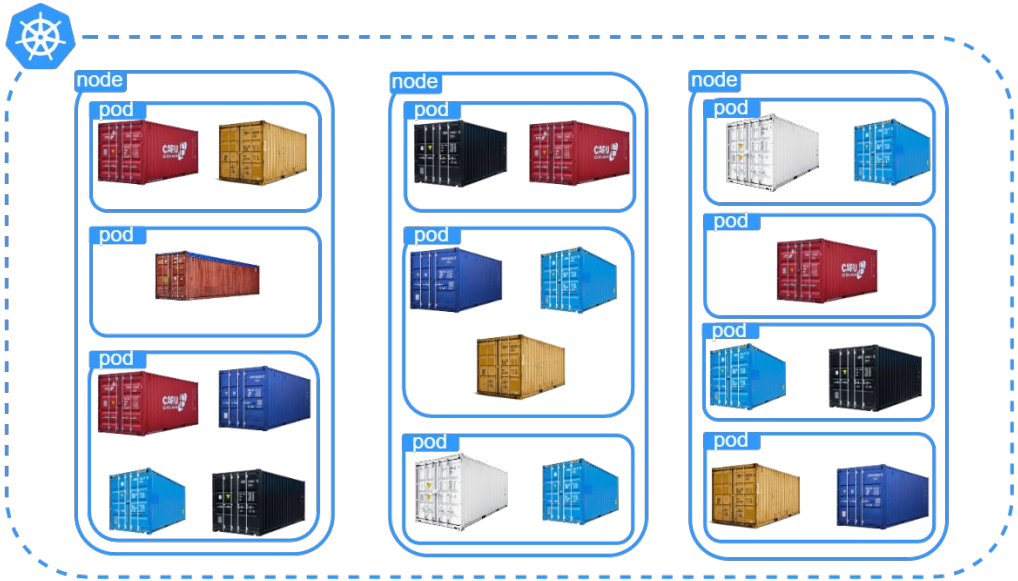
Beyond the Surface

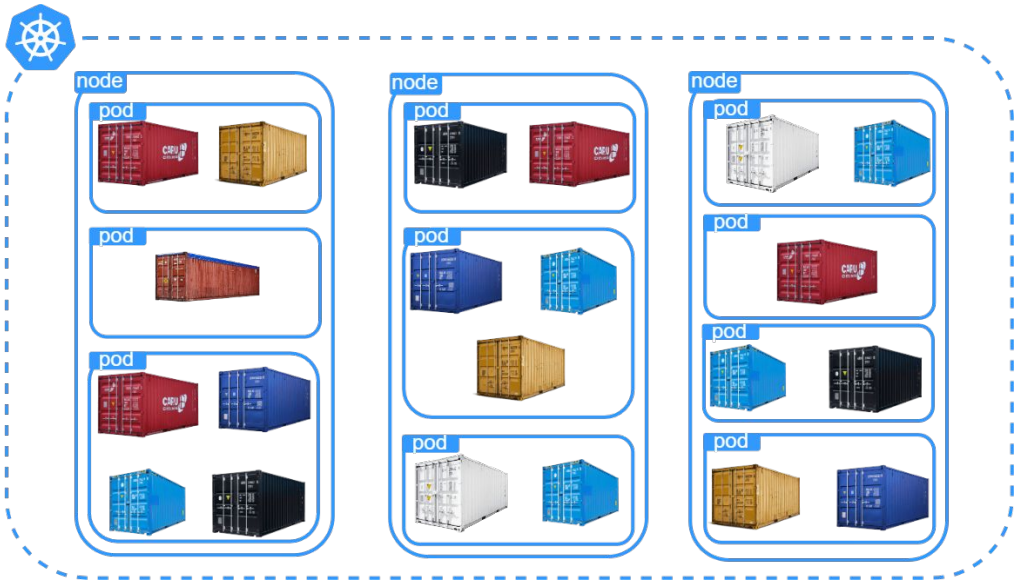
Deep Dive into Kernel Observability with eBPF

Giacomo Belocchi

How to monitor what happens in the cluster?



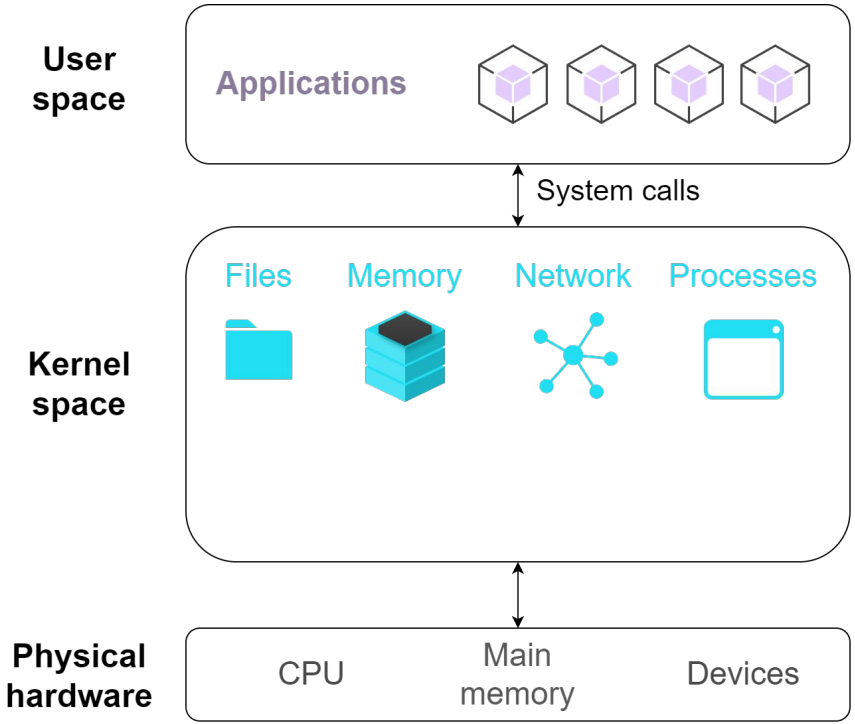
How to monitor what happens in the cluster?



