2021. [Online]. Available: https://github.com/opencontainers/runc/blob/master/libcontainer/README.md. [Accessed: 29-Nov-2021].

[15] Opencontainers, “nsenter Readme,” runc - libcontainer - nsenter, 07-Aug-2018. [Online]. Available: https://github.com/opencontainers/runc/blob/master/libcontainer/nsenter/README.md. [Accessed: 29-Nov-2021].

[16] “Overlayfs storage driver,” Docker Documentation, 26-Nov-2021. [Online]. Available: https://docs.docker.com/storage/storagedriver/overlayfs-driver/. [Accessed: 29-Nov-2021].

[17] N. Brown, “Overlay Filesystem,” The Linux Kernel documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html. [Accessed: 29-Nov-2021].

[18] B. Gregg, “Golang bcc/BPF Function Tracing,” Brendan Gregg's Blog, 31-Jan-2017. [Online]. Available: https://www.brendangregg.com/blog/2017-01-31/golang-bcc-bpf-function-tracing.html. [Accessed: 29-Nov-2021].

[19] M. Kulke, “libbpf tracecon example,” libbpf-bootstrap, 03-Jun-2021. [Online]. Available: https://github.com/libbpf/libbpf-bootstrap/blob/master/examples/rust/tracecon/src/main.rs#L47. [Accessed: 29-Nov-2021].

[20] A. Sarai, “Runc nsexec.c,” runc, 31-Oct-2016. [Online]. Available: https://github.com/opencontainers/runc/blob/master/libcontainer/nsenter/nsexec.c#L917. [Accessed: 29-Nov-2021].

[21] Docker, “Docker-CE start.go ,” cocker-ce, 11-Jun-2021. [Online]. Available: https://github.com/docker/docker-ce/blob/270cf46aa5877653bb2b616eba1498a3657127d5/components/engine/daemon/start.go#L209. [Accessed: 29-Nov-2021].

[22] Docker, “Docker-CE state.go,” docker-ce, 05-Aug-2019. [Online]. Available: https://github.com/docker/docker-ce/blob/270cf46aa5877653bb2b616eba1498a3657127d5/components/engine/container/state.go#L267. [Accessed: 29-Nov-2021].

[23] Z. Asgar, “Debugging with EBPF part 1: Tracing go function arguments in prod,” Pixie Labs Blog, 10-Nov-2020. [Online]. Available: https://blog.px.dev/ebpf-function-tracing/. [Accessed: 29-Nov-2021].

[24] “Osbench,” OpenBenchmarking.org. [Online]. Available: https://openbenchmarking.org/test/pts/osbench. [Accessed: 29-Nov-2021].

[25] “Apache HTTP server,” OpenBenchmarking.org. [Online]. Available: https://openbenchmarking.org/test/pts/apache. [Accessed: 29-Nov-2021].

[26] W. Findlay, A. Somayaji, and D. Barrera, “Bpfbox: Simple precise process confinement with EBPF,” 09-Nov-2020. [Online]. Available:

https://www.cisl.carleton.ca/~will/written/conference/bpfbox-ccsw2020.pdf. [Accessed: 29-Nov-2021].

[27] Softprops, “Softprops/shiplift: rust interface for maneuvering Docker containers,” shiplift. 01-Sept-2021 [Online]. Available: https://github.com/softprops/shiplift. [Accessed: 29-Nov-2021].

Appendix A - Tools

bpftrace
https://bpftrace.org/
Bpftrace is a high-level tracing language built on eBPF which makes it easy to trace system calls, kprobes, uprobes and many more. Bpftrace is used within the project for experimenting with different eBPF probes.

lsns
https://www.man7.org/linux/man-pages/man8/lsns.8.html
Lists information about all existing namespaces.

df
https://www.man7.org/linux/man-pages/man1/df.1.html
Shows information about all currently mounted file systems.

objdump
https://www.man7.org/linux/man-pages/man1/objdump.1.html
Displays information about object files. In this project objdump is used to lookup function symbol names in different Docker binaries.

busybox container
https://hub.docker.com/_/busybox/
A lightweight Docker container. Throughout this project we will use the busybox Docker container for testing.

Appendix B - Relevant Code Repositories

BPFContain
https://github.com/willfindlay/bpfcontain-rs

BPFContain Docker Integration
https://github.com/trentholmes/bpfcontain-rs

dockerd
https://github.com/docker/docker-ce
https://github.com/moby/moby

containerd
https://github.com/containerd/containerd

runc
https://github.com/opencontainers/runc

libbpf
https://github.com/libbpf/libbpf

libbpf-rs
https://github.com/libbpf/libbpf-rs

bpftrace
https://github.com/iovisor/bpftrace/

Appendix C - Performance Benchmarking

Docker Container Creation Performance

Source code

Run using python3 (requires “pip install docker”)

Container id “4024c76d045b” comes from building the Docker container seen below.

```
import docker
import datetime

client = docker.from_env()
creation_start_time = datetime.datetime.now()
containers = []

for i in range(numberOfContainers):
    c = client.containers.run("4024c76d045b", detach=True)
    containers.append(c)

for c in containers:
    while(True):
        if(client.containers.get(c.id).status == "running"):
            break

creation_end_time = datetime.datetime.now()
print("Creation time for " + str(numberOfContainers) + " container(s) = " +
str(creation_end_time - creation_start_time))

for c in containers:
    c.stop(timeout=0)
    c.remove()
```

Docker Container used

Uses the lightway busybox container. It stays alive by sleeping for 30 minutes.

Build the following Dockerfile with “docker build .”

```
FROM busybox
CMD ["sleep", "1800"]
```

Runtime Performance

BPFContain Policy Used

```
name: benchmarking

allow:
  - dev: terminal
  - dev: random
  - dev: null
  - capability: any
  - net: [server, client, send, recv]
```

Docker container used for osbench

Build the following Dockerfile with “docker build .”

```
FROM phoronix/pts:latest
RUN apt-get update
RUN /phoronix-test-suite/phoronix-test-suite install osbench
CMD ["phoronix-test-suite/phoronix-test-suite", "strict-benchmark", "osbench"]
```

Docker container used for apache

Build the following Dockerfile with “docker build .”

```
FROM phoronix/pts:latest

RUN apt-get update
RUN apt-get install wget
RUN wget https://go.dev/dl/go1.17.3.linux-amd64.tar.gz
RUN tar -C /usr/local/ -xzf go1.17.3.linux-amd64.tar.gz
ENV PATH="/usr/local/go/bin:${PATH}"
RUN go get -u github.com/codesenberg/bombardier
RUN /phoronix-test-suite/phoronix-test-suite install apache

CMD ["phoronix-test-suite/phoronix-test-suite", "strict-benchmark", "apache"]
```