BPFContain: Towards Secure and Usable Containers with eBPF

COMP5900I Preliminary Work

William Findlay

December 14, 2020

Abstract

Redo this abstract when I have time Containers are becoming an increasingly important part of the Linux ecosystem. Containerized package managers like Snapcraft [37] and Flat-Pak [14] enable easy distribution and dependency management for desktop applications, while Docker [10] and Kubernetes [19] provide a framework for scaling and composing micro-services, especially in the cloud. While containers offer a convenient abstraction for distributing and configuring software, they are also often used as a light-weight alternative to heavier virtualization techniques, such as virtual machines. Thus, containers can also be thought of as security mechanisms, implementing a form of isolation between processes that share the resources of the underlying operating system.

Despite this clear security use case, existing container implementations do not consider security as a primary goal, and often fall back to insecure defaults when the host does not support the correct security abstractions. Further, container security implementations are often complex, relying on a myriad of virtualization techniques and security abstractions provided by the host operating system to isolate processes and enforce least-privilege. These security abstractions often paradoxically require elevated permissions to use in the first place, resulting in additional security risks when applications are able to escape confinement.

To rectify these container security issues, I present BPFContain[1], a novel approach to containers under the Linux kernel. BPFContain is built from the ground up as a light-weight yet secure process confinement solution for modern applications. Implemented in eBPF, an emerging technology for safely extending the Linux kernel, BPFContain enforces least privilege in containerized applications without requiring any additional privileges from the host operating system. Policies are written in a high-level language that is designed to be readable and modifiable by end-users without requiring significant security expertise. In this paper, I describe BPFContain's design and implementation, evaluate its performance and security, and discuss how it compares with existing container solutions.

[1] BPFContain is a working title and is subject to change in the future.

1 Introduction

Write this, or possibly graft over the abstract and write a new abstract.

2 Background

2.1 The Process Confinement Problem

The *process confinement problem*, also known as the *sandboxing problem*, refers to the goal of isolating a process or group of processes from the rest of the running system. In practice, this is often achieved by restricting an application's possible behaviour to its desired functionality, explicitly targeting its access to security-sensitive system resources such as files, network interfaces, and other running applications. Despite decades of work following Lampson's [20] first proposal of the process confinement problem in 1973, process confinement remains a somewhat open problem to date [8].

Discuss OS facilities at a high level, the reference monitor abstraction for access control

2.2 Low-Level Isolation Techniques

The Linux kernel supports various lower-level abstractions for implementing virtualization and enforcing least-privilege. While many of these mechanisms are insufficient for a full confinement implementation on their own, they are typically used in *combination* by higher-level techniques such as containers (c.f. Section 2.3) to achieve confinement. This section covers these low-level abstractions in detail.

Go over each of the following subsections, since they are mostly unchanged from the literature review

Unix Discretionary Access Control Discretionary access control (DAC) forms the most basic access control mechanism in many operating systems, including popular commodity operating systems such as Linux, macOS, and Windows. First formalized in the 1983 Department of Defense standard [43], a DAC system partitions system objects (e.g. files) by their respective owners and allows resource owners to grant access to other users, at their discretion. Typically, systems implementing discretionary access control also provide an exceptional user or role with the power to override discretionary access controls, such as the superuser (i.e. `root`) in Unix-like operating systems and the Administrator role in Windows.

While discretionary access controls themselves are insufficient to implement proper process confinement, they form the basis for the bare minimum level of protection available on many operating systems; therefore, they are an important part of the process confinement discussion. In many cases, user-centric discretionary access controls are abused to create per-application "users" and

"groups". For instance, a common pattern in Unix-like systems such as Linux, macOS, FreeBSD, and OpenBSD is to have specific users reserved for security-sensitive applications such as network-facing daemons. The Android mobile operating system takes this one step further, instead assigning an application- or developer-specific UID (user ID) and GID (group ID) to *each* application installed on the device [16].

In theory, these abuses of the DAC model would help mitigate the potential damage that a compromised application can do to the resources that belong to other users and applications on the system. However, due to DAC's discretionary nature, nothing prevents a given user from granting permissions to all other users on the system, unless other measures are put in place. Further, the inclusion of non-human users into a user-centric permission model may result in a disparity between an end-user's expectations and the reality of what a "user" is. This gap in understanding could result in further usability and security concerns.

POSIX Capabilities Related to discretionary access control are POSIX capabilities [4, 5, 24], which can be used to grant additional privileges to specific processes, overriding existing discretionary permissions. Further, a privileged process may *drop* specific capabilities that it no longer needs, retaining those it needs. Consequently, POSIX capabilities provide a finer-grained alternative to the all-or-nothing superuser privileges required by certain applications. For instance, a web-facing process that requires access to privileged ports has no business overriding file permissions. POSIX capabilities provide an interface for making such distinctions. Despite these benefits, POSIX capabilities have been criticized for adding additional complexity to an increasingly complex Linux permission model [4, 5]. Further, POSIX capabilities do nothing to confine processes beyond the original DAC model—rather, they help to solve the problem of overprivileged processes by limiting the privileges that they require in the first place.

Namespaces and Cgroups In Linux, *namespaces* and *cgroups* (short for control groups) allow for further confinement of processes by restricting the system resources that a process or group of processes is allowed to access. Namespaces isolate access by providing a process group a private, virtualized naming of a class of resources, such as process IDs, filesystem mountpoints, and user IDs. As of version 5.6, Linux supports eight distinct namespaces, depicted in Table 2.1. Complementary to namespaces, cgroups limit the use of *quantities* of system resources, such as CPU, memory, and block device I/O. Namespaces and cgroups provide fine granularity for limiting a process's view of available system resources. In this sense, they are better classified as a mechanism for implementing virtualization rather than least-privilege. They thus must be combined with other measures to constitute a full confinement implementation.

System Call Interposition **This will be adapted from my literature review**

Table 2.1: Linux namespaces (as of kernel version 5.6) and what they can be used to isolate.

Namespace	Isolates
PID	Process IDs (PIDs)
Mount	Filesystem mountpoints
Network	Networking stack
UTS	Host and domain names
IPC	Inter-process communication mechanisms
User	User IDs (UIDs) and group IDs (GIDs)
Time	System time
Cgroup	Visibility of cgroup membership

Linux Security Modules The Linux Security Modules (LSM) API [44] provides an extensible security framework for the Linux kernel, allowing for the implementation of powerful kernelspace security mechanisms that can be chained together. LSM works by integrating a series of strategically placed *security hooks* into kernelspace code. These hooks roughly correspond with boundaries for the modification of kernel objects. Multiple security implementations can hook into these LSM hooks and provide callbacks that generate audit logs and make policy decisions. Figure 2.1 depicts the LSM architecture in detail.