What if a Kubernetes administrator want to observe what happens?

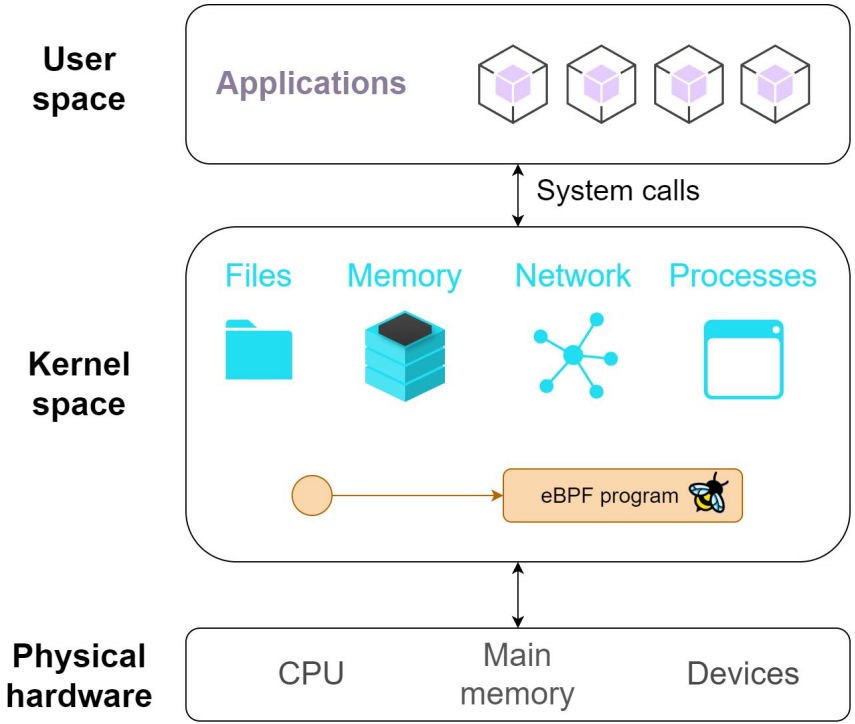
🤯

Kernel



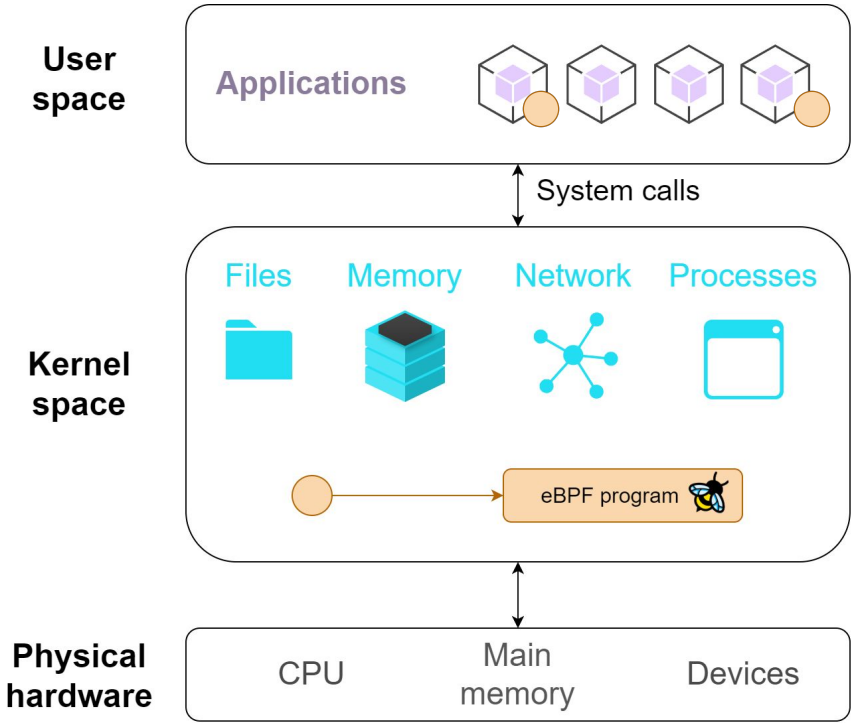
- User space where applications run
- Applications can’t directly access hardware resources
- Applications use the kernel making syscalls
- File read/write, memory accesses, … all go through the kernel

