bpfbox: Simple Precise Process Confinement with eBPF

William Findlay Anil Somayaji David Barrera

Carleton University
will@ccsl.carleton.ca

November 5, 2020

Fall 2020 CCSL/CISL Talk

Outline of this Talk

1. The Process Confinement Problem
2. The Status Quo
3. eBPF 101
4. bpfbox Overview
5. bpfbox Design and Implementation
6. bpfbox Performance Evaluation
7. The Future of eBPF and Security
8. Conclusion

The Process Confinement Problem

What is Process Confinement?

We want to be able to **confine** our **processes**.

- Also known as *sandboxing*

Why do we want to do this?

- Default protection mechanisms are too:
  - Granular
  - User-centric
  - Discretionary
- Protection can be **overridden**
  - Superuser (*nix)
  - Administrator (Windows)

How do we protect the user from *themselves* and *their own processes*?

An Interesting Question

How many processes do you think are running on your computer **right now**?

- Probably a lot more than you think
- You probably didn't start most of them yourself
- You might not even know what some of them are for

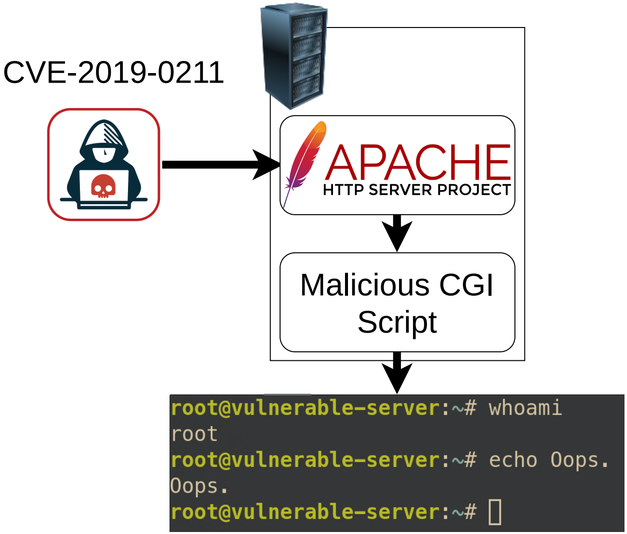
Threat Model

Compromised Processes

- Web servers
- Daemons
- Chat applications
- etc.

The Morris Worm

- Backdoor in `sendmail` daemon
- Buffer overflow in `fingerd`
- Estimated damage: \$100,000–\$10,000,000



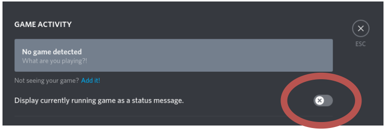
Threat Model

Semi-Honest Software

- Software that does its job...
- But also performs potentially **unwanted actions**

```
[16:13] hh@arch:~ | strace -fqo /tmp/discord.log discord-canary
```

```
912433 readlink("/proc/506824/exe", "/opt/draw.io/drawio", 1023) = 19
912433 readlink("/proc/634535/exe", "/usr/lib/firefox/firefox", 1023) = 24
912433 readlink("/proc/909407/exe", "/usr/bin/alacritty", 1023) = 18
912433 readlink("/proc/3205375/exe", "/usr/bin/tmux", 1023) = 13
912433 readlink("/proc/3244098/exe", "/usr/bin/zsh", 1023) = 12
912433 readlink("/proc/3387/exe", "/usr/bin/picom", 1023) = 14
912433 readlink("/proc/3412/exe", "/usr/bin/xbindkeys", 1023) = 18
912433 readlink("/proc/3421/exe", "/usr/bin/volnoti", 1023) = 16
912433 readlink("/proc/3449/exe", "/usr/bin/pulseaudio", 1023) = 19
```

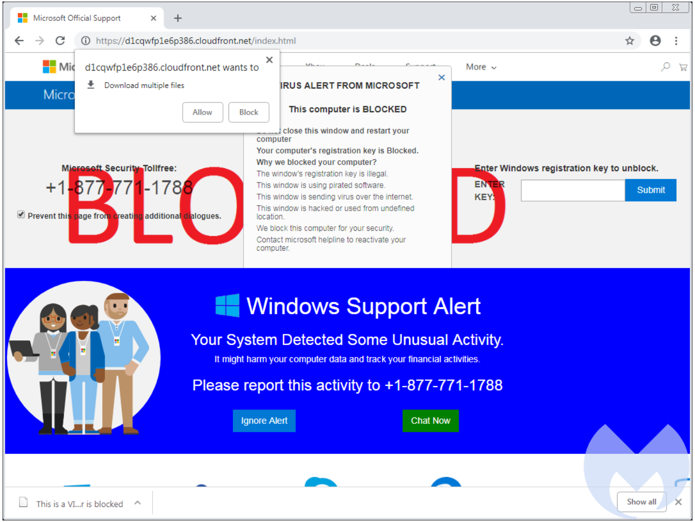


Umm... What?