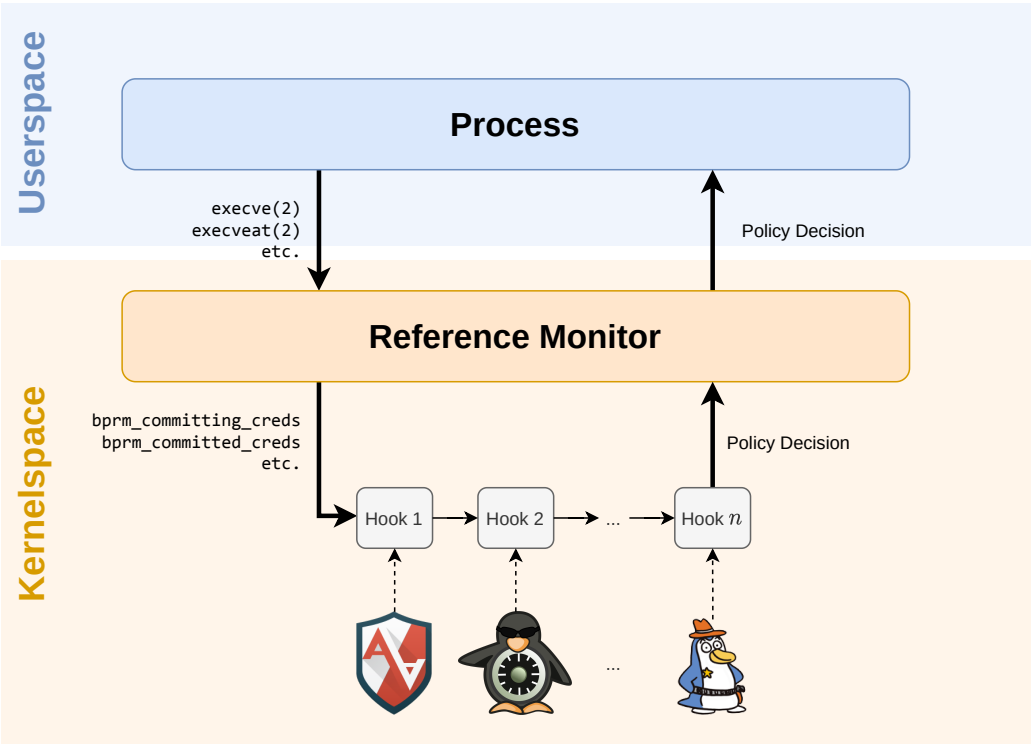

Figure 2.1: The LSM architecture. Note the many-to-many relation between access requests and hook invocations. Multiple LSM hooks may be chained together, incorporating policy from many security mechanisms. All hooks must agree to allow the access or it will be denied.

The LSM API sits at a level of abstraction just above the system call API—a single LSM hook

may cover multiple system calls and a single system call may contain multiple such LSM hooks. For instance, the `execve(2)` and `execveat(2)` calls both result in a call to the `bprm_committing_creds` and `bprm_committed_creds` hooks (among others). This provides a nice level of abstraction compared to system-call-based approaches like seccomp-bpf [11, 26] in that a single LSM hook can cover all closely related security events (recall the issue of `open(2)` vs `openat(2)` in Seccomp-bpf).

The Linux kernel contains several in-tree LSM-based security modules, which may be enabled by default on certain distributions. Many such modules implement *mandatory access control* (MAC) schemes, which enable fine-grained access control that can limit the privileges of *all users*—even the superuser. SELinux [35] and AppArmor [7] are two such MAC LSMs, each with its own policy semantics. I discuss each in turn.

SELinux [35] was originally developed by the NSA as a Linux implementation of the Flask [39] security model. Under SELinux, system subjects (users, processes, etc.) and system objects (files, network sockets, etc.) are assigned corresponding labels. Security policy is then written based on these labels, specifying the allowed access patterns between a particular object type and subject type. SELinux's policy language is famously arcane [33]. Despite multiple efforts to introduce automated policy generation [28, 36, 38], writing and auditing SELinux security policy remains a task for security experts rather than end-users. Further, due to the difficulty of writing and auditing the complex SELinux policy language, there is a natural tendency for human policy authors to err on the side of over-permission, violating the principle of least privilege.

AppArmor (originally called SubDomain) [7] is often touted as a more usable alternative to SELinux, although usability studies have shown that this claim merits scrutiny [33]. Rather than basing security policy on labelling system subjects and objects, AppArmor instead employs path-based enforcement. AppArmor defines policy in per-application profiles, which contain rules specifying what system objects the application can access. System objects are identified directly (for example, via pathnames, socket classes, or IP network addresses) rather than labelled. AppArmor also supports the notion of *changing hats*, wherein a process may change its AppArmor profile under certain conditions specified in the policy. Although AppArmor profiles are more conforming to standard Unix semantics than their SELinux counterparts, users who wish to write AppArmor policy still require a considerable amount of knowledge about operating system security [33].

2.3 Containers

Containers use OS-level virtualization and confinement mechanisms (c.f. Section 2.2) to provide a (semi-)isolated environment for the execution of processes [42]. Since they run directly on the host operating system and share the underlying OS kernel, containers do not require a full guest operating system to implement virtualization. This technique has the advantage of offering a light-weight alternative to traditional hardware virtualization approaches using full virtual machines [42]. Compared with containers, traditional approaches to virtualization involve hypervisors, which virtualize and provide access to the underlying hardware, either running on top of a host operating

system or directly on top of the hardware itself. Full virtual machines run on top of these hypervisors, each running a guest operating system with a full userland and kernel. Full virtualization provides stronger isolation guarantees than containers but involves significantly more overhead imposed by the guest operating system [42]. Figure 2.2 depicts an overview of the architectural differences between containers and full hardware virtualization solutions (i.e. virtual machines running on top of hypervisors).

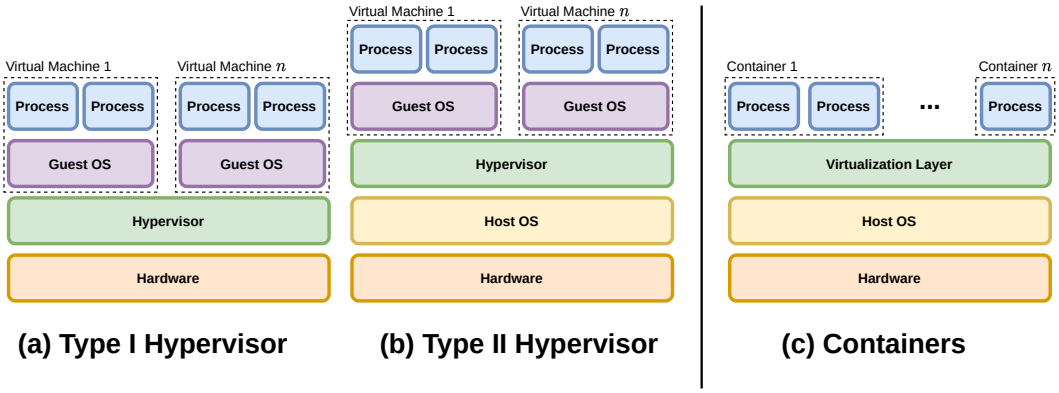

Figure 2.2: Virtual machine and container architectures. Type I hypervisors **(a)** virtualize and control the underlying hardware directly, but require full guest operating systems on top of the virtualization layer. Type II hypervisors **(b)** run on top of a host operating system but still require full guest operating systems above the virtualization layer. Containers **(c)** achieve virtualization using a thin layer provided by the host operating system itself. They share the underlying operating system kernel and resources, and therefore require no guest operating system [42].