Kernel - eBPF to the rescue



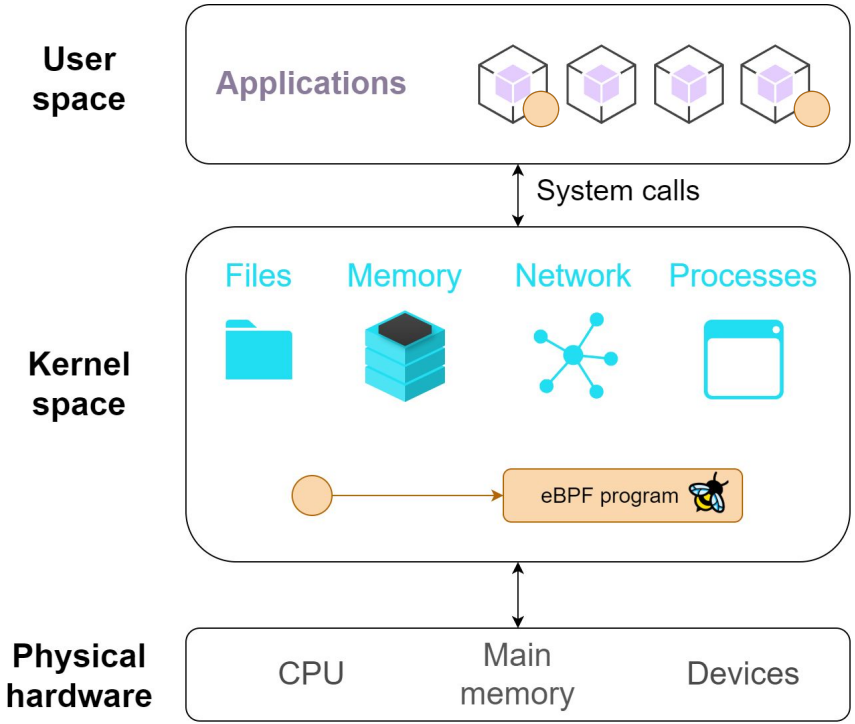
- Hooks inside the kernel

Kernel - eBPF to the rescue



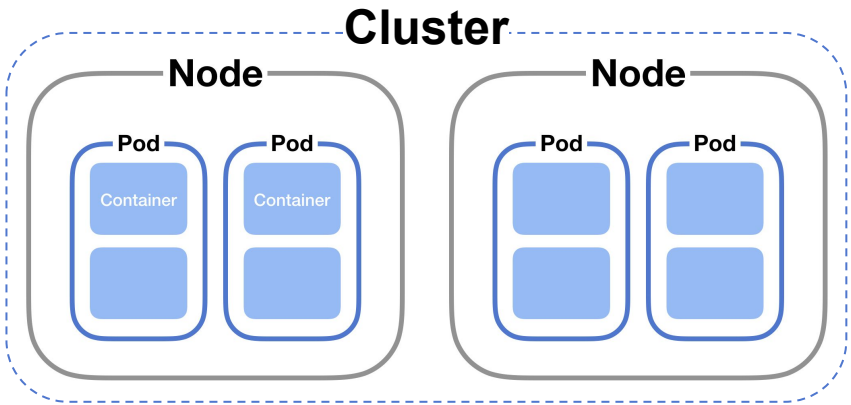
- Hooks inside the kernel
- Or inside user space applications

Kernel - eBPF to the rescue

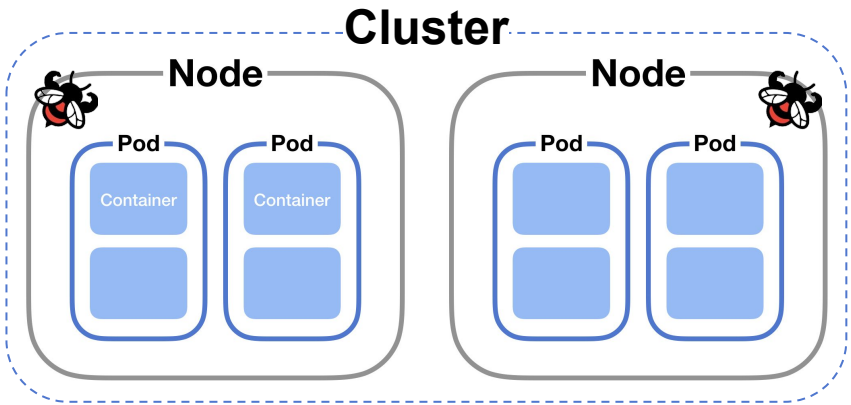


- Hooks inside the kernel
- Or inside user space applications
- When execution reach the hook
  ⇒ eBPF program is invoked
- eBPF program can access data visible at the hook

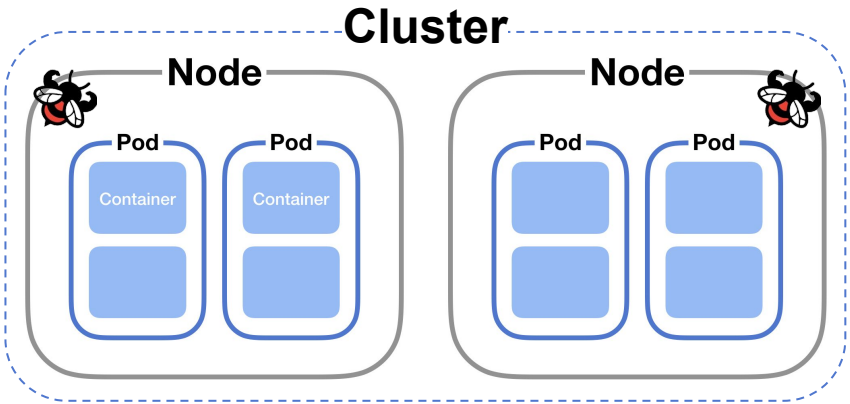
Extending kernel functionalities for security/observability



Extending kernel functionalities for security/observability



Extending kernel functionalities for security/observability



- **Security** - check unexpected behaviour, react, raising alerts
- **Observability** - generation of visibility events and the collection and in-kernel aggregation of custom metrics based on a broad range of potential sources