Threat Model

Malicious Software

- Viruses
- Trojans
- Ransomware
- Spyware
- etc.



Threat Model

Attack goals?

- Installing backdoors/rootkits
- Information leakage
- Denial of service
- Data ransom
- Setting up a botnet

Process confinement reduces the attack surface.

The Process Confinement Problem

- “A Note on the Confinement Problem” (Lampson, 1973)

Operating Systems — C. Weissman, Editor

A Note on the Confinement Problem

Butler W. Lampson
Xerox Palo Alto Research Center

- An open problem for nearly **five decades**

The Status Quo

Unix DAC

POSIX Capabilities

Namespaces and Cgroups

System Call Interposition

Linux MAC

Containers / Containerized Package Management

eBPF 101

eBPF in the Beginning

eBPF $\equiv$ **E**xtended **B**erkley **P**acket **F**ilter

- But it has little to do with Berkley, packets, or filtering nowadays
- The name BPF is preserved for historical reasons

So then **what is eBPF?**

- A major re-write of the Linux BPF engine
  - Alexei Starovoitov and Daniel Borkmann
- Merged into the Linux kernel in 2014
- The original goal was fine-grained, cross-layer **system introspection**

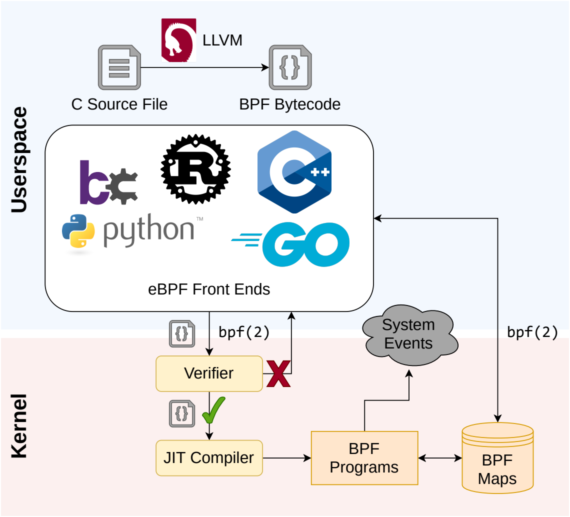
What Can eBPF Do?

eBPF

- Direct Packet Access
- Userspace Functions
- Hardware
- LSM Hooks
- TCP / IP
- System Calls
- Block Device I/O
- Sockets
- Kernelspace Functions
- Stack Traces

And much more...

How eBPF Works



Verifiably Safe Programs

Restricted execution context.

- 512 byte stack limit
- 11 registers (10 general purpose)
- Memory access must be bounds-checked
- No unbounded loops
- No back-edges in control flow

eBPF in 2020

eBPF is now **more than just an observability tool**.

- eBPF provides a **safe**, **efficient**, and **flexible** way for privileged users to extend the kernel
- eBPF turns Linux into a **programmable kernel**

Linux 5.7 → KRSI (**K**ernel **R**untime **S**ecurity **I**nstrumentation)

- Attach BPF programs to LSM hooks
- Make security decisions and generate audit logs with eBPF

KRSI: BPF LSM Programs

- TODO explain KRSI with a picture

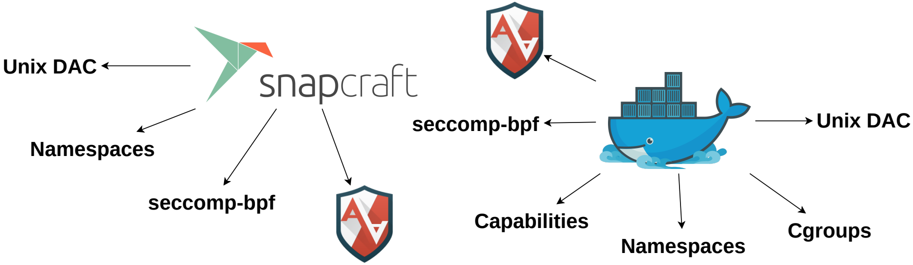
bpfbox Overview

bpfbox at a Glance

- bpfbox is a novel **process confinement mechanism** for Linux
  - Using a new Linux technology called eBPF
- Users write per-application policy in a **simple policy language**
- Policy is enforced by attaching **eBPF programs** to **LSM hooks**
  - Integrates cross-layer state into policy decisions

