Tom Cassar

addrfilter

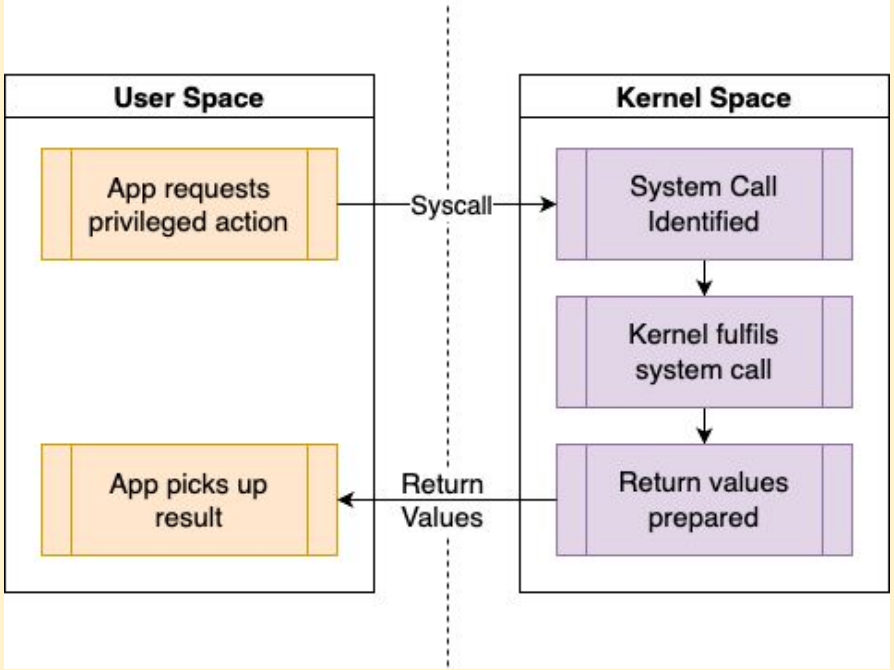
Fine-Grained System Call Filtering for Linux

What do any of those words mean?