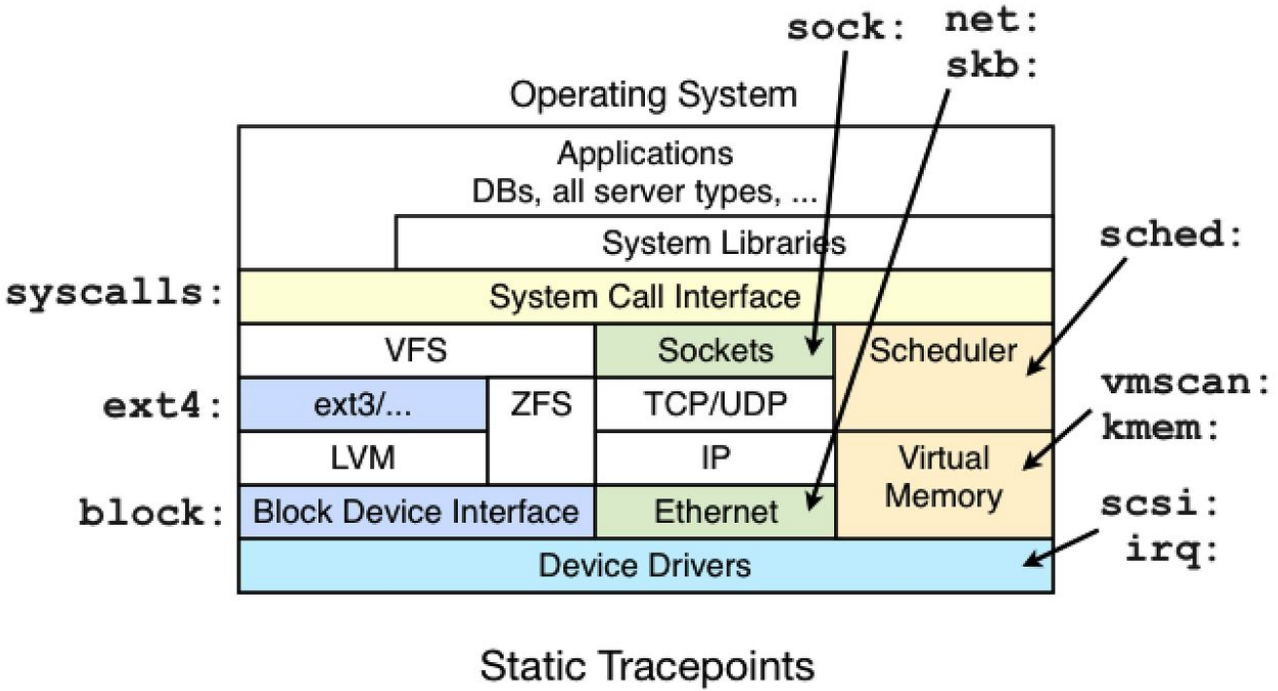
eBPF hooks

	Static	Dynamic	Kernel tracing	Userland Tracing
Tracepoints				
Kprobes				
Uprobes				
USDT				

Kernel Tracepoints

- Pre-defined hooks in kernel for custom tracing
- Stable across kernel versions
- Used for debugging, performance analysis, real-time monitoring
- Mount debugfs
  - `sudo mount -t debugfs nodev /sys/kernel/debug`

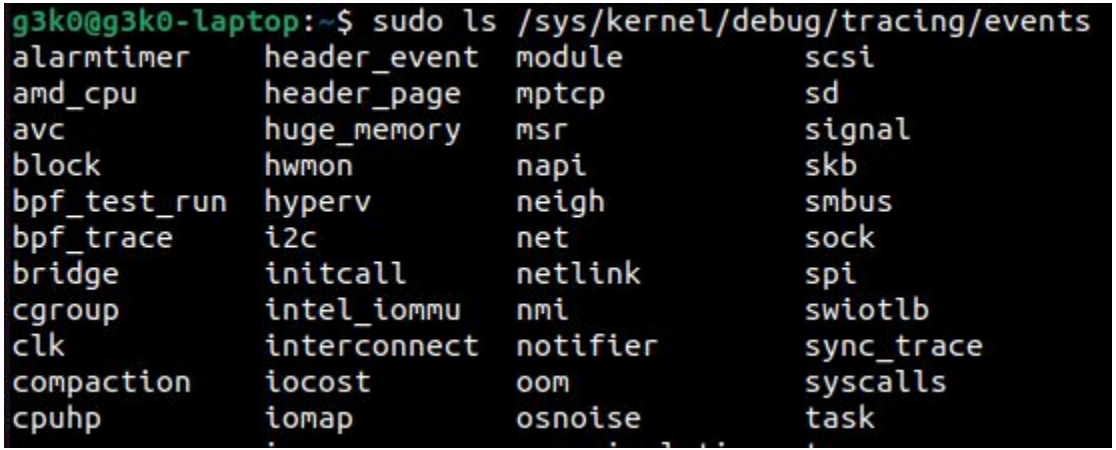
Tracepoints are located everywhere



Static Tracepoints

List of available tracepoints

```
sudo ls /sys/kernel/debug/tracing/events
```



Tracing syscalls

```
sudo ls /sys/kernel/debug/tracing/events/syscalls
```

```
g3k0@g3k0-laptop:~$ sudo ls /sys/kernel/debug/tracing/events/syscalls
enable                          sys_enter_writev
filter                          sys_exit_accept
sys_enter_accept                sys_exit_accept4
sys_enter_accept4               sys_exit_access
sys_enter_access                sys_exit_acct
sys_enter_acct                  sys_exit_add_key
sys_enter_add_key               sys_exit_adjtimex
sys_enter_adjtimex              sys_exit_alarm
sys_enter_alarm                 sys_exit_arch_prctl
sys_enter_arch_prctl            sys_exit_bind
sys_enter_bind                  sys_exit_bpf
sys_enter_bpf                   sys_exit_brk
```

Interacting with debugfs

- Inside each we have special purpose files: `enable`, `format`, `filter`
- Enable 'sched/sched_switch' tracepoint
  - `echo 1 | sudo tee /sys/kernel/debug/tracing/events/sched/sched_switch/enable`
- Only trace when next process PID is 1000
  - `echo 'next_pid == 1000' | sudo tee /sys/kernel/debug/tracing/events/sched/sched_switch/filter`

Tracepoint parameters

`sudo cat /sys/kernel/debug/tracing/events/syscalls/sys_enter_openat/format`

```
g3k0@g3k0-laptop:~$ sudo cat /sys/kernel/debug/tracing/events/syscalls/sys_enter_openat/format
name: sys_enter_openat
ID: 690
format:
        field:unsigned short common_type;       offset:0;       size:2; signed:0;
        field:unsigned char common_flags;       offset:2;       size:1; signed:0;
        field:unsigned char common_preempt_count;       offset:3;       size:1; signed:0;
        field:int common_pid;   offset:4;       size:4; signed:1;

        field:int   syscall_nr; offset:8;       size:4; signed:1;
        field:int dfd;  offset:16;      size:8; signed:0;
        field:const char * filename;    offset:24;      size:8; signed:0;
        field:int flags;        offset:32;      size:8; signed:0;
        field:umode_t mode;     offset:40;      size:8; signed:0;

print fmt: "dfd: 0x%08lx, filename: 0x%08lx, flags: 0x%08lx, mode: 0x%08lx", ((unsigned long)(R
EC->dfd)), ((unsigned long)(REC->filename)), ((unsigned long)(REC->flags)), ((unsigned long)(RE
C->mode))
```