Since containers must share the host operating system kernel and related resources, it is essential to consider how best to isolate them from one another. Therefore, a container management system generally seeks to achieve the following security-related goals[2]:

CG1. Virtualization. Virtualization aims to provide each container a *virtual view* of system resources. Containers generally achieve virtualization using a combination of Linux namespaces, cgroups, and filesystem mounts. Namespaces provide a private view of enumerable resources (i.e. a virtual mapping of IDs to resources). Such enumerable resources include process IDs, user and group IDs, mountpoints, and the network interfaces. Cgroups similarly provide a virtual view of *quantifiable resources* (i.e. how much of a given resource is available). Such quantifiable resources include the CPU, persistent storage, memory, and I/O bandwidth. Filesystem mounts, combined with mount namespaces, provide a virtual view of visible files.

Although full virtualization may be desirable in the ideal case, containers often only implement partial virtualization [22, 42] due to a variety of factors. Pragmatically, it is often

[2]Dependency management is another goal of container management systems like Docker **cite**, but it is out of scope for his paper.

beneficial for containers to have a shared view of specific system resources, depending on the use case. For instance, two containers might share a copy of the same shared library or require access to a shared IPC namespace to enable communication. In practice, containers often leverage layered filesystems such as overlayfs [12] to deduplicate files across containers and the host system. Partial virtualization can enable lighter-weight containers and easier communication between two containers to satisfy the goal of composability [42].

CG2. Least-Privilege. For a container to be considered secure, it must enforce least-privilege on its processes [42]. This requirement makes practical sense, given that a container runs directly on the host system and must share the underlying OS kernel and resources with both other containers and the host system itself. Without least-privilege, a process running in a container has virtually the same access rights as an unconfined process. When the container itself is running with privileged access to the system (as is often the case [22, 42]), this may even result in an *escalation of privilege* compared to the scenario where the process runs directly on the host. For these reasons, it is neither practical nor advisable to rely on weak virtualization guarantees to protect the host system with no means of enforcing least-privilege [42].

A least-privilege implementation for containers typically involves a combination of multiple enforcement mechanisms, including Unix DAC, seccomp-based system call filtering, dropped POSIX capabilities, and mandatory access control mechanisms (using one of the Linux MAC LSMs) [10, 22, 42]. This complexity can lead to usability and auditability concerns, as a simple policy language must compile down to multiple complex enforcement mechanisms that need to work cooperatively [13].

Despite its evident importance for container security, existing container management solutions generally treat least-privilege as a secondary goal [42]. Docker attempts to provide sensible security defaults for containers. Still, these defaults may be easily overridden and often rely on the presence of extra kernel security features such as the AppArmor LSM [10]. When AppArmor is not available, Docker falls back to relying exclusively on its default seccomp policy and dropped capabilities. Security defaults for containers also often do not adhere to the principle of least privilege. For instance, Docker provides containers with 15 Linux capabilities by default, including `CAP_DAC_OVERRIDE`, which allows a container to override all discretionary access control checks [10, 42].

CG3. Composability. Increasingly, containers are being used to implement composable microservices [42]. For instance, Kubernetes [19] allows the user to group containers into *pods*, which are then allowed to communicate with each other in pre-defined ways. For composability, a container needs to be able to communicate with another container without sacrificing virtualization or least-privilege. In practice, containers achieve such composability by defining specific inter-container exceptions to virtualization and least-privilege

policy [42]. Naturally, these exceptions can increase the risk of an insecure configuration and the user must carefully manage them to avoid overprivilege.

Discuss prominent examples: Docker, Kubernetes
Discuss containerized package management: Snap, FlatPak

2.4 Classic and Extended BPF

The original Berkeley Packet Filter (BPF) [29], hereafter referred to as Classic BPF, was a packet filtering mechanism implemented initially for BSD Unix. Classic BPF was created as a lightweight replacement for traditional packet filtering mechanisms, which relied on frequent context switches between userspace and kernelspace while making filtering decisions. Instead, Classic BPF implemented a simple register-based virtual machine language and efficient buffer data structures to minimize the required context switches. As an efficient packet filtering mechanism, Classic BPF quickly gained traction in the *NIX community and was subsequently ported to various open-source Unix and Unix-like operating systems, most notably Linux [23], OpenBSD [30], and FreeBSD [15].

The Linux kernel development community eventually realized that the BPF engine could be applied to more than just packet filtering. The 2012 introduction of seccomp-bpf [11, 26] enabled Classic BPF programs to be written and applied to make system call filtering decisions for userspace applications. This extension to seccomp transformed it into a powerful (yet notoriously difficult-to-use [1]) mechanism for making security decisions about system calls in a confined process.

In 2014, Starovoitov and Borkmann merged a complete rewrite of the Linux BPF engine, dubbed Extended BPF (eBPF), into the mainline kernel [40]. eBPF expands on the original BPF specification by introducing:

- An extended instruction set;
- 11 registers (10 of which are general-purpose);
- Access to allow-listed kernel helpers;
- Just-in-time (JIT) compilation to native instruction sets;
- A program safety verifier;
- A large collection of specialized data structures; and
- New program types which can be attached to a variety of system events in both userspace and kernelspace.

These extensions to the Classic BPF engine effectively turn eBPF into a general-purpose execution engine in the kernel with powerful system introspection and kernel extension capabilities. eBPF programs execute in the kernel with supervisor privileges but are limited by a restricted execution

context and pre-checked for safety by an in-kernel verification engine. In particular, eBPF programs are limited to a 512-byte stack, cannot access unbounded memory regions, must not have back-edges in their control flow, and must provably terminate [17]. As a consequence of these restrictions, eBPF programs are not Turing-complete. Where necessary, an eBPF program can make calls to a set of allow-listed kernel helpers to obtain additional functionality, such as access to external memory regions and various kernel facilities such as signalling or random number generation [17].