Motivation

- Existing process confinement mechanisms are **complex**



- Existing process confinement mechanisms are **difficult to use**

SELinux AppArmor TOMOYO

- Can we do any better?

eBPF Changes the Game

eBPF enables:

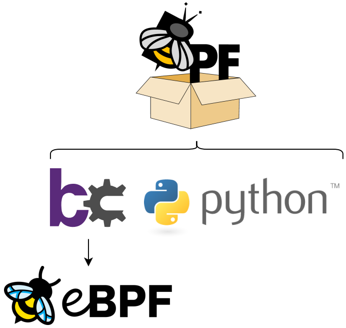
- Fine-grained system introspection
- Integration of **cross-layer state** with policy enforcement
- Rapid prototyping
- Safe production deployment of new security solutions

We have an opportunity to **rethink process confinement** from the ground up.

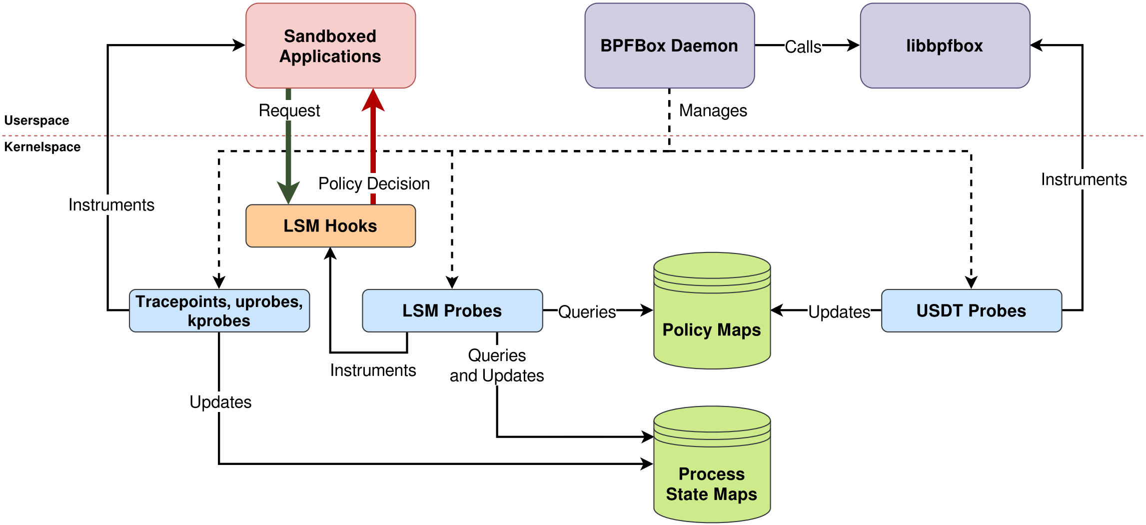
bpfbox Design and Implementation

bpfbox Implementation

- Userspace daemon using the Python3 bcc framework
- Kernelspace components are all written in eBPF
  - LSM probes (KRSI), kprobes, uprobes
  - Under 2000 source lines of kernelspace code
- Thanks to eBPF, bpfbox is **light-weight**, **flexible**, and **production-safe**
  - Works out of the box on any vanilla Linux kernel ≥ 5.8



bpfbox Architecture



Policy Design Goals

1. **Simplicity**
   - Policy should be simple enough for ad hoc confinement
2. **Application transparency**
   - Policy should not require changes to the confined application
3. **Flexibility**
   - Policy should offer optional layers of granularity
4. **Security**
   - Policy should follow the principle of least privilege
   - It should be difficult to write an insecure policy

Rules and Directives

Rules specify access to system objects:

- `fs(file, access)`
- `net(socket, access)`
- `signal(prog, sig)`
- etc.