Tracepoint parameters

```
sudo cat /sys/kernel/debug/tracing/events/syscalls/sys_enter_openat/format
```

```
g3k0@g3k0-laptop:~$ sudo ca
name: sys_enter_openat
ID: 690
format:
        field:unsigned shor
        field:unsigned char
        field:unsigned char
        field:int common_pi

        field:int   syscall
        field:int dfd;  offset:16;      size:8; signed:0;
        field:const char * filename;    offset:24;      size:8; signed:0;
        field:int flags;        offset:32;      size:8; signed:0;
        field:umode_t mode;     offset:40;      size:8; signed:0;

print fmt: "dfd: 0x%08lx, filename: 0x%08lx, flags: 0x%08lx, mode: 0x%08lx", ((unsigned long)(R
EC->dfd)), ((unsigned long)(REC->filename)), ((unsigned long)(REC->flags)), ((unsigned long)(RE
C->mode))
```

/ fs / open.c

```
1428  }
1429
1430  SYSCALL_DEFINE4(openat, int, dfd, const char __user *, filename, int, flags,
1431                  umode_t, mode)
1432  {
1433          if (force_o_largefile())
1434                  flags |= O_LARGEFILE;
1435          return do_sys_open(dfd, filename, flags, mode);
1436  }
```

Tracepoint hands on

- For security reasons we want to block access to `/etc/passwd`
- Applications use `openat` syscall to open a file
- eBPF program attached to the `sys_enter_openat` tracepoint

libbpf-bootstrap

- Scaffolding playground for eBPF development
- Contains examples with many different hooks
- Bundled with libbpf and bpftools (for x86-64 architecture only)
- Rely on kernel to be built with BTF (BPF Type Format) type information
  - `CONFIG_DEBUG_INFO_BTF=y` Kconfig
  - Metadata format to encode debug info related to BPF program/maps
  - See https://www.kernel.org/doc/html/latest/bpf/btf.html for more information about BTF
  - Some major Linux distributions come with kernel BTF already built in
  - List here https://github.com/libbpf/libbpf?tab=readme-ov-file#bpf-co-re-compile-once--run-everywhere

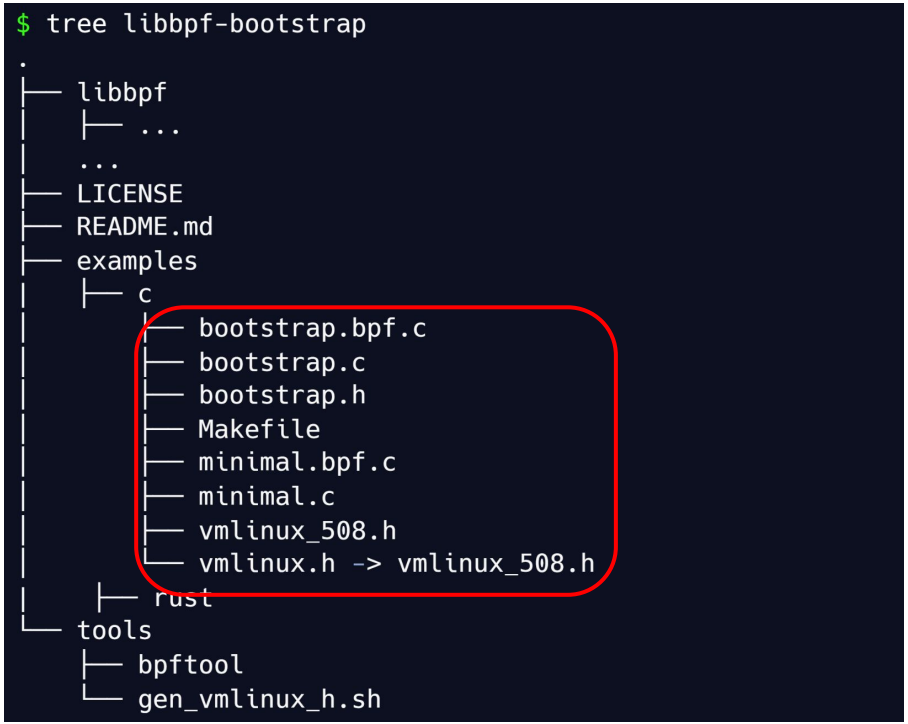
libbpf-bootstrap - setup

- Dependencies install (Ubuntu)
  - `sudo apt-get update -y`
  - `sudo apt-get install -y make gcc clang libelf1 libelf-dev zlib1g-dev`
- Clone the repository and submodules
  - `git clone --recurse-submodules https://github.com/libbpf/libbpf-bootstrap.git`
- For convenience here's a repo with Docker+scripts
  - https://drive.google.com/drive/folders/1GECYcQnQBzJdlLQKVdJA5K7zARkRBXMM?usp=sharing