System Calls

- User space asking the kernel to do something privileged

System Calls



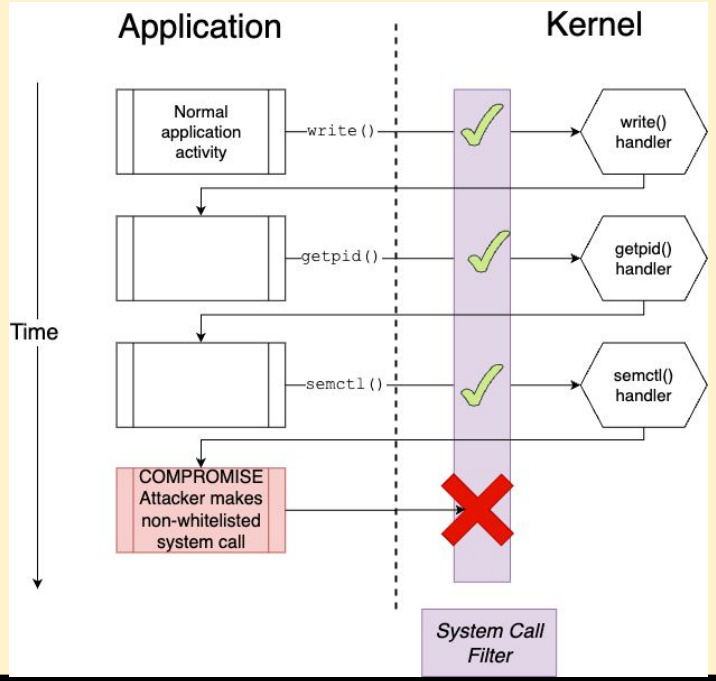
System Call *Filtering*

- User space asks to do something it shouldn't
  =>
  compromised! *(probably)*

- Take action: kill the offending process

- Lightweight, effective against 0-days, Principle of Least Privilege

System Call *Filtering*



Seccomp

Industry Standard System Call Filter

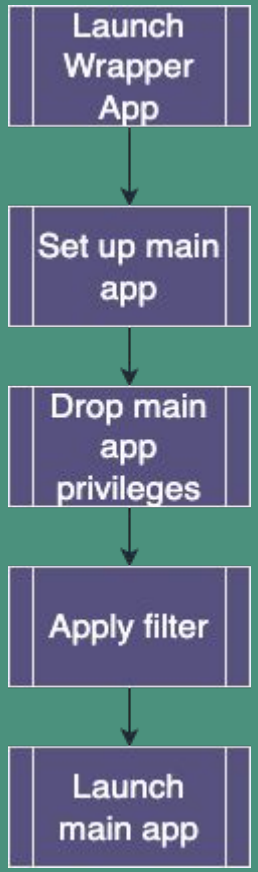
- Commonly used system call filter (2005)
- Everywhere:
  - Docker containers
  - Android zygote
  - Firefox (tab isolation)
  - Package Managers (Flatpak, AppImage)

Seccomp

How does it work?

1. Generate a whitelist
2. Mount the filter to an application
3. Done!

So, *what's the problem*?



Seccomp is
too lenient!

- Applications are getting larger
  - `/bin/true`: 0 LoC when first released, 2.3k in 2012
  - `/bin/bash`: 11.3KB in 1974, 2.1MB in 2014
- Applications need to make **more system calls**
- => filters less effective!
  - (attacker has more *tools* to exploit the system)

(DeMarinis et al., RAID 2020)

Solution:
`addrfilter`

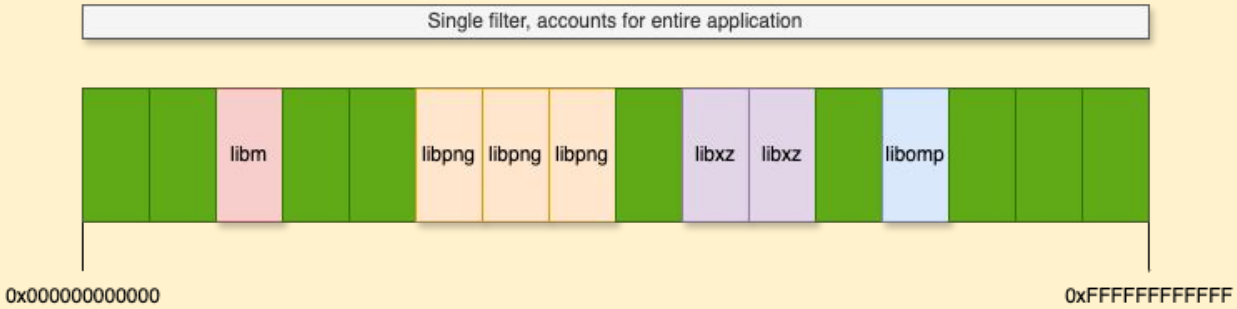
Fine-Grained Filtering

Seccomp applies a single whitelist to an application

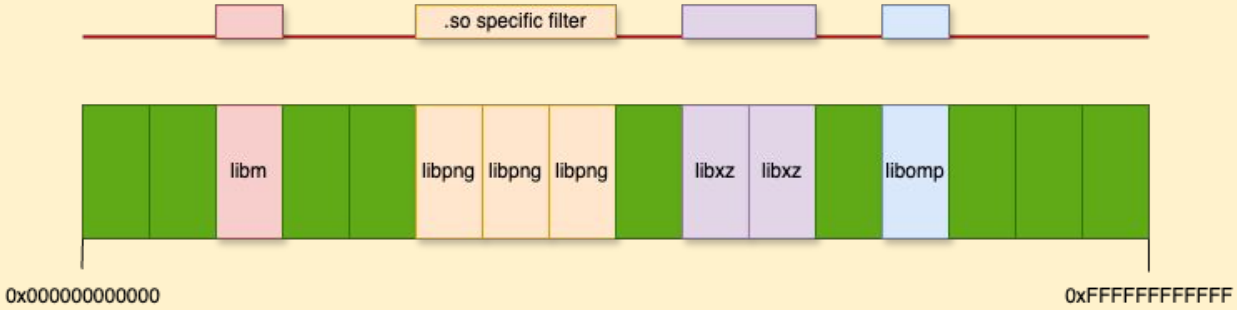
`addrfilter` applies a *different whitelist to each part of the process's address space;*

