BPFCon: Towards Secure and Usable Containers with eBPF

COMP5900I Preliminary Work Document

William Findlay

November 29, 2020

Abstract

Containers **[what do I want to say about containers?]**

In this paper, I present BPFCon[1], a novel approach to containers under the Linux kernel, built from the ground up as a light-weight yet secure process confinement solution for modern desktop applications. BPFCon leverages eBPF, a new Linux kernel technology, to implement fully application-transparent confinement without the need for a privileged setuid root wrapper application.

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

1 Introduction

[The final version of the paper will include an introduction partially adapted from my literature review.]

2 Background

[The background section will be mostly adapted from my literature review.]

2.1 The Process Confinement Problem

[This will be adapted from my literature review]

2.2 The Confinement Threat Model

[This will be adapted from my literature review]

2.3 Low-Level Isolation Techniques

Unix Discretionary Access Control [This will be adapted from my literature review]

POSIX Capabilities [This will be adapted from my literature review]

Namespaces and Cgroups [This will be adapted from my literature review]

System Call Interposition [This will be adapted from my literature review]

Linux Security Modules [This will be adapted from my literature review]

2.4 Containers

[This will be adapted from my literature review]

2.5 Extended BPF

3 BPFCon Design Goals

Five specific goals informed the design of BPFCon's policy language and enforcement mechanism, enumerated below as Design Goals D1 to D5. Note that there is a natural interplay between some of these design goals, while others are typically perceived as being in contention (specifically usability and security). In cases of conflict, it is essential to strike a careful balance between each property.

D1. Usability. BPFCon'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, BPFCon takes some inspiration from other high-level policy languages for containerized applications, such as those used in Snapcraft [5].

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

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

D5. Security. BPFCon should be built from the ground up with security in mind. In particular, security should not be an opt-in feature and BPFCon should adhere to the principle of least privilege [4] by default. It should be easy to tune a BPFCon policy to respond to new threats.

4 BPFCon Policy

BPFCon policy consists of simple manifests written in YAML [1], a human-readable data serialization language based on key-value pairs. Each BPFCon container is associated with a manifest, which 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 accordance with the principle of least privilege [4], BPFCon implements a strict default-deny policy, 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

Table 4.1: A list of accesses supported by BPFCon policy, along with their parameters, if any, and descriptions. Square brackets denote an optional parameter. **[This is currently non-exhaustive, come back and revise.]**

Access	Parameters	Description
`filesystem`	Mountpoint, [Read-only]	Grants access at the granularity of a filesystem mountpoint. This access may optionally be restricted to read-only.
`file`	Pathname, Access	Grants access at the granularity of individual files. Access may be specified as read, write, link, delete, or execute.
`directory`	Pathname, Access	Grants access at the granularity of individual directories. Access may be specified as read, write, link, delete, or chdir.
`network`	[Interface]	Grants access to network communications. A specific interface may optionally be specified.
`tty`	N/A	Grants access to tty devices.
`graphics`	N/A	Grants access to the graphical server.
`microphone`	N/A	Grants access to the microphone.
`sound`	N/A	Grants access to the microphone.

specific unwanted behaviour in a given program, without worrying about the details of constructing a rigorous security policy.

As a motivating example of BPFCon security policy, consider Discord [2], a popular cross-platform voice chat client designed for gamers. Discord comes with an optional feature, "Display Active Game", which displays whatever game the user is currently playing in their status 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` [6] 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 BPFCon policy like the examples depicted in Listing 4.1 and Listing 4.2.

Listing 4.1: A sample manifest for Discord [2] using BPFCon'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` overrides the access right on the root filesystem.

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

Listing 4.2: A sample manifest for Discord [2] using BPFCon'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
```

To run a BPFCon container, the user invokes `bpfcon run <name>` where `name` is the unique container name declared in the manifest. The `bpfcon run` command is a thin wrapper around that target application, whose only purpose is to invoke a special library call, `bpfcon_confine()`, that marks the process group for confinement before executing the command(s) defined in the manifest. Besides the fact that it invokes a special library call, there is nothing special about the `bpfcon run` wrapper—nothing is stopping a typical application from invoking `bpfcon_confine()` directly, as it requires no special privileges. In this sense, one can also think of BPFCon as a mechanism for *unprivileged self-confinement*. Section 5 describes in detail the specifics of how BPFCon implements its confinement mechanism.

5 BPFCon Implementation

6 Evaluation

7 Discussion

8 Related Work

9 Conclusion

10 Acknowledgements

The idea for BPFCon was conceived during a discussion with Anil Somayaji. This work directly builds on my previous work, BPFBox [3], first presented in a paper co-authored with Anil Somayaji and David Barrera.

References

[1] 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).

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

[3] William Findlay, Anil Somayaji, and David Barrera, "bpfbox: Simple Precise Process Confinement with eBPF," in *Proceedings of the 2020 ACM Cloud Computing Security Workshop (CCSW'2020)*, To appear, Nov. 2020. DOI: 10.1145/3411495.3421358. [Online]. Available: https://www.cisl.carleton.ca/~will/written/conference/bpfbox-ccsw2020.pdf.

[4] 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.

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

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