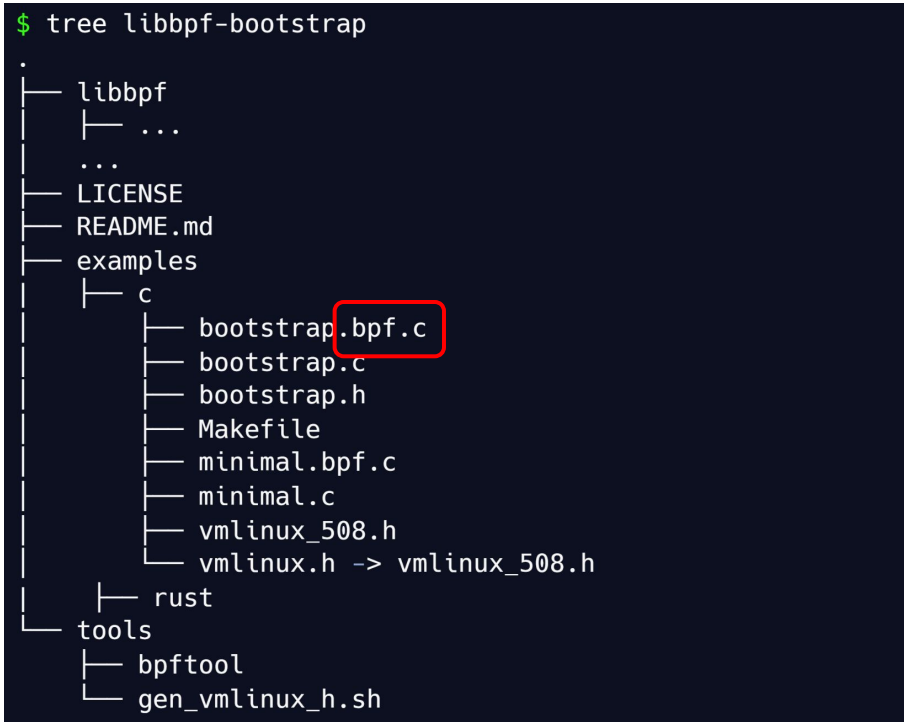
libbpf-bootstrap - structure

```
$ tree libbpf-bootstrap
.
├── libbpf
│   ├── ...
│   ...
├── LICENSE
├── README.md
├── examples
│   ├── c
│       ├── bootstrap.bpf.c
│       ├── bootstrap.c
│       ├── bootstrap.h
│       ├── Makefile
│       ├── minimal.bpf.c
│       ├── minimal.c
│       ├── vmlinux_508.h
│       └── vmlinux.h -> vmlinux_508.h
│    ├── rust
└── tools
    ├── bpftool
    └── gen_vmlinux_h.sh
```

