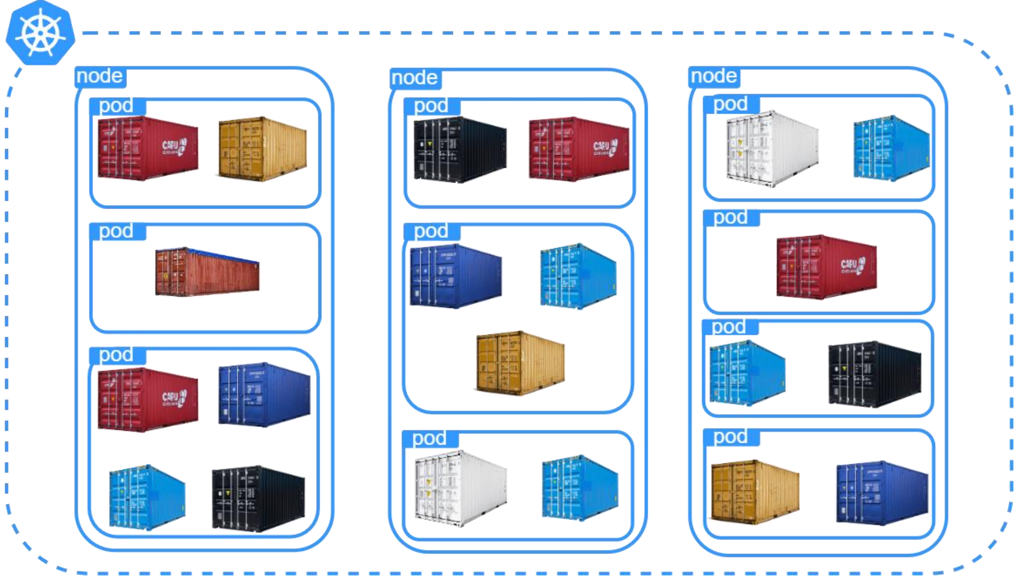
Beyond the Surface: Deep Dive into Kernel Observability with eBPF

Angelo Tulumello

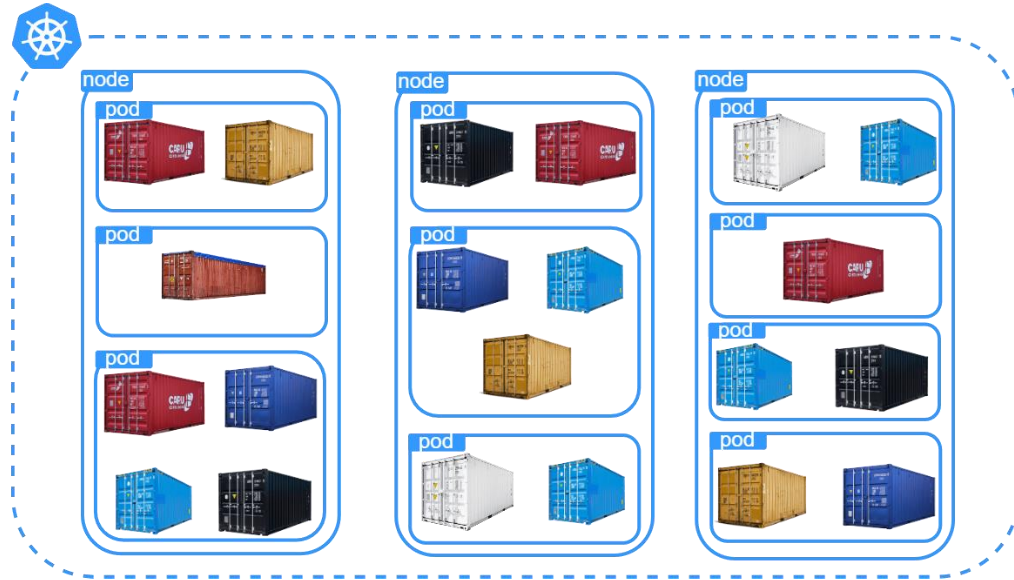
PhD School @ TMA '24 - May 21st, Dresden

Special thanks to Giacomo Belocchi for the material

How to monitor what happens in the cluster?

