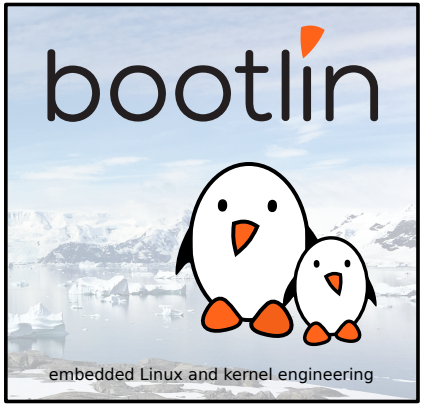
Modify your kernel at runtime with eBPF !

Alexis Lothoré
alexis.lothore@bootlin.com



Alexis Lothoré

- Embedded Linux engineer at Bootlin since 2023
  - Expertise in Embedded Linux
  - Development, consulting and training
  - Strong open-source focus
- Working on embedded systems since 2016
- BSP, device drivers, networking, wireless, CI, **eBPF**
  - Training courses
  - Kernel testing contributions
- Not really used to the mustache
- Lives in Toulouse, France
- alexis.lothore@bootlin.com

https://bootlin.com/company/staff/alexis-lothore/

You are probably already using eBPF !

```
$ apt install bpftool
$ bpftool prog list
```

Topics

- What is eBPF and why should we use it?
- eBPF core components
- Processes and tools to use eBPF
- Showtime!

eBPF: what, why, when

What is eBPF?

- "Extended BPF", evolution from Berkeley Packet Filter
- A "virtual machine" inside the kernel, allowing to run user programs directly in kernel space:
  - without having to modify/reboot the kernel
  - safely (can not make the kernel hang or crash)
  - almost anywhere in the kernel
- Event-driven
- Multiple elements: a dedicated ISA, kernel helpers, a pseudo-filesystem, a dedicated syscall, and offload mechanisms.

Why using eBPF

- Initially developped for networking use cases
- But is being used more and more for other topics: system monitoring, debugging, profiling, security...
- A few (simple) examples:
  - A program attached to a network interface performing some filtering and/or traffic redirection
  - A program attached to the `open` system call to monitor any access to a specific file on the system
  - A program attached to the `malloc` and `free` functions of your C library to create a custom memory leak detector
  - A custom scheduler ! (see `scheduler/sched-ext`)

Solutions and tools based on eBPF

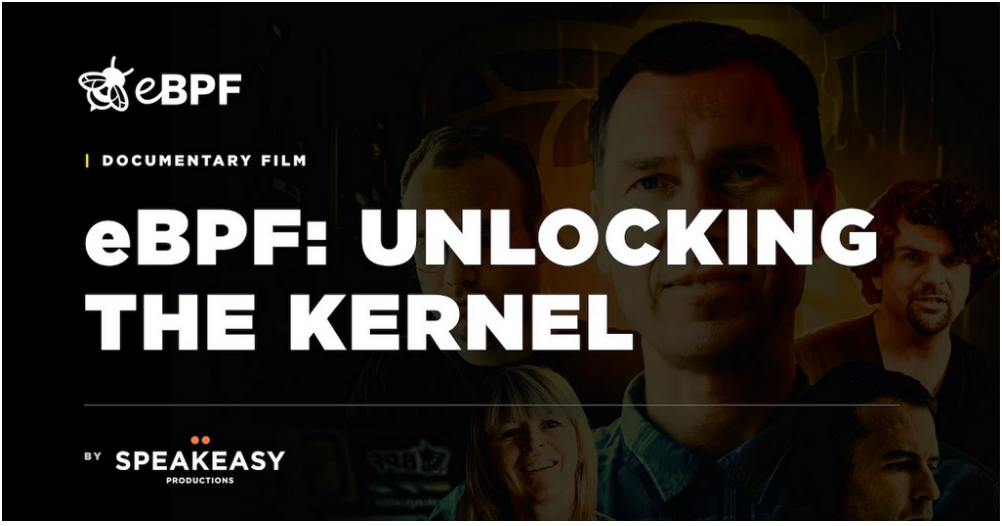
- Tracing, profiling: `BCC`, `bpftrace`, `pwru`
- Network infrastructure: `Cilium`, `Calico`
- Monitoring, Security: `Tetragon`, `Falco`
- More examples: see ebpf.io