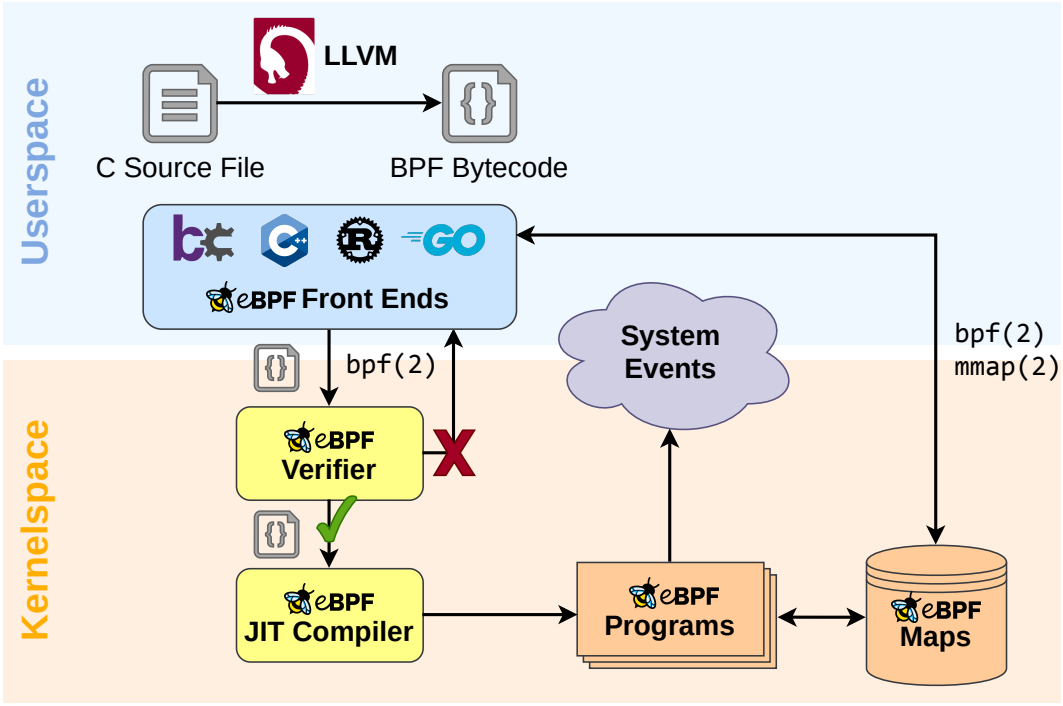

Figure 2.3: eBPF mechanisms in the kernel. Userspace front-ends compile C source code into eBPF bytecode using the LLVM toolchain and load it into the kernel with the `bpf(2)` system call. The in-kernel verifier either accepts or rejects the program based on its adherence to safety invariants. Accepted programs are attached to system events across the userspace and kernelspace boundary where they are just-in-time compiled to the native instruction set. Programs can store and fetch data using data structures called "eBPF maps" which can also be accessed directly from userspace.

A privileged userspace process may load an eBPF program into the kernel using Linux's `bpf(2)` system call. While it is possible to write eBPF bytecode by hand [17], several front-ends exist for compiling eBPF bytecode from a restricted subset of the C programming language[3], including bcc [18] and libbpf [21]. These front-ends typically use the LLVM [27] compiler toolchain to produce BPF bytecode. When the kernel receives a request to load an eBPF program, it first checks the bytecode to ensure that it conforms to the safety invariants outlined above. If the verifier accepts the program, it may then be attached to one or more system events. When an event fires, the eBPF program is executed via just-in-time compilation to the native instruction set. eBPF programs

[3]In principle, this language need not be C. For instance, a framework exists for writing eBPF programs in pure Rust [31]. However, C is a popular choice since it is tightly coupled with the underlying implementation of the kernel.

can store data in several specialized in-kernel data structures, which are also made accessible to userspace via the `bpf(2)` system call or a direct memory mapping. Figure 2.3 depicts this process in detail.

All paragraphs up to this point have been checked with Grammarly

3 Motivation

3.1 Threat Model

Sultan *et al.* [42] propose four broad categories of container-related threats. Figure 3.1 presents an overview of each category. An application running within a container might attack the container itself, attempting to escape confinement or interfere with the execution of co-located applications running within the same container. Inter-container threats are similarly possible, wherein one container attempts to interfere with or take over another. Since containers share the underlying host operating system, it is also possible for a container to directly attack the host, either by escaping confinement altogether or by launching denial of service or resource consumption attacks. Finally, a malicious or semi-honest host system may attack containers running within it. Researchers have generally recognized that mitigating this fourth category of attack requires the use of hardware security mechanisms [42] **cite others** such as trusted execution environments or trusted platform modules. Such host-to-container attacks are, therefore, out of scope for this paper.

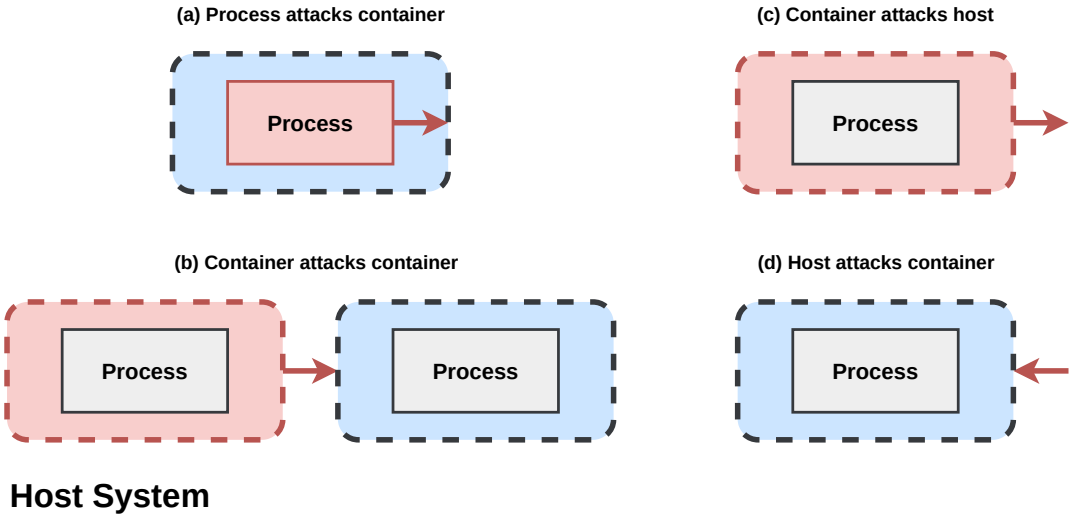

Figure 3.1: Four categories of container-related attacks [42]. **(a)** A process running within a container attacks the container itself. **(b)** One container attacks another container. **(c)** A container attacks the host system. **(d)** The host system attacks the container. This fourth category of attack is out of scope for this paper.

In this threat model we consider three broad classes of attack vector, comprised of the containers and the applications that run within them. These attack vectors are described in turn below.

AV1. Malicious Applications. A malicious application is designed with express malicious intent. Malicious software may actively attempt to subvert other applications, other containers, or the host system itself. This subversion could include privilege escalation attacks on the host system, denial of service attacks on other containers or the host, or the installation of backdoors. A sophisticated attacker could even abuse a malicious application to install a rootkit [2] in the container, on the host system, or in the host's firmware, stealthily gaining permanent and possibly overprivileged access.