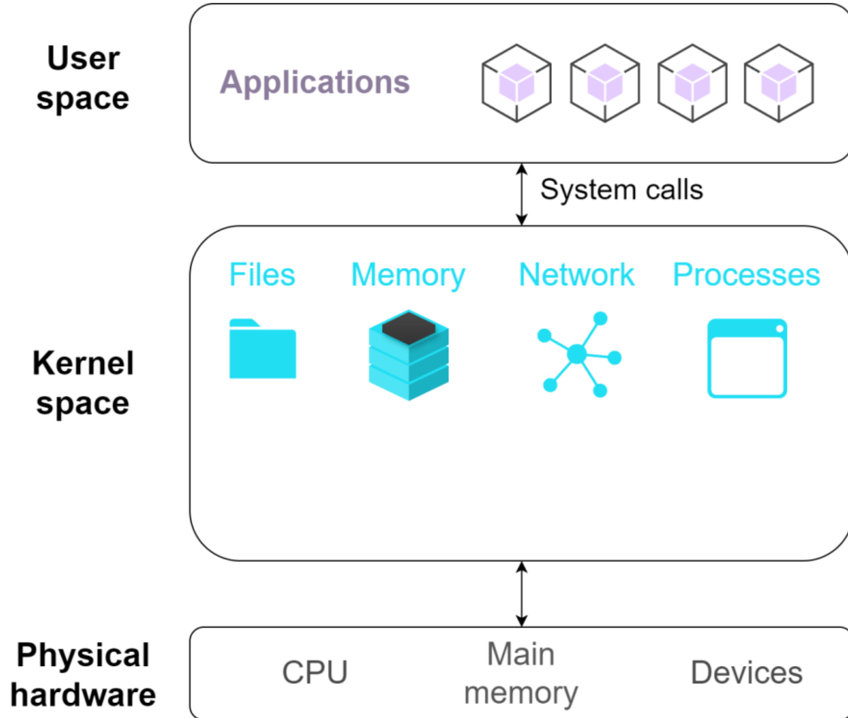


How to monitor what happens in the cluster?



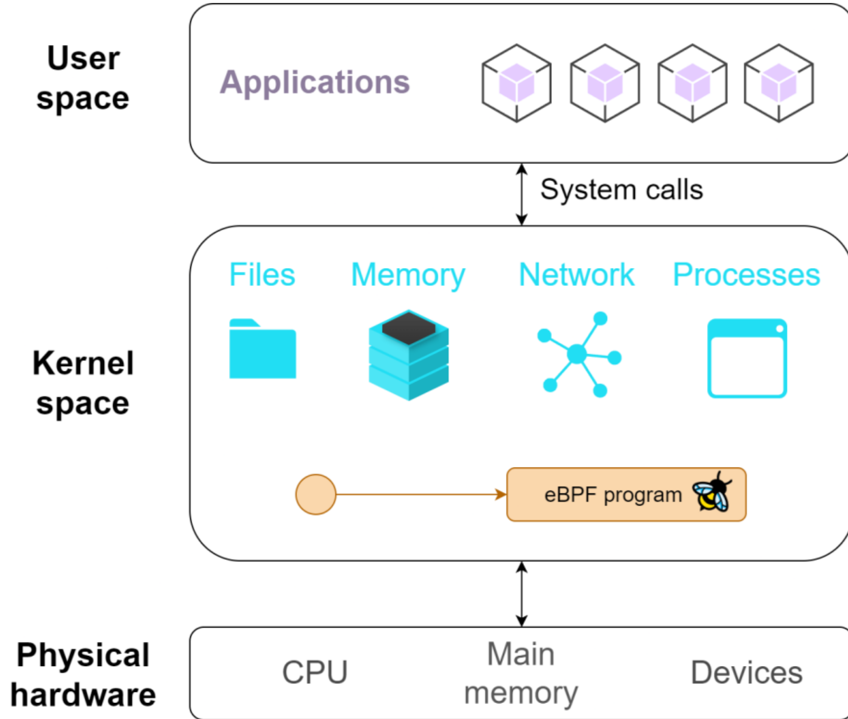
What if a Kubernetes administrator want to observe what happens? 🤯