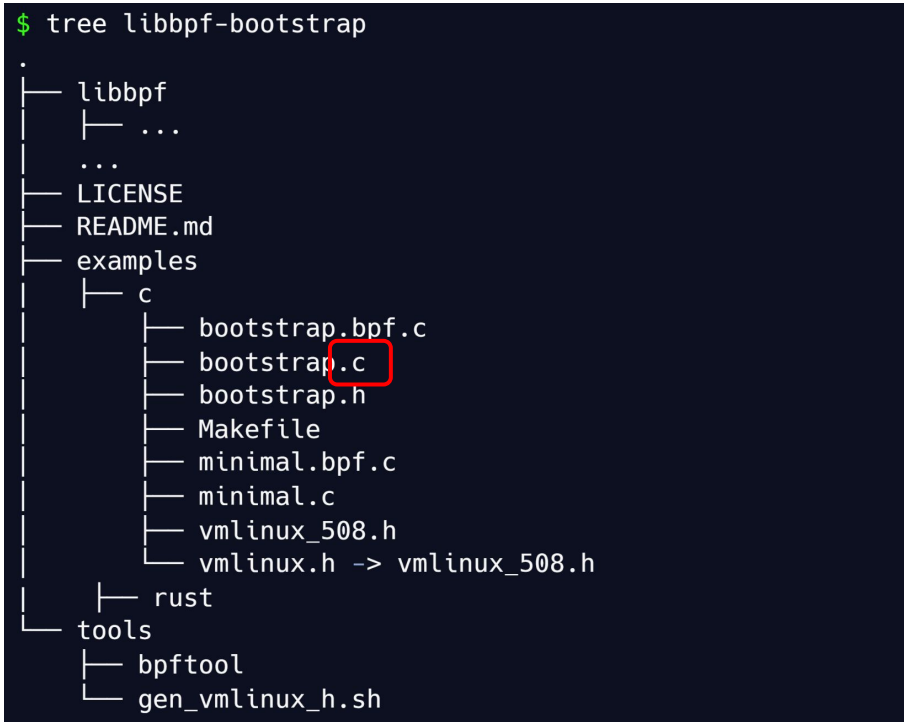
libbpf-bootstrap - structure



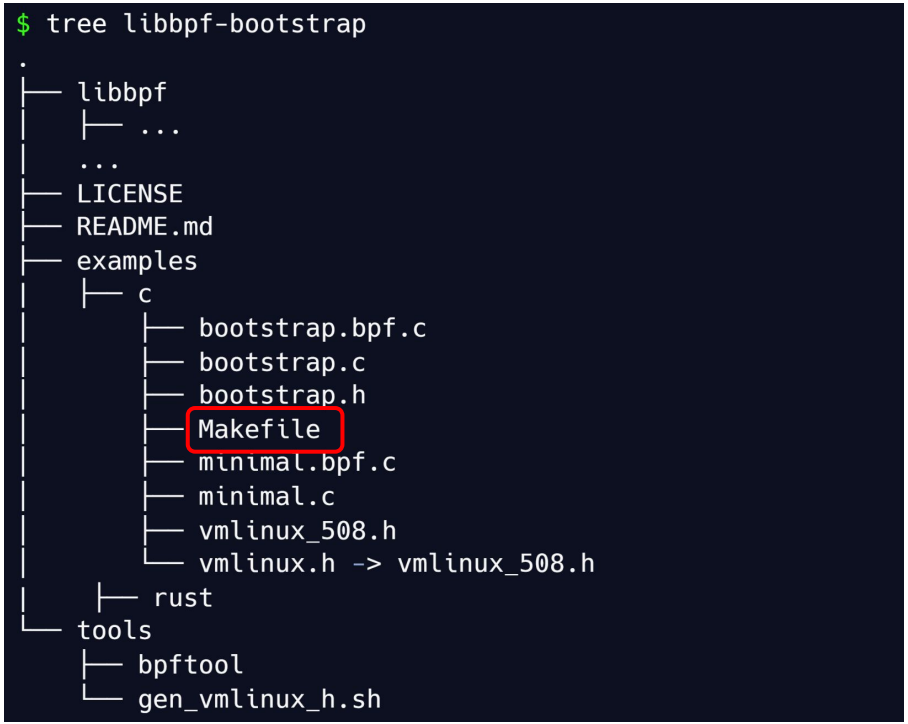
libbpf-bootstrap - structure



libbpf-bootstrap - structure



libbpf-bootstrap - structure



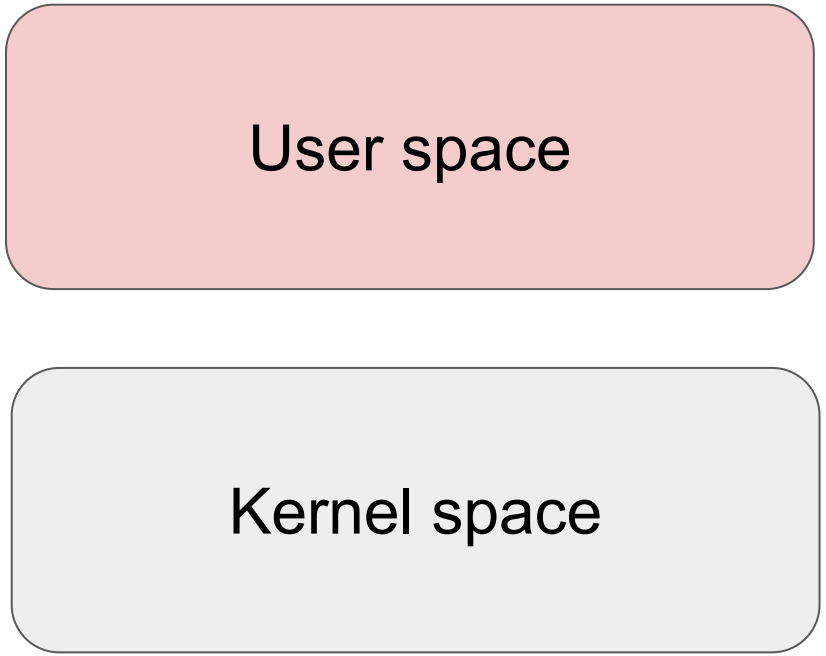
Makefile

```
# SPDX-License-Identifier: (LGPL-2.1 OR BSD-2-Clause)
OUTPUT := .output
CLANG ?= clang
LIBBPF_SRC := $(abspath ../../libbpf/src)
BPFTOOL_SRC := $(abspath ../../bpftool/src)
LIBBPF_OBJ := $(abspath $(OUTPUT)/libbpf.a)
BPFTOOL_OUTPUT ?= $(abspath $(OUTPUT)/bpftool)
BPFTOOL ?= $(BPFTOOL_OUTPUT)/bootstrap/bpftool
LIBBLAZESYM_SRC := $(abspath ../../blazesym/)
LIBBLAZESYM_INC := $(abspath $(LIBBLAZESYM_SRC)/capi/include)
LIBBLAZESYM_OBJ := $(abspath $(OUTPUT)/libblazesym_c.a)
ARCH ?= $(shell uname -m | sed 's/x86_64/x86/' \
                         | sed 's/arm.*/arm/' \
                         | sed 's/aarch64/arm64/' \
                         | sed 's/ppc64le/powerpc/' \
                         | sed 's/mips.*/mips/' \
                         | sed 's/riscv64/riscv/' \
                         | sed 's/loongarch64/loongarch/')
VMLINUX := ../../vmlinux/$(ARCH)/vmlinux.h
# Use our own libbpf API headers and Linux UAPI headers distributed with
# libbpf to avoid dependency on system-wide headers, which could be missing or
# outdated
INCLUDES := -I$(OUTPUT) -I../../libbpf/include/uapi -I$(dir $(VMLINUX)) -I$(LIBBLAZESYM_INC)
CFLAGS := -g -Wall
ALL_LDFLAGS := $(LDFLAGS) $(EXTRA_LDFLAGS)

APPS = minimal minimal_legacy bootstrap uprobe kprobe fentry usdt sockfilter tc ksyscall task_iter lsm yourprogram
```

Openat tracepoint programs

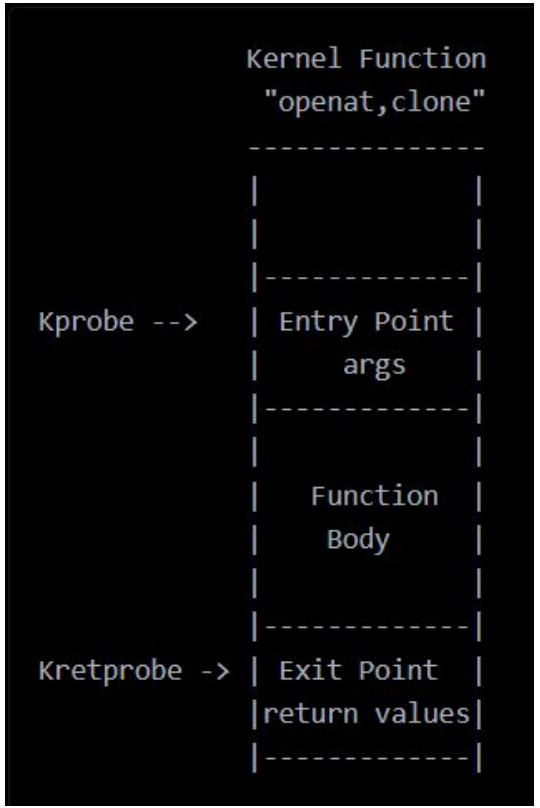
- `ebpf_day_openat.c`
  - Loads eBPF program (`ebpf_day_openat.bpf`)
  - Attach it to the tracepoint
  - Wait for termination
  - De-attach program
- `ebpf_day_openat.bpf.c`
  - Actual eBPF code triggered by the tracepoint
  - Controls what file is trying to be open
  - If is `/etc/passwd` react!