Smaller, specialised whitelists = more secure!

A Single Seccomp Filter



`addrfilter` Fine-Grained Filters

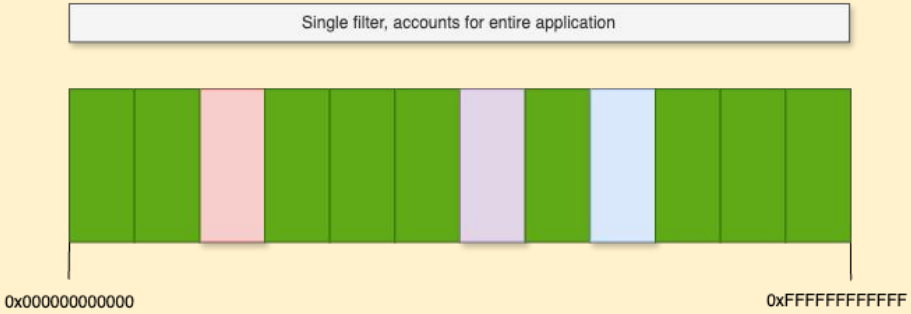


A Small Example

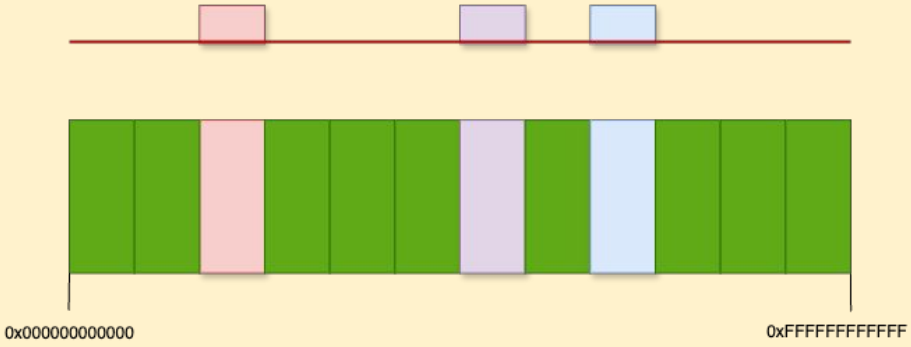
Imagine an executable with *two shared libraries*

The first shared library only calls `getpid()`
The second only calls `write()`

A Single Seccomp Filter



`addrfilter` Fine-Grained Filters



Whole app needs
- `write()`
- `getpid()`
- ...
- (setup, coordination, etc.)

seccomp

First shared library needs:
- `getpid()`

Second shared library:
- `write()`

(setup, coordination calls are associated with relevant `.so`s)