System Architecture



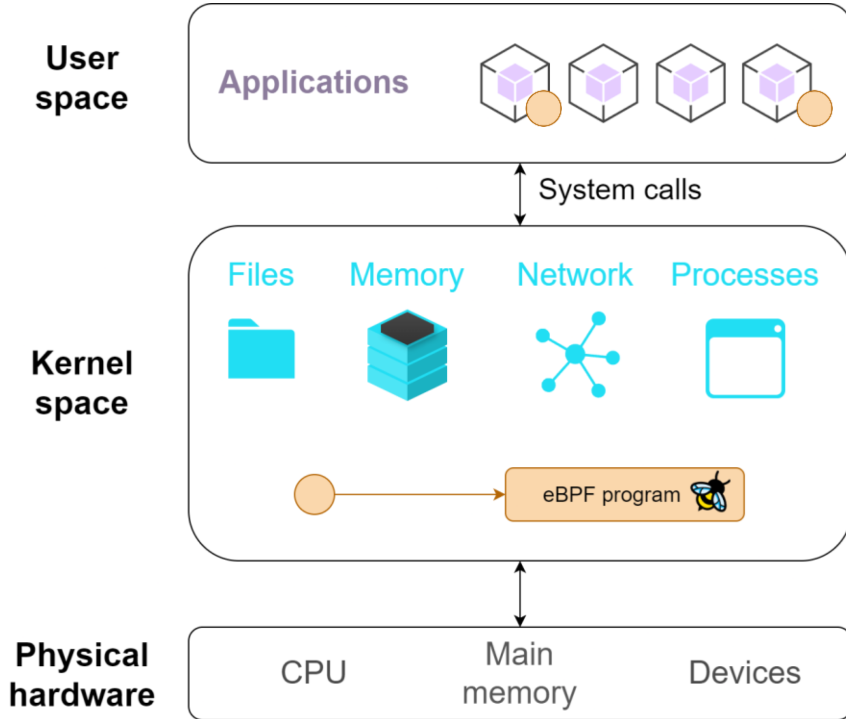
- User space where applications run
- Applications can't directly access hardware resources
- Applications use the kernel making syscalls
- File read/write, memory accesses, etc.
  - ALL goes through the kernel

Kernel - eBPF to the rescue



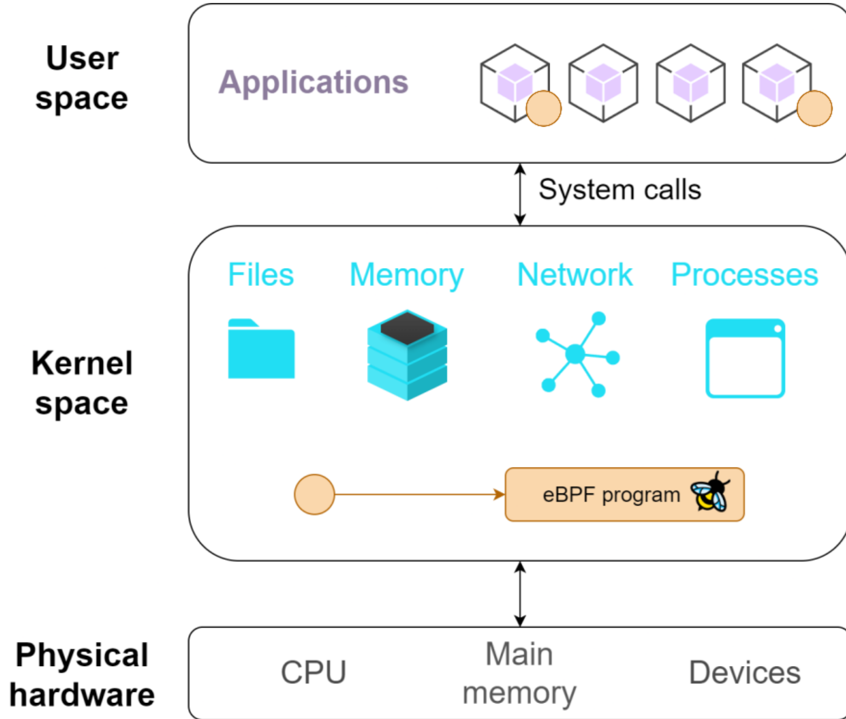
- Hooks inside the kernel

Kernel - eBPF to the rescue



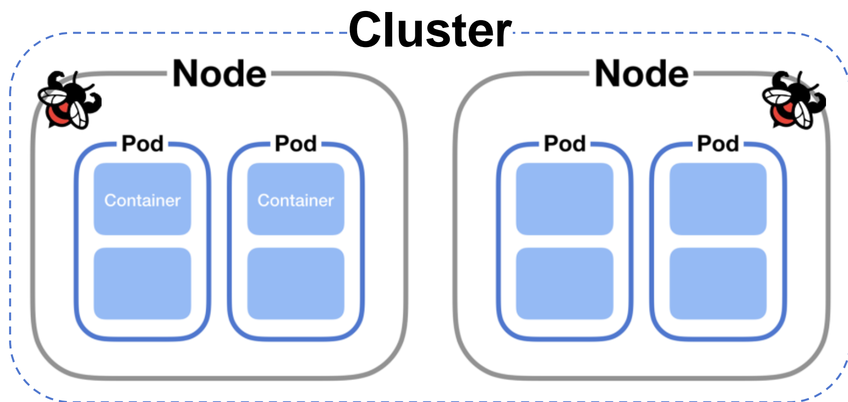
- Hooks inside the kernel
  - Or inside user space applications

