BPFContain: Towards Secure and Usable Containers with eBPF

COMP5900I Preliminary Work

William Findlay

November 30, 2020

Abstract

Containers are becoming an increasingly important part of the Linux ecosystem. Containerized package managers like Snapcraft [14] and FlatPak [8] enable easy distribution and dependency management for desktop applications, while Docker [7] and Kubernetes [9] 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]

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, specifically targeting its access to security-sensitive system resources such as files, network interfaces, and other running applications. Despite decades of work following Lampson's [10] first proposal of the process confinement problem in 1973, process confinement remains a somewhat open problem to date [5]. **[Probably a lot more to say here]**

2.2 The Confinement Threat Model

To understand why process confinement is a desirable goal in operating system security, we must first identify the credible threats that process confinement addresses. To that end, here I describe three attack vectors (items A1 to A3), followed by three attack goals (items G1 to G3) which highlight just a few of the threats posed by unconfined processes to system security, stability, and user privacy.

A1. Compromised processes. Unconfined running processes have classically presented a valuable target for attacker exploitation. With the advent of the Internet, web-facing processes that handle untrusted user input are especially vulnerable, particularly as they often run with heightened privileges [3]. An attacker may send specially crafted input to the target application, hoping to subvert its control flow integrity via a classic buffer overflow, return-oriented programming [12], or some other means. The venerable Morris Worm, regarded as the first computer worm on the Internet, exploited a classic buffer overflow vulnerability in the `fingerd` service for Unix, as well as a development backdoor left in the `sendmail` daemon [15]. In both cases, proper process confinement would have eliminated the threat by preventing the compromised programs from impacting the rest of the system.

A2. Semi-honest software. Here, I define semi-honest software as that which appears to perform its desired functionality, but which additionally may perform some set of unwanted

[2] Note for Anil: When you see orange square brackets **[like this]**, treat them as a todo.

actions without the user's knowledge. Without putting a proper external confinement mechanism in place to restrict the behaviour of such an application, it may continue to perform the undesired actions *ad infinitum*, so long as it remains installed on the host. As a topical example, an `strace` of the popular Discord [6] voice communication client on Linux reveals that it repeatedly scans the process tree and reports a list of *all applications* running on the system, even when the user has turned off the "display active game" feature[3]. This scanning behaviour represents a clear violation of the user's privacy expectations.

A3. Malicious software. In contrast to semi-honest software, malicious software is that which is expressly designed and distributed with malicious intent. Typically, this software would be downloaded by an unsuspecting user either through social engineering (e.g. fake antivirus scams) or without the user's knowledge (e.g. a drive-by download attack). In the case of a computer virus, malicious software may replicate itself on the host by infecting other (originally benign) binaries. It would be useful to provide the user with a means of running such potentially untrustworthy applications in a sandbox so that they cannot damage the rest of the system.

G1. Installation of backdoors/rootkits. Potentially, the most dangerous attack goal in the exploitation of unconfined processes is the establishment of a backdoor on the target system. A backdoor needn't be sophisticated—for example, installing the attacker's RSA public key in `ssh`'s list of authorized keys would be sufficient—however, the most sophisticated backdoors may result in permanent and virtually undetectable escalation of privilege. For instance, a sophisticated attacker with sufficient privileges may load a *rootkit* [1] into the operating system kernel, at which point she has free reign over the system in perpetuity (unless the user manages to somehow remove the rootkit or reinstalls the infected operating system).

G2. Information leakage. An obvious goal for attacks on unconfined processes (and indeed the focus of the earliest literature on process confinement [10]) is information leakage. An adversary may attempt to gain access to personal information or other sensitive data such as private keys, password hashes, or bank credentials. Depending on the type of information, an unauthorized party may not even necessarily require elevated privileges to access it—for instance, no special privileges are required to leak the list of processes running on a Linux system (as in the case of Discord [6] highlighted above).

G3. Denial of service. A compromised process could be used to mount a denial of service attack against the host system. For example, an attacker could take down network interfaces, consume system resources, kill important processes, or cause the system to shut down or reboot at an inopportune moment.

[3]This feature allows Discord to report, in the user's status message, what game the user is currently playing. The "display active game" feature appears to be the original motivation behind scanning the process tree.

As shown in the examples above, unconfined processes can pose significant threats to system security and stability as well as user privacy. The advent of the Internet has exacerbated many of these threats. Unconfined network-facing daemons continually process untrusted user input, resulting in an easy and highly valuable target for attacker exploitation. Email and web browsers have enabled powerful social engineering and drive-by download attacks, which often result in the installation of malicious software. Semi-honest software can violate user expectations of security and privacy by performing unwanted actions without the user's knowledge. It is clear that a solution is needed to mitigate these threats—for this, we turn to process confinement.

2.3 Low-Level Isolation Techniques

The Linux kernel supports a variety of lower-level abstractions for implementing virtualization, access control, 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 2.4) to achieve confinement.

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

[Write this]

3 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 [14].

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 [11] by default. It should be easy to tune a BPFContain policy to respond to new threats.

3.1 BPFContain Policy

BPFContain policy consists of simple manifests written in YAML [2], 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 3.1 describes the various access labels that can be used in BPFContain policy.

Following the principle of least privilege [11], 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

Table 3.1: A list of accesses supported by BPFContain policy, along with their parameters, if any, and descriptions. Square brackets denote an optional parameter. N/A denotes an access which cannot be parameterized. **[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.
`ipc`	Container	Grants access to communicate with processes in *another* BPFContain container.
`tty`	N/A	Grants access to tty devices.
`video`	N/A	Grants access to video devices.
`sound`	N/A	Grants access to sound devices.