`addrfilter`

First compartment calls `write()`

=> seccomp sees no issue; no compromise detected

seccomp

First compartment calls `write()`

=> `addrfilter` detects disallowed system call; compromise detected

`addrfilter`

Second compartment calls `getpid()`

=> seccomp sees no issue; no compromise detected

seccomp

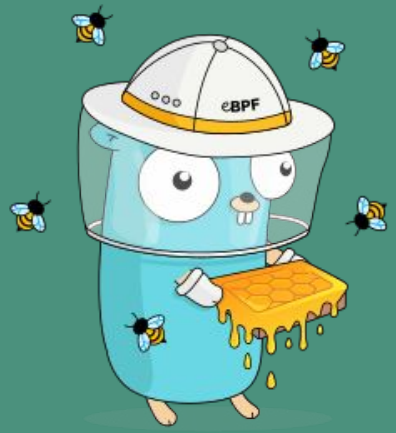
Second compartment calls `getpid()`

=> `addrfilter` detects disallowed system call; compromise detected

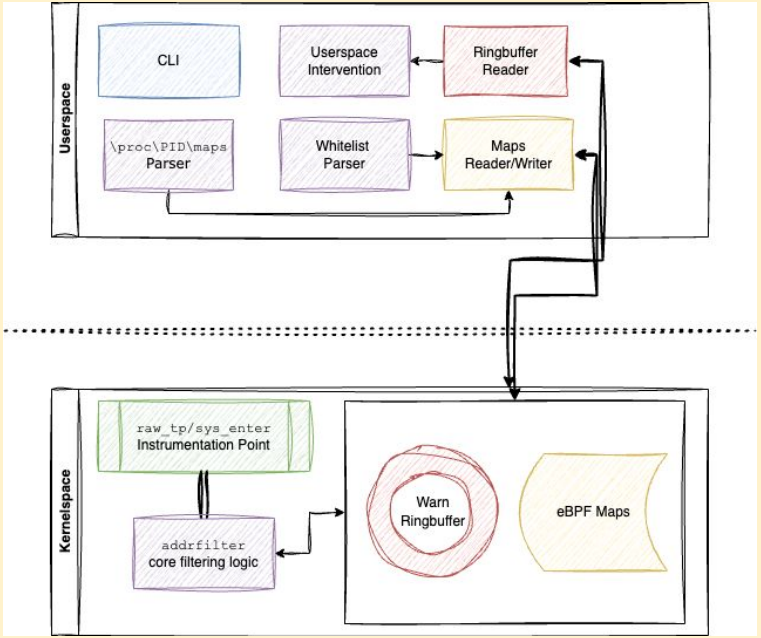
`addrfilter`

Live Demo Time!

How does it work?



briefly


Userspace
CLI
Userspace Intervention
Ringbuffer Reader
\proc\PID\maps Parser
Whitelist Parser
Maps Reader/Writer
Kernelspace
raw_tp/sys_enter Instrumentation Point
addrfilter core filtering logic
Warn Ringbuffer
eBPF Maps

Kernel Space

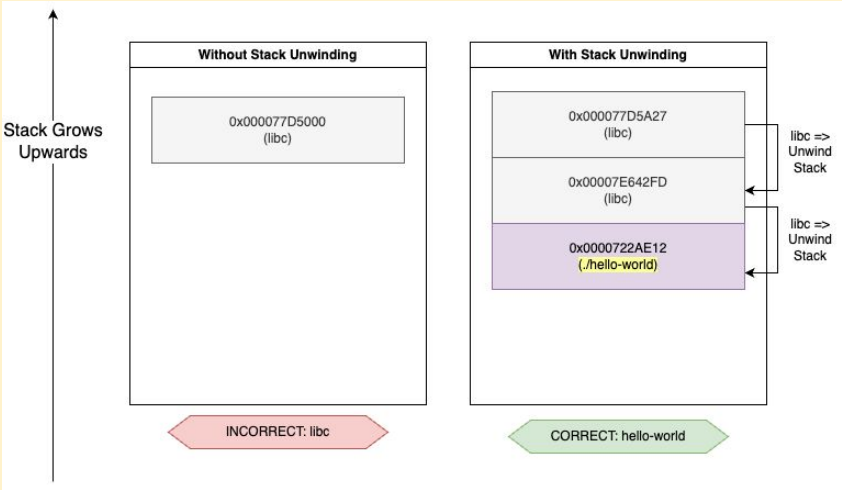
On every system call...