Kernel - eBPF to the rescue

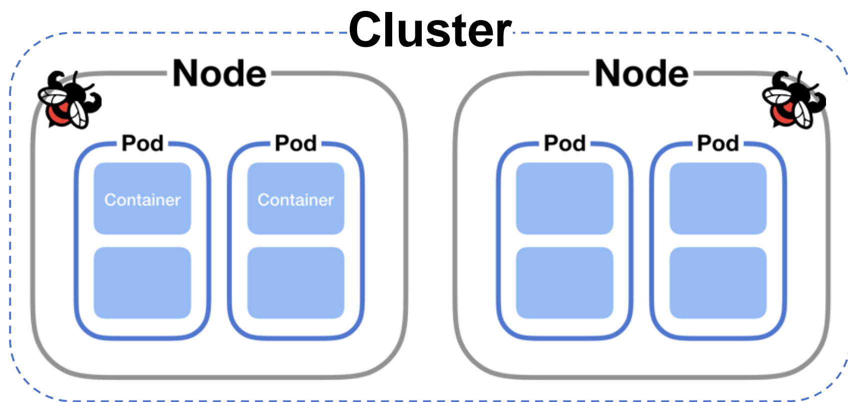


- Hooks inside the kernel
  - Or inside user space applications
- When execution reach the hook ⇒ eBPF program is invoked
- eBPF program can access data visible at the hook

Extending kernel functionalities for security/observability



Extending kernel functionalities for security and 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

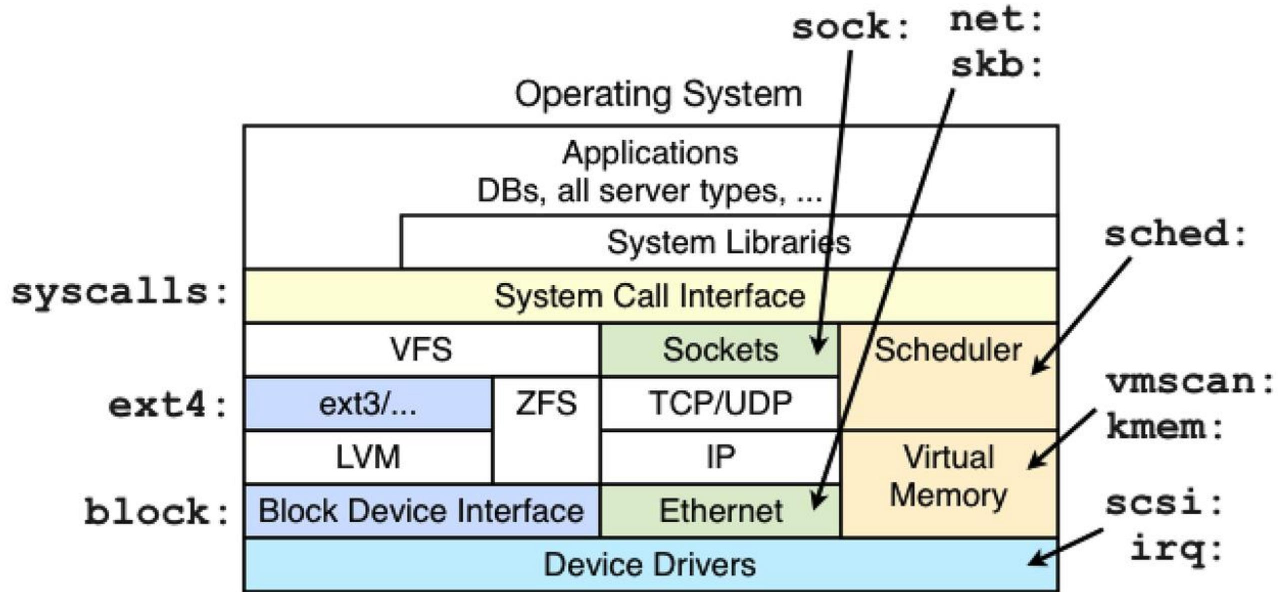
eBPF hooks

	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



List of available tracepoints

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

```
g3k0@g3k0-laptop:~$ sudo ls /sys/kernel/debug/tracing/events
alarmtimer     header_event  module        scsi
amd_cpu        header_page   mptcp         sd
avc            huge_memory   msr           signal
block          hwmon         napi          skb
bpf_test_run   hyperv        neigh         smbus
bpf_trace      i2c           net           sock
bridge         initcall      netlink       spi
cgroup         intel_iommu   nmi           swiotlb
clk            interconnect  notifier      sync_trace
compaction     iocost        oom           syscalls
cpuhp          iomap         osnoise       task
```