This talk is not about a specific solution but about eBPF in general

eBPF: a few dates

- 2014:
  - eBPF interpreter added into the kernel (v3.15)
  - eBPF interpreter exposed to userspace (v3.17)
- 2015: eBPF extended to kprobes (v4.0)
- 2016: XDP, eXpress Data Path (v4.7)
- 2017: eBPF becomes a standalone subsystem
- 2018: BTF (BPF Type format) is added (v4.18)
- 2020: GCC is able to build eBPF programs
- 2021: creation of the eBPF Foundation
- 2024: eBPF ISA RFC published (RFC 9669)

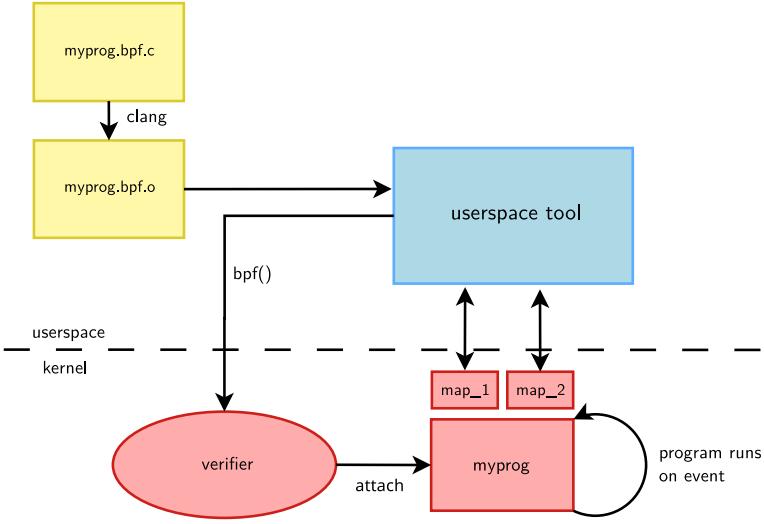
eBPF: a few dates



eBPF: Unlocking the kernel [Official Documentary]

eBPF components

eBPF program lifecycle



eBPF ISA

- eBPF defines its own virtual instruction set, bringing specific instructions and registers
- Those instructions are the one understood and run by the eBPF virtual machine in the kernel
- An eBPF program must then use those instructions to be able to run inside the kernel
- Standardization: see RFC9669

eBPF ISA

- A set of standard registers and a calling convention:

Register(s)	Convention
R0	function return value
R1-R5	function arguments
R6-R9	used by callee to save caller registers
R10	frame pointer

- A set of simple instructions:
  - Load/store instruction: `LD`, `LDX`, `ST`, `STX`,...
  - Arithmetic operations: `ADD`, `SUB`, `MUL`, `DIV`, `OR`...
  - Jump operations: `JEQ`, `JGT`, `JNE`, `CALL`, `EXIT`...
- Instructions are either interpreted at run time or translated to native instructions (JIT)

A simple program

```
0: (b7) r0 = 1
1: (79) r2 = *(u64 *)(r1 +8)
2: (79) r1 = *(u64 *)(r1 +0)
3: (bf) r3 = r1
4: (07) r3 += 14
5: (2d) if r3 > r2 goto pc+13
6: (71) r3 = *(u8 *)(r1 +12)
7: (71) r4 = *(u8 *)(r1 +13)
8: (67) r4 <<= 8
9: (4f) r4 |= r3
10: (b7) r0 = 2
11: (55) if r4 != 0x8 goto pc+7
12: (bf) r3 = r1
13: (07) r3 += 34
14: (b7) r0 = 1
15: (2d) if r3 > r2 goto pc+3
16: (71) r1 = *(u8 *)(r1 +23)
17: (15) if r1 == 0x1 goto pc+1
18: (b7) r0 = 2
19: (95) exit
```