AV2. Semi-Honest Applications. In contrast with malicious applications, semi-honest applications are not necessarily expressly designed with malicious intent and might even be cooperative with other applications or containers. However, the semi-honest application can passively participate in unwanted activity, such as surveillance or consumption of the host's resources.

AV3. Vulnerable Applications. Vulnerable applications running inside containers may become compromised by attackers, often with the goal of using these applications to orchestrate a more sophisticated attack on other applications running within the same container, other containers, or the host system itself [42]. Common vulnerabilities here include code execution vulnerabilities on untrusted input, memory corruption bugs, and privilege escalation vulnerabilities in a container's configuration. Exploitation of kernel-level vulnerabilities are also a concern here, as an attacker can potentially abuse a legitimate application to target vulnerable code paths in the kernel [22].

Since containers are often co-located in large-scale, multi-tenant systems such as the cloud [42], the potential for exploitation or abuse by foreign threat actors is exacerbated. Thus, it is imperative that container management systems enforce least-privilege on containers to prevent such exploitation from negatively impacting the rest of the system.

3.2 The Quest for Secure Containers

3.3 Why an eBPF Implementation?

- **Can be dynamically loaded without rebooting the kernel**
- **Guaranteed safety**
- **Can be used in production without downtime**
- **Container-specific LSM → Only apply policy to a given container**
- **Dynamically instrument non-lsm functions in the kernel, such as `commit_creds`, this allows us to prevent kernel privilege escalation attacks like the ones described in Xin *et al.* [22]**

4 BPFContain Design and Implementation

Five specific goals informed the design of BPFContain's policy language and enforcement mechanism, enumerated below as Design Goals D1 to D5.

D1. Usability. BPFContain's basic functionality should not impose unnecessary usability barriers on end-users. Its policy language should be easy to understand and semantically meaningful to users without significant security knowledge. To accomplish this goal, BPFContain takes some inspiration from other high-level policy languages for containerized applications, such as those used in Snapcraft [37].

D2. Configurability. It should be easy for an end-user to reconfigure policy to match their specific use case, without worrying about the underlying details of the operating system or the policy enforcement mechanism. It should be possible to use BPFContain to restrict specific unwanted behaviour in a given application without needing to write a rigorous security policy from scratch.

D3. Transparency. Containing an application using BPFContain should not require modifying the application's source code or running the application using a privileged SUID (Set User ID root) binary. BPFContain should be entirely agnostic to the rest of the system and should not interfere with its regular use.

D4. Adoptability. BPFContain should be adoptable across a wide variety of system configurations and should not negatively impact the running system. It should be possible to deploy BPFContain in a production environment without impacting system stability and robustness or exposing the system to new security vulnerabilities. BPFContain relies on the underlying properties of its eBPF implementation to achieve its adoptability guarantees.

D5. Security. BPFContain should be built from the ground up with security in mind. In particular, security should not be an opt-in feature and BPFContain should adhere to the principle of least privilege [32] by default. It should be easy to tune a BPFContain policy to respond to new threats.

4.1 BPFContain Policy

BPFContain policy consists of simple manifests written in YAML [3], a human-readable data serialization language based on key-value pairs. Each BPFContain container is associated with a manifest, which itself consists of a few lines of metadata followed by a set of *rights* and *restrictions*. A *right* specifies access that should be granted to a container, while a *restriction* is used revoke access. While rights and restrictions may be combined at various levels of granularity, a restriction *always* overrides a right, without exception. In practice, this allows the construction of nuanced

policies that specify coarse-grained access with finer-grained exceptions. Table 4.1 describes the various access labels that can be used in BPFContain policy.

Table 4.1: A list of resources which can be confined by BPFContain policy, along with their parameters and descriptions. Square brackets denote an optional parameter.

Resource	Parameters	Description
`filesystem`	Mountpoint, [Access]	Grants or revokes access at the granularity of a filesystem mountpoint. Access defaults to {read, write, append, chdir, setattr}, unless otherwise specified. An optional `readonly` flag grants {read, chdir} access.
`file`	Pathname, Access	Grants or revokes access at the granularity of individual files. Access must be specified.
`capability`	Capability	Grants or revokes access to the specified POSIX capability **cite**. Note that this cannot be used to grant additional capabilities—it merely enforces on capabilities already granted to the process.
`network`	[Interface], [Access]	Grants or revokes access to network communications. A specific interface and access pattern may optionally be specified.
`ipc`	Container	Grants or revokes access to communicate with processes in *another* BPFContain container. This covers all supported System V IPC categories as well as signals.
`tty`	[Access]	Grants or revokes access to `/dev/tty*` devices.
`pts`	[Access]	Grants or revokes access to `/dev/pts/*` devices.
`video`	[Access]	Grants or revokes access to `/dev/pts/video*` devices.
`sound`	[Access]	Grants or revokes access to `/dev/pts/snd*` devices.
		...

Following the principle of least privilege [32], BPFContain implements strict default-deny enforcement, only granting access that the policy specifically declares under the container's set of rights. The user may optionally change this behaviour and elect to enforce a default-allow policy instead, by setting `default: allow` in the manifest. A default-allow policy enables the easy restriction of specific unwanted behaviour in a given program, without worrying about the details of constructing a rigorous security policy.

Rework this, since the threat model section is now new As a motivating example of BPFContain security policy, consider the Discord client, discussed briefly in **??**. Discord is a popular cross-platform voice chat client designed for gamers and comes with an optional feature, "Display Active Game", which displays whatever game the user is currently playing in their status

Table 4.2: Implicit policy in BPFContain, which is enforced regardless of a container's manifest. Implicit restrictions generally correspond with resources which a well-behaved container should never need. Such accesses are typically recognized by the community as dangerous or enabling a container to escape confinement **cite**. Implicit rights permit certain sane behaviours, such as interprocess communication between processes *within* a container. These rights effectively constitute exceptions to ordinary enforcement. Note that implicit rights may still be overridden by an explicit restriction specified in the container's manifest. A third category, implicit death, refers to accesses that cause BPFContain to send an uncatchable SIGKILL to the process.