Kernel Space

1. Check PID - are we protecting this process?

Kernel Space

2. Who made the system call?
    a. Unwind the stack
    b. Identify **first non-libc return pointer** (RP)




```c
#include <stdio.h>

int main() {
    // printf @ 0x0000722AE12
    printf("Hello, World!");
}
```

Kernel Space

3. Figure out which shared library the RP came from

Kernel Space

4. Check if the current system call is whitelisted: if not, SIGKILL!

User space

- Parse whitelist, setup in kernel space
- Spawns application (handles permissions)
- Metrics, logging, CLI

Redis:
A Real Example

Secure?

```
~/dissertation/report/evaluation-artefacts/results main* ❱ \
python3 score_syscalls.py -i redis/syso-raw.json -r syscall-ranking.yaml
addrfilter: 20
seccomp: 45
privilege reduction: 55.56%
~/dissertation/report/evaluation-artefacts/results main* ❱
```

What is *Privilege Reduction?*

Privilege Reduction

1. Each system call gets a score
   a. *Dangerous* system calls = 3 (e.g. `execve()`, `chmod()`, `mount()`)
   b. *Medium* = 2
   c. *Safe* = 1

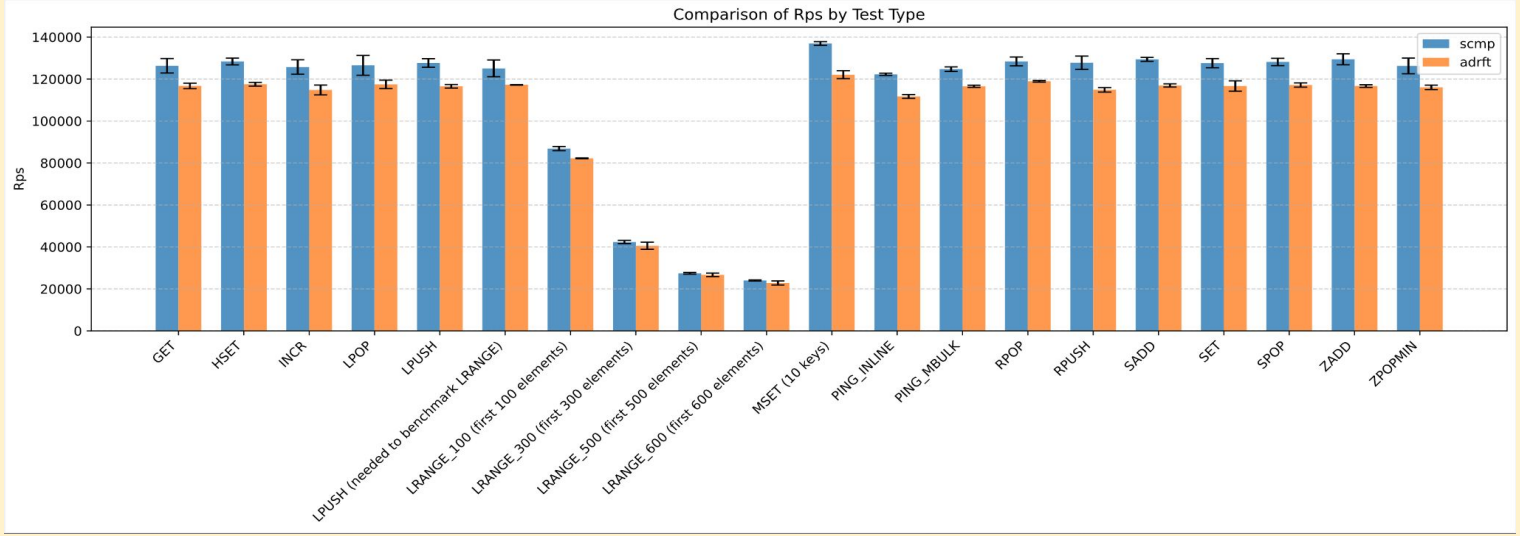
> using Mussa et al., IEEE HASE 2014

Privilege Reduction

2. Sum scores of system calls *in a compartment* to get a total danger score
3. For `addrfilter`, let total privilege equal the *most privileged compartment*