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

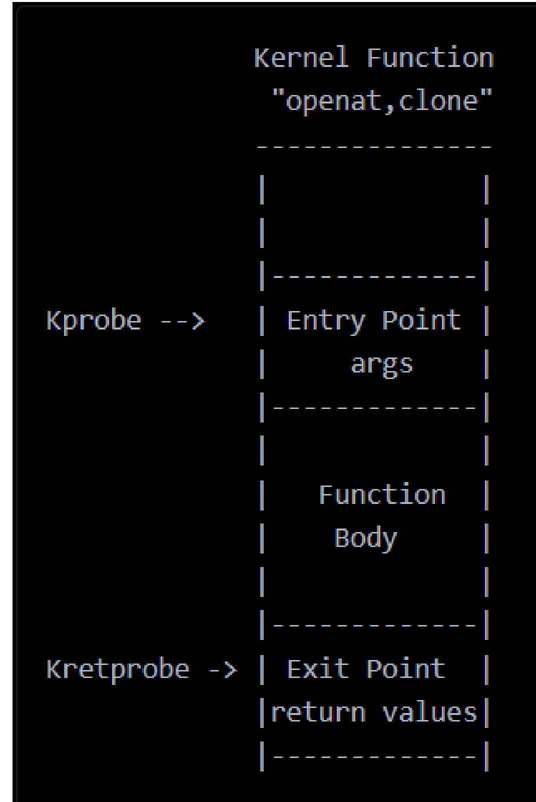
Openat tracepoint programs

- `ebpf_tutorial_tracepoint.c`
  - Loads eBPF program (`ebpf_day_tracepoint.bpf.c`)
  - Attach it to the tracepoint
  - Wait for termination
  - De-attach program
- `ebpf_tutorial_tracepoint.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 delete a file/directory

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    }
4452
```

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    }
4452
```

Kprobe for do_unlinkat

- Available kprobes in `/proc/kallsyms` file

```
g3k0@g3k0-laptop:~$ cat /proc/kallsyms | grep unlinkat
0000000000000000 T __pfx_do_unlinkat
0000000000000000 T do_unlinkat
0000000000000000 T __pfx___ia32_sys_unlinkat
0000000000000000 T __ia32_sys_unlinkat
0000000000000000 T __pfx___x64_sys_unlinkat
0000000000000000 T __x64_sys_unlinkat
0000000000000000 T __pfx_io_unlinkat_prep
0000000000000000 T io_unlinkat_prep
0000000000000000 T __pfx_io_unlinkat
```

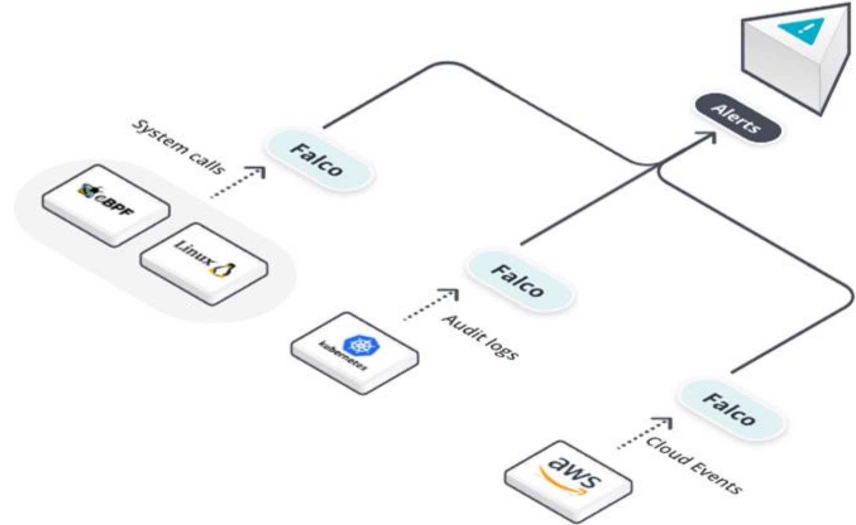
Let’s see it in action

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/

Challenge

Challenge: detect user switching to root

- We want to detect when an user switches to root privileges
- Hints:
  - Start from the tracepoint example
  - Which is the syscall used for changing user privileges?
    - Find the syscall and retrieve its format
    - Attach an eBPF program to it and print a log message when the event is detected
      - ***Only*** when an user switches to **root**!