Policy	Kind	Description
BPF	Restriction	A container is disallowed from making *any* `bpf(2)` system calls. This restriction prevents a container from loading, unloading, and accessing any eBPF programs and maps, including those which belong to BPFContain itself.
Ptrace	Restriction	A container may not use `ptrace(2)` to trace or control processes.
Kernel Lockdown	Restriction	A container is subject to full Kernel Lockdown [25] restrictions, disabling all operations that could be used for arbitrary code execution in the kernel.
Kernel Modules	Restriction	A container may not load any modules into the kernel.
Kexec	Restriction	A container may not use `kexec`-family system calls to load new kernels.
Shutdown	Restriction	A container may not shut down or reboot the system.
Key Management	Restriction	A container may not interface with the kernel's key management mechanisms.
Quotactl	Restriction	A container cannot use the `quotactl(2)` system call to bypass restrictions on resource consumption.
Rlimit	Restriction	A container cannot use the `getrlimit(2)`, `setrlimit(2)`, or `prlimit(2)` system calls to get or set resource limits.
Scheduler	Restriction	A container cannot inspect or modify process scheduling or I/O scheduling priority.
Mount	Restriction	A container cannot mount, remount, unmount filesystems or move filesystem mounts.
Pivot Root	Restriction	A container cannot pivot the root directory of a filesystem.
Syslog	Restriction	A container cannot use the `syslog(2)` system call to access the kernel logs.
Set Time	Restriction	A container cannot use the `settime(2)` system call to change the system time.
Kernel Privilege Escalation	Death	To prevent kernel privilege exploits which typically rely on forcing the execution of `commit_creds` [22], BPFContain will outright kill a contained process that invokes this function to try to escalate its privileges.
IPC	Right	A process can always perform interprocess communication with another process within the same container.
Procfs	Right	A process is granted full access to its own procfs entries, as well as those belonging to other processes within the same container.
New Files	Right	A container is granted full access to any new files or directories that it creates.

message. To accomplish this, the Linux Discord client periodically scans the `procfs` filesystem to obtain a list of all running processes. While this feature may seem innocuous at first glance, an `strace` [41] of Discord reveals that it continually scans the process tree even when the "Display Active Game" feature is *disabled*. This behaviour represents a gross violation of the user's privacy expectations. To rectify this issue, a user might write a BPFContain policy like the examples depicted in Listing 4.1 and Listing 4.2.

Listing 4.1: A sample manifest for Discord [9] using BPFContain's more restrictive default-deny confinement. All accesses which are not listed under the container's rights are implictly denied. The explicit restriction on access to `procfs` prevents Discord from scanning the process tree, regardless of its rights.

```
name: discord
command: /bin/discord
rights:
  - filesystem /
  - network
  - video
  - sound
restrictions:
  - filesystem /proc
```

Listing 4.2: A sample manifest for Discord [9] using BPFContain's optional default-allow confinement. This permits a much simpler policy that directly targets Discord's `procfs` scanning behaviour.

```
name: discord
command: /bin/discord
default: allow
restrictions:
  - filesystem /proc
```

In the first example (Listing 4.1), the container grants access to the root filesystem, networking capabilities, and video and sound devices. It explicitly restricts access to the `procfs` filesystem, preventing Discord from scanning the process tree. In the second example (Listing 4.2), a more permissive policy is defined which serves *only* to restrict access to `procfs`. The choice of which alternative to use is left entirely up to the user, and may depend on various factors such as the existence of a pre-configured policy file, the desired use case, and the user's level of comfort with BPFContain's policy semantics.

To run a BPFContain container, the user invokes `bpfcontain run <name>` where `name` is the unique container name declared in the manifest. The `bpfcontain run` command is a thin wrapper around that target application, whose only purpose is to invoke a special library call, `bpfcontain_confine`, that marks the process group for confinement before executing the command(s) defined in the manifest.

An important feature of BPFContain is that the `bpfcontain_confine` library call requires no

additional operating system privileges to start confinement. This notion of unprivileged confinement is a unique advantage over other container implementations in Linux. Somewhat counter-intuitively, traditional container implementations often rely on binaries with escalated privileges (e.g. setuid root) to set up confinement. Failure to correctly drop these elevated privileges may result in *escalation of privilege* in the host system, particularly if the confined process manages to escape the container. By obviating this need for elevated privileges, BPFContain conforms with the principle of least privilege and improves Linux containers' overall security.

As a side effect of BPFContain's design, it is also possible for a generic application to invoke the `bpfcontain_confine` library call directly, eliminating the need to start the target application using the `bpfcontain run` wrapper. This notion of self-confinement enables application developers and package maintainers to ship BPFContain policy with their software and enforce it transparently to the end-user. Since BPFContain policy is designed to be readable and modifiable by end-users, a security policy shipped with an application could optionally be tuned by the user according to their specific needs.

4.2 Architecture

BPFContain consists of both userspace and kernelspace components, which interact co-operatively to implement the containerization and policy enforcement mechanisms. Roughly, its architecture (depicted in Figure 4.1) can be broken down into the following four components:

C1. A privileged daemon, responsible for loading and managing the lifecycle of eBPF programs and maps, as well as logging security events to userspace.

C2. A small shared library and unprivileged wrapper application used to initiate confinement.

C3. A set of eBPF programs, running in kernelspace. These programs are attached to LSM hooks in the kernel as well as the shared library in C2.

C4. A set of eBPF maps, special data structures which allow bidirectional communication between userspace and kernelspace. These maps are used to track the state of running containers and to store the active security policy for each container.

In userspace, BPFContain runs in the background as a privileged daemon. The daemon is responsible for loading BPFContain's eBPF programs and maps and logging security events to userspace, such as policy violations. When it first starts, the daemon invokes a series of `bpf(2)` system calls to load its eBPF programs and maps into the kernel. After loading all eBPF programs and maps, the daemon then parses, translates, and loads each per-container policy file into specialized policy maps.

To allow processes to request that they be placed into a container, BPFContain attaches a specialized eBPF program type called a `uprobe` (userspace probe) to a userspace library call,