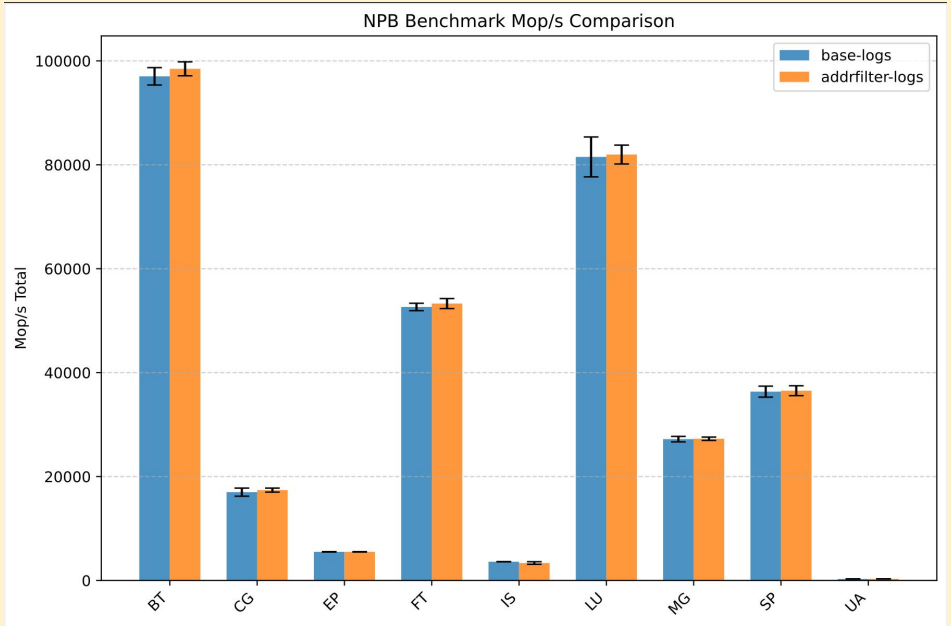
=> we are **underestimating privilege reduction**

Slow? Depends...



For redis, slower than a seccomp filter (~24%)

Slow? Depends...



Some cases, no significant slowdown vs **no system call filter!**

Some Limitations…

DLEs Only...

1. `addrfilter` applies a whitelist to each shared library present in a process's address space
2. Static linking => no shared libraries => `addrfilter` is functionally equivalent to seccomp!

On startup?

1. `addrfilter` protects based on Process ID (PID)
2. Need to wait for a process to spawn before we have it's PID
3. => period of time where addrfilter is disabled
   a. In the `./linked` example, this is why we slept for 0.375s

Users and Permissions

1. `addrfilter` executable needs `CAP_SYS_ADMIN` to mount eBPF
2. Need to ensure that filtered app does NOT have `CAP_SYS_ADMIN` privileges (otherwise can unmount filter)
3. Achieved through changing UID/GID; brittle…

Project Challenges

eBPF

1. Very restrictive (not even Turing complete)
2. Poorly documented
3. Constantly and rapidly evolving

Linux Kernel and Systems Security

1. No prior experience in either
2. Both massive topics, lots of ‘footguns’ (e.g. ASLR kicking in on `execve()`)

A lot of code

1. 5000 lines for `addrfilter` (750 bpf)
2. 1500 for syso (analysis tool, gave privilege reduction scores)
3. 3000 lines to parse benchmarks, generate metrics, etc (not DRY...)

addrfilter

- Publicly listed on GitHub under MIT @ github.com/tcassar-diss
  - addrfilter source code @ tcassar-diss/addrfilter
  - syso analysis tool @ tcassar-diss/syso
  - evaluation artefacts and analysis scripts @ tcassar-diss/evaluation-artefacts
  - Report (src) @ tcassar-diss/report