Let’s see it in action

Kernel Probes (Kprobes)

- Breakpoints in the kernel code for inspection or modification of kernel behavior at runtime
- Ability to insert probes on almost any kernel symbol at runtime
  - the symbol has to be exported by the kernel (`EXPORT_SYMBOL` macro)
- Handlers can gather/modify function data



Kprobe hands on

- For observability reasons we want to track what files are deleted
- Applications use `unlink` syscall to open a file

Unlink syscall

/ fs / namei.c

```
4435            goto exit3;
4436    }
4437
4438    SYSCALL_DEFINE3(unlinkat, int, dfd, const char __user *, pathname, int, flag)
4439    {
4440            if ((flag & ~AT_REMOVEDIR) != 0)
4441                    return -EINVAL;
4442
4443            if (flag & AT_REMOVEDIR)
4444                    return do_rmdir(dfd, getname(pathname));
4445            return do_unlinkat(dfd, getname(pathname));
4446    }
4447
4448    SYSCALL_DEFINE1(unlink, const char __user *, pathname)
4449    {
4450            return do_unlinkat(AT_FDCWD, getname(pathname));
4451    }
```

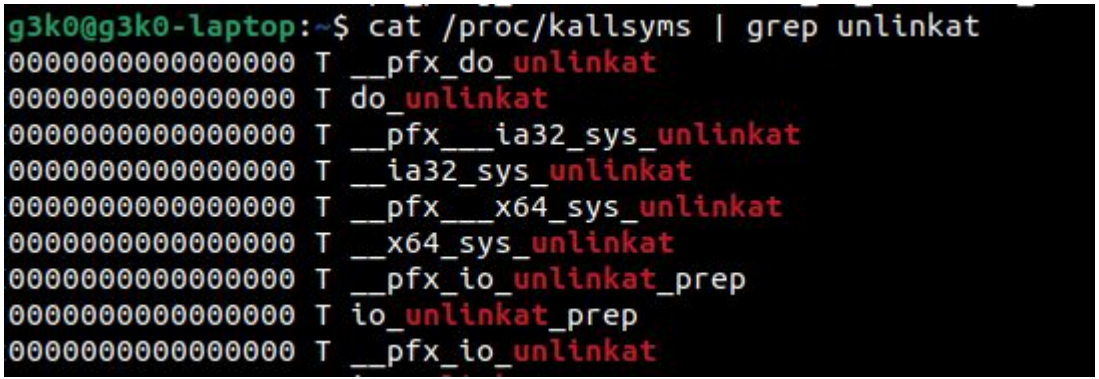
Unlink syscall

/ fs / namei.c

```
4435            goto exit3;
4436    }
4437
4438    SYSCALL_DEFINE3(unlinkat, int, dfd, const char __user *, pathname, int, flag)
4439    {
4440            if ((flag & ~AT_REMOVEDIR) != 0)
4441                    return -EINVAL;
4442
4443            if (flag & AT_REMOVEDIR)
4444                    return do_rmdir(dfd, getname(pathname));
4445            return do_unlinkat(dfd, getname(pathname));
4446    }
4447
4448    SYSCALL_DEFINE1(unlink, const char __user *, pathname)
4449    {
4450            return do_unlinkat(AT_FDCWD, getname(pathname));
4451    }
```

Kprobe for do_unlinkat

- Available tracepoint in `/proc/kallsyms` file



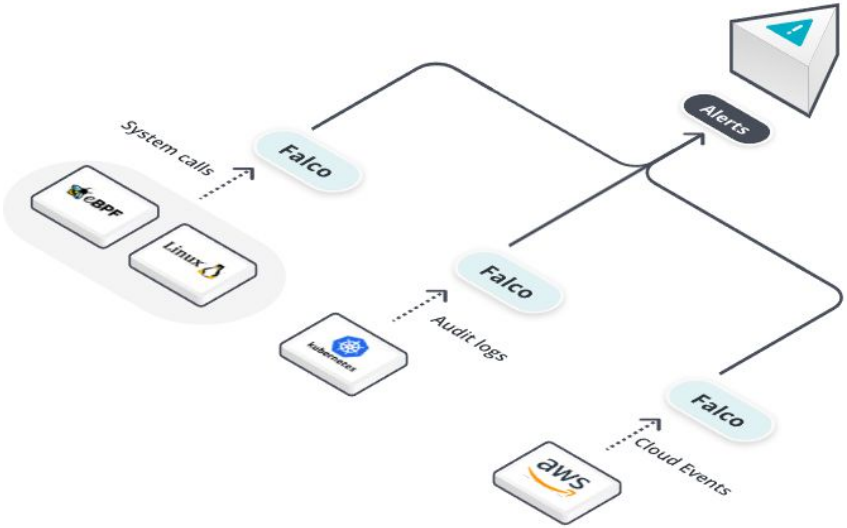
Let’s see it in action

Real world examples - Tetragon

- Real time eBPF-based Security Observability and Runtime Enforcement
- Detect and to react to security-significant events
- Cilium's component
- Cilium is used by many big players https://cilium.io/adopters/

Real world examples - Falco

- Real time detection of unexpected behavior, configuration changes, attacks
- Custom rules on kernel events enriched with containers metadata
- Notable users like AWS, IBM, Red Hat
  - https://github.com/falcosecurity/falco/blob/master/ADOPTERS.md



Useful resources

- https://docs.cilium.io/en/latest/bpf/
- https://eunomia.dev/tutorials/
- https://douglasmakey.medium.com/beyond-observability-modifying-syscall-behavior-with-ebpf-my-precious-secret-files-62aa0e3c9860
- https://nakryiko.com/posts/bcc-to-libbpf-howto-guide/#bpf-skeleton-and-bpf-app-lifecycle
- https://nakryiko.com/posts/libbpf-bootstrap/

Thanks for the attention!

📧 giacomo.belocchi@uniroma2.it