A simple program: dropping ICMP packets

```
int drop_icmp(struct xdp_md * xdp):
; int drop_icmp(struct xdp_md *xdp)
0: (b7) r0 = 1
; void *data_end = (void *)(long)xdp->data_end;
1: (79) r2 = *(u64 *)(r1 +8)
; void *data = (void *)(long)xdp->data;
2: (79) r1 = *(u64 *)(r1 +0)
; if (eth + 1 > data_end)
3: (bf) r3 = r1
4: (07) r3 += 14
; if (eth + 1 > data_end)
5: (2d) if r3 > r2 goto pc+13
; if (eth->h_proto != bpf_htons(ETH_P_IP))
6: (71) r3 = *(u8 *)(r1 +12)
7: (71) r4 = *(u8 *)(r1 +13)
8: (67) r4 <<= 8
9: (4f) r4 |= r3
```

```
10: (b7) r0 = 2
; if (eth->h_proto != bpf_htons(ETH_P_IP))
11: (55) if r4 != 0x8 goto pc+7
; if (ip + 1 > data_end)
12: (bf) r3 = r1
13: (07) r3 += 34
14: (b7) r0 = 1
; if (ip + 1 > data_end)
15: (2d) if r3 > r2 goto pc+3
; if (ip->protocol != IPPROTO_ICMP)
16: (71) r1 = *(u8 *)(r1 +23)
;
17: (15) if r1 == 0x1 goto pc+1
18: (b7) r0 = 2
; \}
19: (95) exit
```

Program types and attach points

- eBPF programs are executed on events generated by the kernel
- There are different "types" of places in the kernel able to generate events:
  - a kernel-defined static tracepoint (see `/sys/kernel/tracing/available_events`)
  - an arbitrary kprobe
  - on security events (LSM)
  - when a packet is received in the kernel network stack
  - When a packet is received at network driver level
  - and a lot more, see `bpf_attach_type`

Program types and attach points

- A specific attach-point type can only be hooked with a specific program type, see `bpf_prog_type` and `bpf/libbpf/program_types`.
- The program type then defines the data passed to an eBPF program as input when it is invoked. For example:
  - A `BPF_PROG_TYPE_TRACEPOINT` program will receive a structure containing all data returned to userspace by the targeted tracepoint.
  - A `BPF_PROG_TYPE_SCHED_CLS` program (used to implement packets classifiers) will receive a `struct __sk_buff`, the kernel representation of a socket buffer.
  - A `BPF_PROG_TYPE_XDP` will receive a `struct xdp_md` context representing the raw packet received on the NIC
  - You can learn about the context passed to any program type by checking `include/linux/bpf_types.h`

eBPF program return value

- eBPF can be used to alter the kernel behavior at runtime.
- This is generally done thanks to the program return value, and interpretation depends on the program type:
  - XDP programs can return `XDP_PASS` to let a packet continue its journey in the kernel, or `XDP_DROP` to drop it
  - `BPF_MODIFY_RETURN` programs can replace the hooked function and provide an arbitrary return value
  - `LSM programs` can allow or refuse an operation (opening a file, loading a kernel module, modifying a process property...) by returning either `0` or `-EPERM`

A simple program: dropping ICMP packets

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

SEC("xdp")
int drop_icmp(struct xdp_md *xdp)
{
        [...]

        return XDP_DROP;

}
```

Program type	Attach Type	ELF section
BPF_PROG_TYPE_XDP	BPF_XDP_CPUMAP	xdp/cpumap
	BPF_XDP_DEVMAP	xdp/devmap
	BPF_XDP	xdp

The verifier

- Any program must be accepted by the verifier before being accepted into the kernel
- Prevents programs from breaking the kernel at runtime
- Works by analysing the "submitted" program and validating it against a set of rules
  - Must terminate
  - No infinite loop
  - No null pointer dereference
  - Must not access arbitrary memory
  - etc

The verifier

```
libbpf: prog 'drop_icmp': BPF program load failed: Permission denied
libbpf: prog 'drop_icmp': -- BEGIN PROG LOAD LOG --
0: R1=ctx() R10=fp0
; void *data = (void *)(long)xdp->data; @ simple_filter.bpf.c:12
0: (61) r2 = *(u32 *)(r1 +0)          ; R1=ctx() R2_w=pkt(r=0)
; if (eth->h_proto != bpf_htons(ETH_P_IP)) @ simple_filter.bpf.c:19
1: (71) r3 = *(u8 *)(r2 +13)
invalid access to packet, off=13 size=1, R2(id=0,off=13,r=0)
R2 offset is outside of the packet
processed 2 insns (limit 1000000) max_states_per_insn 0 total_states 0 peak_states 0 mark_read 0
-- END PROG LOAD LOG --
libbpf: prog 'drop_icmp': failed to load: -13
libbpf: failed to load object 'simple_filter.bpf.o'
Error: failed to load object file
```

eBPF maps

- Data structures manipulated by both eBPF programs and userspace programs
- Different types of maps depending on the use case
  - Generic types: `ARRAY`, `HASH`, `QUEUE`...
  - Map in map: `ARRAY_OF_MAPS`, `HASH_OF_MAPS`
  - For large amounts of data: `PERF_EVENT_ARRAY`, `RINGBUF`...
  - For packets steering: `DEVMAP`, `CPUMAP`, `SOCKMAP`...
  - Storage: `CGROUP_STORAGE`, `TASK_STORAGE`, `SK_STORAGE`...
  - and many more, check `bpf_map_type` for the exact list