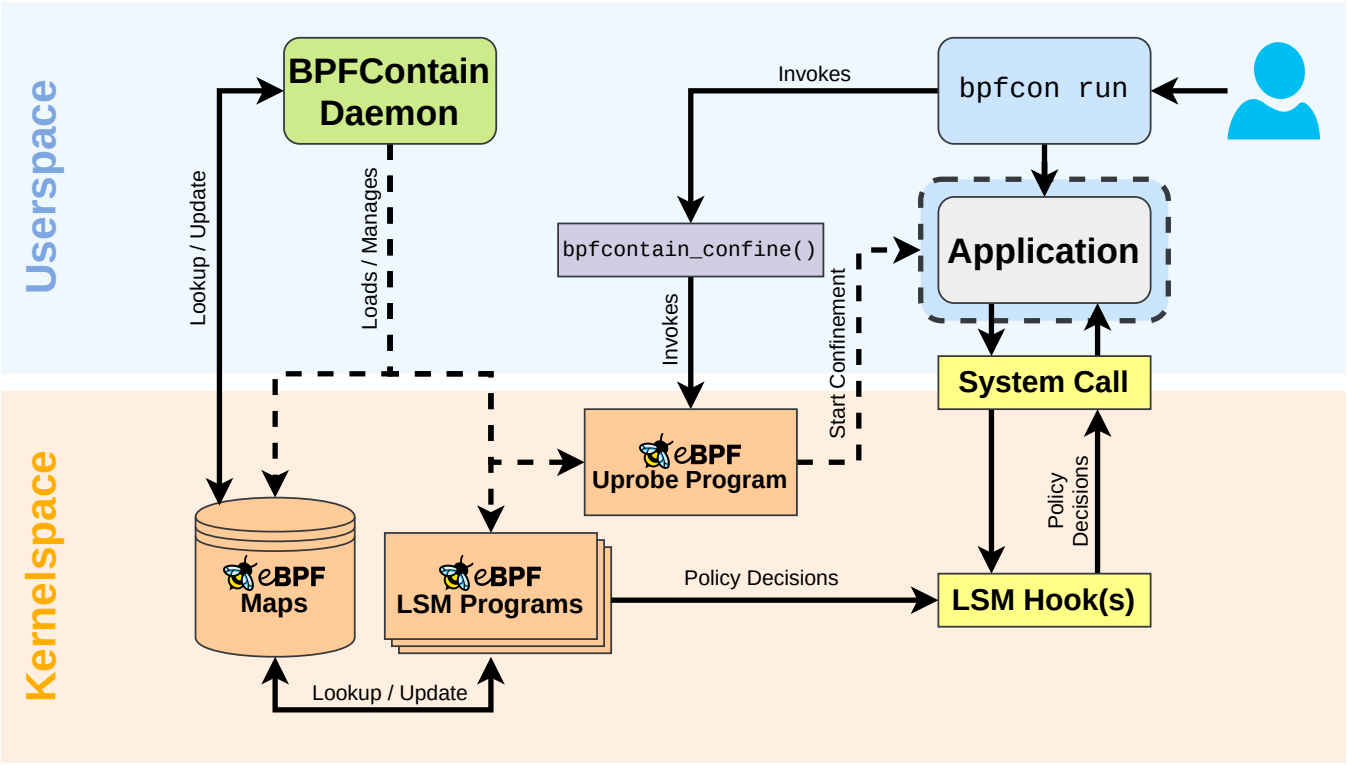

Figure 4.1: A diagram of BPFContain's architecture. The privileged daemon (green) is responsible for loading the necessary eBPF maps and programs (orange) into the kernel and managing their lifecycle. The user starts a container by executing an unprivileged wrapper application (blue), which invokes the `bpfcontain_confine` library call (purple), trapping to a special eBPF program that associates the process group with the correct policy. When the confined application (grey) makes a system call to request access to a sensitive resource, the kernel invokes one or more LSM hooks (yellow) which in turn trap to corresponding eBPF LSM programs that make the correct policy decision.

`bpfcontain_confine`. This function is a stub, whose only purpose is to trap to the uprobe—if this function fails to trap the corresponding eBPF program (for example if BPFContain has not yet loaded its eBPF programs into the kernel), this function returns `-EAGAIN` to indicate that the caller should repeat the request. Attaching a `uprobe` to a library call in this way is a common eBPF design pattern, which effectively allows eBPF programs to make almost arbitrary extensions to the kernel's API.

4.3 Enforcing Policy

BPFContain enforces policy using eBPF programs attached to LSM hooks, a feature introduced in Linux 5.7 by KP Singh's KRSI (Kernel Runtime Security Instrumentation) patch [6, 34]. KRSI enables the attachment of multiple eBPF programs to a given LSM hook, which work co-operatively with each other and other Linux security modules to come to a policy decision, with any one denial resulting in a denial for the given operation.

BPFContain only enforces security policy on processes which are currently associated with a container. The list of processes associated with a container is tracked using an eBPF map, which is updated whenever a process invokes the `bpfcontain_confine` library call (assuming it is not already in a container), and whenever a process that is currently in a container forks itself. Once a process has been associated with a container, it remains associated with that container until it terminates.

Security policy in BPFContain falls into two categories: *implicit* and *explicit*. Implicit policy is the set of sensible defaults that are defined to allow interaction *within* a given container. For instance, all processes in the same container may communicate with each other using various interprocess communication mechanisms. Explicit policy, on the other hand, refers to the rights and restrictions which have been explicitly defined in a container's manifest. Unless a container has been marked as `default: allow`, all access requests which are not covered under the implicit or explicit policies for a container are denied by default, and the access request is logged to userspace by the BPFContain daemon.

BPFContain policy is stored in kernelspace using several eBPF maps, one for each policy category. These maps are keyed using a composite key comprised of a unique ID associated with each container combined with another unique identifier for the given resource. For instance, filesystem policy is keyed using the container's ID and the unique identifier associated with the mounted device. Each key in a policy map is associated with a vector describing the allowed access, depending on the granularity of the rule and its associated parameters.

As instrumented LSM hooks are invoked, BPFContain queries the map of active processes to determine which container the process is associated with, if any. The corresponding policy map is then queried using the appropriate key, derived from the container ID associated with the currently running process and the unique identifiers corresponding to the requested resource. If no matching entry is found, access is denied (assuming that the policy has not been marked default allow). Otherwise, the requested access is compared with the value found in the map, and access is only

granted if the values match.

5 Evaluation

Write this

6 Discussion

Write this

7 Related Work

This will be adapted from my literature review.

8 Conclusion

Write this

9 Acknowledgements

The idea for BPFContain was conceived during a discussion with Anil Somayaji.

References

[1] James P. Anderson, "A Comparison of Unix Sandboxing Techniques," *FreeBSD Journal*, 2017. [Online]. Available: http://www.engr.mun.ca/~anderson/publications/2017/sandbox-comparison.pdf.

[2] Lynn Erla Beegle, "Rootkits and Their Effects on Information Security," *Information Systems Security*, vol. 16, no. 3, pp. 164–176, 2007. DOI: 10.1080/10658980701402049.

[3] Oren Ben-Kiki, Clark Evans, and Ingy döt Net, *YAML Ain't Markup Language (YAML™) Version 1.2*, YAML specification. [Online]. Available: https://yaml.org/spec/1.2/spec.html (visited on 11/29/2020).

[4] Jonathan Corbet, "A Bid to Resurrect Linux Capabilities," *LWN.net*, 2006. [Online]. Available: https://lwn.net/Articles/199004/.

[5] Jonathan Corbet, "File-Based Capabilities," *LWN.net*, 2006. [Online]. Available: https://lwn.net/Articles/211883/.

[6] Jonathan Corbet, "KRSI — the other BPF security module," *LWN.net*, Dec. 2019. [Online]. Available: https://lwn.net/Articles/808048/.

[7] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor, "SubDomain: Parsimonious Server Security," in *Proceedings of the 14st Large Installation Systems Administration Conference (LISA)*, New Orleans, LA, United States: USENIX Association, 2000. [Online]. Available: https://www.usenix.org/legacy/event/lisa2000/full_papers/cowan/cowan.pdf.

[8] Alexander Crowell, Beng Heng Ng, Earlence Fernandes, and Atul Prakash, "The Confinement Problem: 40 Years Later," *Journal of Information Processing Systems*, vol. 9, no. 2, pp. 189–204, 2013. DOI: 10.3745/JIPS.2013.9.2.189.

[9] Discord, *Discord Privacy Policy*. [Online]. Available: https://discord.com/privacy (visited on 10/25/2020).