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

As a motivating example of BPFContain security policy, consider the Discord client, discussed briefly in Section 2.2. 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 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` [16] 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 3.1 and Listing 3.2.

Listing 3.1: A sample manifest for Discord [6] 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 3.2: A sample manifest for Discord [6] 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 3.1), 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 3.2), 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.

3.2 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 3.1) 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, `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.

3.3 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 [4, 13]. 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

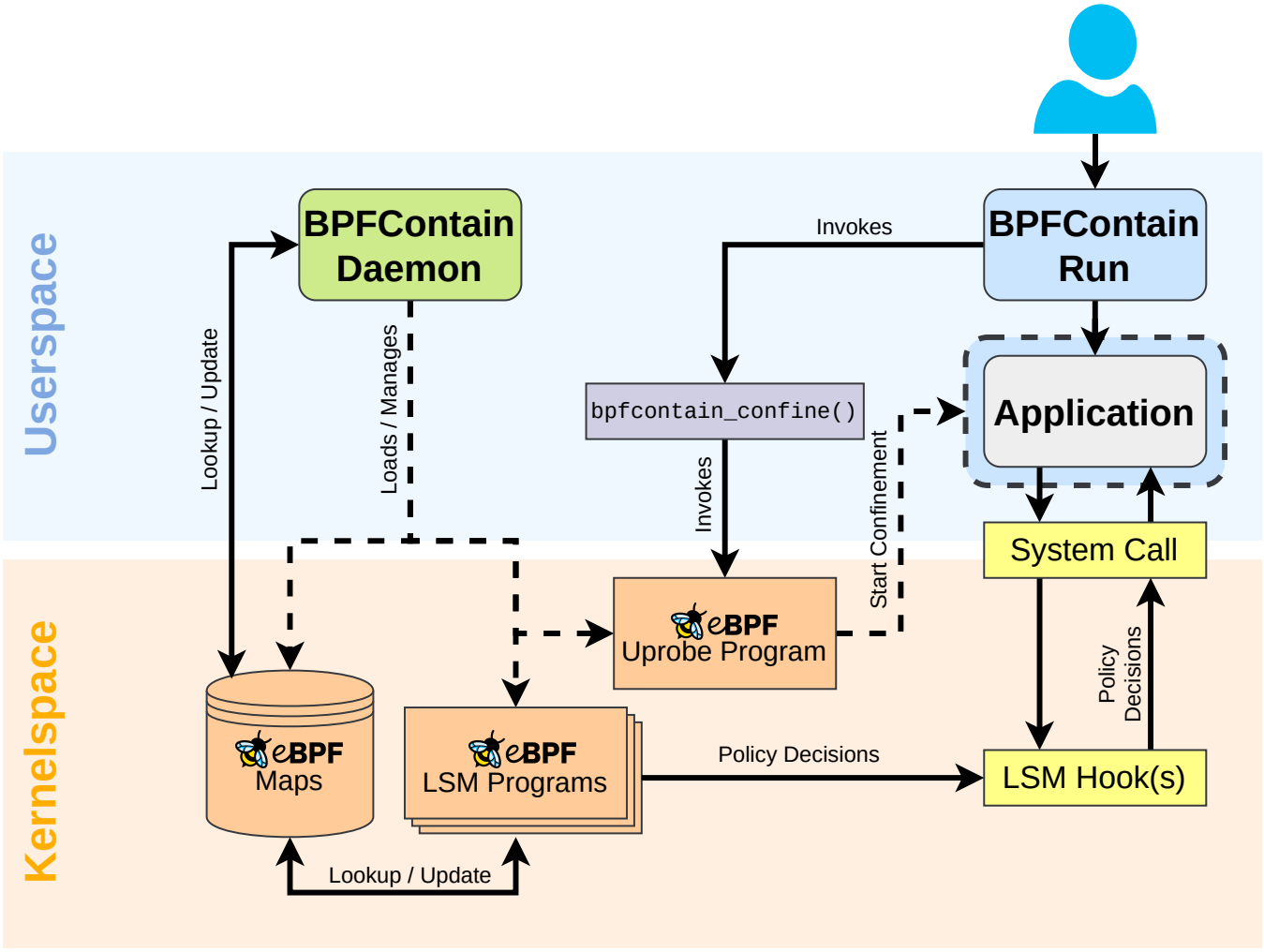

Figure 3.1: A diagram of BPFContain's architecture. The privileged daemon (green) is responsible for loading the necessary eBPF maps and programs (orange) into the kernel and managing their lifecycle. The user starts a container by executing an unprivileged wrapper application (blue), which invokes the `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.

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.

4 Evaluation

[Write this]

5 Discussion

[Write this]

6 Related Work

[This will be adapted from my literature review.]

7 Conclusion

[Write this]

8 Acknowledgements

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

References

[1] 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.

[2] 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).

[3] Frederick B. Cohen, "A Secure World-Wide-Web Daemon," *Comput. Secur.*, vol. 15, no. 8, pp. 707–724, 1996. DOI: 10.1016/S0167-4048(96)00009-0.

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

[5] 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.

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

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

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

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

[10] 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.

[11] 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.

[12] Hovav Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07, Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 552–561, ISBN: 9781595937032. DOI: 10.1145/1315245.1315313. [Online]. Available: https://doi-org.proxy.library.carleton.ca/10.1145/1315245.1315313.

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

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

[15] Eugene H. Spafford, "The Internet Worm Incident," in *ESEC '89, 2nd European Software Engineering Conference, University of Warwick, Coventry, UK, September 11-15, 1989, Proceedings*, ser. Lecture Notes in Computer Science, vol. 387, Springer, 1989, pp. 446–468. DOI: 10.1007/3-540-51635-2_54.

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