Directives augment blocks of rules:

- `#[directive]` syntax
- Specify **actions to be taken** on a block of rules
- Add **additional context** to a block of rules

Taints and Transitions

- `#[taint]` → Start confinement
- `#[transition]` → Switch profiles on execve

```
#![profile "/bin/mywebdaemon"]

#[taint] {
    net(inet, any)
    net(inet6, any)
}

/* ... */

#[transition] {
    fs("/bin/myhelper", getattr|read|exec)
}
```

Policy at the Function Call Level

- `#[func "foo"]` → Apply rules only within a call to `foo()`
- `#[kfunc "foo"]` → Same thing, but for kernel functions

```
#![profile "/sbin/mylogin"]

#[func "check_password"]
#[allow] {
    fs("/etc/passwd", read)
    fs("/etc/shadow", read)
}

#[func "add_user"]
#[allow] {
    fs("/etc/passwd", read|append)
    fs("/etc/shadow", read|append)
}

/* ... */
```

bpfbox Performance Evaluation

Methodology

- Phoronix Test Suite OSBench
  - Measures basic OS functionality
  - (spawning processes, memory allocations, etc.)
- Phoronix Test Suite Apache
  - Benchmark Apache `httpd` packets per second
- Kernel compilation benchmarks
  - Measure Linux kernel compilation performance
  - Heavy workload, spawning lots of processes

Methodology

Two modes of operation for each test.

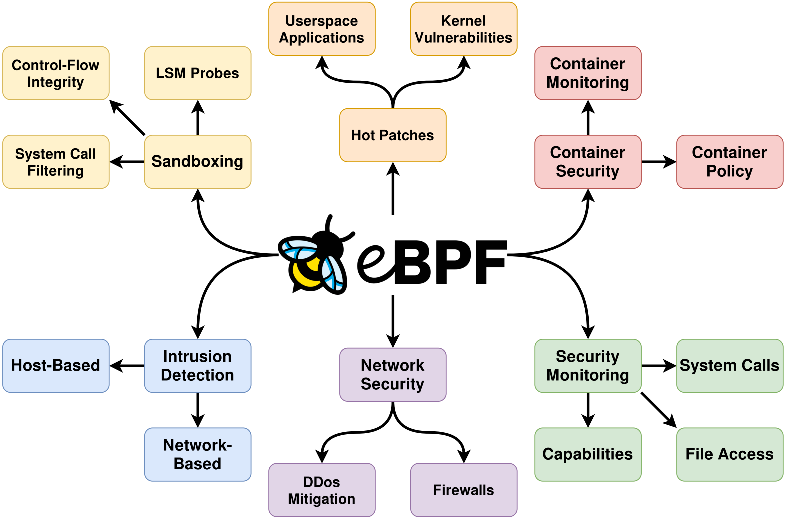
- Passive mode
  - bpfbox and AppArmor instrument hooks, but do not enforce or audit
  - Test lowest possible overhead
- Complaining mode
  - bpfbox and AppArmor complain about (log) every security-sensitive operation
  - Test worst case overhead

Results

- Phoronix OSBench
  - Passive: bpfbox is **roughly equivalent** to AppArmor
  - Complaining: bpfbox performs **significantly better** than AppArmor
- Phoronix Apache
  - bpfbox and AppArmor are **roughly equivalent**
- Kernel compilation
  - Passive: bpfbox is **roughly equivalent** to AppArmor
  - Complaining: bpfbox performs **better in kernelspace** overhead and **worse in userspace** overhead

The Future of eBPF and Security

Security Applications of eBPF



New Directions

Userspace LSM (Self-Confinement)

- Attach `uprobes` to a shared library
- Userspace applications make calls to the library to declare privileges
- `uprobes` update a policy map in kernelspace

Dynamic Capabilities

- Users define custom capabilities
- Enforced in kernelspace with dynamic LSM probes
- E.g. `CAP_ACCESS_PHOTOS` to grant access to `~/pictures`

New Directions

Hot Patches (Userspace)

- Patch vulnerabilities before security updates are available
- `uprobes` to hook into functions
- `bpf_probe_write_user()` to replace userspace memory

Hot Patches (Kernel)

- Replace vulnerable kernel functions with BPF programs
- Alter/drop malicious packets before they reach the networking stack
- E.g. patch packet-of-death vulnerability with an XDP program

Conclusion

bpfbox Future Work

- TODO

Acknowledgements

Special thanks to:

- **Alexei Starovoitov** and **Daniel Borkmann** (creators of eBPF)
- **K.P. Singh** (creator of KRSI)
- Fellow **bcc contributors** (an awesome eBPF framework)
- Anonymous **CCSW'2020 reviewers** (valuable feedback)

This work was supported by NSERC through a Discovery Grant.

Contributions

- First **policy enforcement engine** written in **eBPF**
- Integration of **userspace** and **kernelspace** state with **LSM layer enforcement**
- A simple policy language for **ad hoc process confinement**
  - But with optional complexity for **fine-grained protection**

github.com/willfindlay/bpfbox

Check out the project on GitHub!