[10] Docker, *Docker Security*, 2020. [Online]. Available: https://docs.docker.com/engine/security/security (visited on 10/25/2020).

[11] Will Drewry, "Dynamic seccomp policies (using BPF filters)," Kernel patch, 2012. [Online]. Available: https://lwn.net/Articles/475019/.

[12] Jake Edge, "Another Union Filesystem Approach," *LWN.net*, 2010. [Online]. Available: https://lwn.net/Articles/403012/ (visited on 12/13/2020).

[13] William Findlay, Anil Somayaji, and David Barrera, "bpfbox: Simple Precise Process Confinement with eBPF," in *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, ser. CCSW'20, Virtual Event, USA: Association for Computing Machinery, 2020, pp. 91–103. DOI: 10.1145/3411495.3421358.

[14] Flatpak, *Sandbox Permissions*, 2020. [Online]. Available: https://docs.flatpak.org/en/latest/sandbox-permissions.html (visited on 10/25/2020).

[15] FreeBSD, *bpf(4)*, BSD Kernel Interfaces Manual. [Online]. Available: https://www.unix.com/man-page/FreeBSD/4/bpf (visited on 12/13/2020).

[16] Google, *Android Security Features*, Android security documentation. [Online]. Available: https://source.android.com/security/features (visited on 10/26/2020).

[17] Brendan Gregg, *BPF Performance Tools*. Addison-Wesley Professional, 2019, ISBN: 0-13-655482-2.

[18] IOVisor, *iovisor/bcc*, GitHub repository. [Online]. Available: https://github.com/iovisor/bcc (visited on 12/13/2020).

[19] Kubernetes, *Kubernetes*, 2020. [Online]. Available: https://kubernetes.io (visited on 11/30/2020).

[20] Butler W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM*, vol. 16, no. 10, pp. 613–615, 1973, ISSN: 0001-0782. DOI: 10.1145/362375.362389.

[21] libbpf contributors, *libbpf*, GitHub repository. [Online]. Available: https://github.com/libbpf/libbpf (visited on 12/13/2020).

[22] Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou, "A Measurement Study on Linux Container Security: Attacks and Countermeasures," in *Proceedings of the 34th Annual Computer Security Applications Conference*, ser. ACSAC '18, San Juan, PR, USA: Association for Computing Machinery, 2018, pp. 418–429, ISBN: 9781450365697. DOI: 10.1145/3274694.3274720.

[23] Linux, *bpf(2)*, Linux Programmer's Manual. [Online]. Available: https://www.man7.org/linux/man-pages/man2/bpf.2.html (visited on 12/13/2020).

[24] Linux, *capabilities(7)*, Linux User's Manual. [Online]. Available: https://linux.die.net/man/7/capabilities.

[25] Linux, *kernel_lockdown(7)*, Linux programmer's manual. [Online]. Available: https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html (visited on 12/13/2020).

[26] Linux, *Seccomp BPF (SECure COMPuting with filters)*, Linux kernel documentation. [Online]. Available: https://static.lwn.net/kerneldoc/userspace-api/seccomp_filter.html (visited on 10/27/2020).

[27] LLVM, *BPF Directory Reference*, Developer documenation. [Online]. Available: https://llvm.org/doxygen/dir_b9f4b12c13768d2acd91c9fc79be9cbf.html (visited on 12/13/2020).

[28] Karl MacMillan, "Madison: A new approach to policy generation," in *SELinux Symposium*, 2007. [Online]. Available: http://selinuxsymposium.org/2007/papers/08-polgen.pdf.

[29] Steven McCanne and Van Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," *USENIX Winter*, vol. 93, 1993. [Online]. Available: https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[30] OpenBSD, *bpf(4)*, Device Drivers Manual. [Online]. Available: https://man.openbsd.org/bpf (visited on 12/13/2020).

[31] RedSift, *redsift/redbpf*, GitHub repository. [Online]. Available: https://github.com/redsift/redbpf (visited on 12/13/2020).

[32] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975. DOI: 10.1109/PROC.1975.9939.

[33] Z. Cliffe Schreuders, Tanya Jane McGill, and Christian Payne, "Towards Usable Application-Oriented Access Controls," in *International Journal of Information Security and Privacy*, vol. 6, 2012, pp. 57–76. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.963.860&rep=rep1&type=pdf.

[34] KP Singh, "MAC and Audit policy using eBPF (KRSI)," Kernel patch, 2019. [Online]. Available: https://lwn.net/ml/linux-kernel/20191220154208.15895-1-kpsingh@chromium.org/.

[35] Stephen Smalley, Chris Vance, and Wayne Salamon, "Implementing SELinux as a Linux security module," 43, vol. 1, 2001, p. 139. [Online]. Available: https://www.cs.unibo.it/~sacerdot/doc/so/slm/selinux-module.pdf.

[36] Justin R. Smith, Yuichi Nakamura, and Dan Walsh, *audit2allow(1)*, Linux user's manual. [Online]. Available: http://linux.die.net/man/1/audit2allow.

[37] Snapcraft, *Security Policy and Sandboxing*, 2020. [Online]. Available: https://snapcraft.io/docs/security-sandboxing (visited on 10/25/2020).

[38] Brian T. Sniffen, David R. Harris, and John D. Ramsdell, "Guided policy generation for application authors," in *SELinux Symposium*, 2006. [Online]. Available: http://gelit.ch/td/SELinux/Publications/Mitre_Tools.pdf.

[39] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, Dave G. Andersen, and Jay Lepreau, "The Flask Security Architecture: System Support for Diverse Security Policies," in *Proceedings of the 8th USENIX Security Symposium, Washington, DC, USA, August 23-26, 1999*, USENIX Association, 1999. [Online]. Available: https://www.usenix.org/conference/8th-usenix-security-symposium/flask-security-architecture-system-support-diverse-security.

[40] Alexei Starovoitov and Daniel Borkmann, "Rework/optimize internal BPF interpreter's instruction set," Kernel patch, Mar. 2014. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8.

[41] Strace, *strace: linux syscall tracer*, Official strace website. [Online]. Available: https://strace.io (visited on 11/29/2020).

[42] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou, "Container Security: Issues, Challenges, and the Road Ahead," *IEEE Access*, vol. 7, pp. 52 976–52 996, 2019. DOI: 10.1109/ACCESS.2019.2911732.

[43] US Department of Defense, "Trusted Computer System Evaluation Criteria," DOD Standard DOD 5200.58-STD, 1983.

[44] Chris Wright, Crispin Cowan, Stephen Smalley, James Morris, and Greg Kroah-Hartman, "Linux Security Modules: General Security Support for the Linux Kernel," in *Proceedings of the 11th USENIX Security Symposium, San Francisco, CA, USA, August 5-9, 2002*, USENIX, 2002, pp. 17–31. [Online]. Available: http://www.usenix.org/publications/library/proceedings/sec02/wright.html.