```
struct {
        __uint{type, BTF_MAP_TYPE_ARRAY};
        __type{key, int};
        __type{value, int}
        __uint{max_entries, 16};
} drop_count SEC{".maps"}
```

eBPF helpers

- Set of stable kernel functions usable in eBPF programs
  - `bpf_trace_printk`: Emit a log to the trace buffer
  - `bpf_map_{lookup,update,delete}_elem`: Manipulate maps
  - `bpf_get_current_pid_tgid`: Get current Process ID and Thread group ID
  - `bpf_get_current_uid_gid`: Ger current User ID and Group ID
  - `bpf_get_current_comm`: Get the name of the executable running in the current task
  - `bpf_get_current_task`: Get the current `struct task_struct`
  - Many other helpers are available, see `man 7 bpf-helpers`

Writing eBPF programs: a simple example

```c
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/in.h>
#include <bpf/bpf_endian.h>
#include <bpf/bpf_helpers.h>

struct {
        __uint(type, BPF_MAP_TYPE_ARRAY);
        __type(key, int);
        __type(value, int);
        __uint(max_entries, 1);
} drop_count SEC(".maps");

SEC("xdp")
int drop_icmp(struct xdp_md *xdp)
{
        void *data_end = (void *)(long)xdp->data_end;
        void *data = (void *)(long)xdp->data;
        struct ethhdr *eth = data;
        struct iphdr *ip;
        int *count;
        int key=0;

        if (eth + 1 > data_end)
                return XDP_DROP;

        [...]
```

Writing eBPF programs

```
        [...]

        if (eth->h_proto != bpf_htons(ETH_P_IP))
                return XDP_PASS;

        ip = data+sizeof(struct ethhdr);
        if (ip + 1 > data_end)
                return XDP_DROP;

        if (ip->protocol != IPPROTO_ICMP)
                return XDP_PASS;

        char fmt[] = "Dropping ICMP packet !";
        bpf_trace_printk(fmt, sizeof(fmt));

        count = bpf_map_lookup_elem(&drop_count, &key);
        if (count)
                *count+=1;

        return XDP_DROP;
}

char __license[] SEC("license") = "GPL";
```

Processes and tools

Building eBPF programs

- We need of a compiler able to translate C programs into eBPF instructions.
- As of today, both LLVM (clang) and GCC are capable.

```
clang -target bpf -O2 -g -c my_program.bpf.c -o my_program.bpf.o
```

or

```
bpf-unknown-gcc -O2 -g -c my_program.bpf.c -o my_program.bpf.o
```

Loading eBPF programs

- Multiple ways of loading a program:
  - Write our own loader and use the `bpf()` syscall, see `man 2 bpf` syscall
  - Use `bpftool` and/or `iproute2`
  - Write our custom loader but with higher level languages/libraries

Loading eBPF programs: bpf() syscall

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

- A single syscall for all eBPF operations, split into subcommands:
  - BPF_PROG_LOAD
  - BPF_MAP_CREATE
  - BPF_MAP_LOOKUP_ELEM
  - BPF_MAP_UPDATE_ELEM
  - BPF_MAP_DELETE_ELEM
  - BPF_BTF_LOAD
  - BPF_LINK_CREATE
  - BPF_PROG_TEST_RUN
  - ...
- Most subcommands work on file descriptors (pointing to a program, a map, btf data...)

Loading eBPF programs: bpftool

- The swiss army knife of eBPF development/management/debugging
- Developed in the kernel source tree, see `tools/bpf/bpftool/`
- A single commandline tool to manipulate programs, maps, links, btf data, etc...

```
$ bpftool help
Usage: bpftool [OPTIONS] OBJECT { COMMAND | help }
       bpftool batch file FILE
       bpftool version

       OBJECT := { prog | map | link | cgroup | perf | net | feature | btf | gen | struct_ops | iter }
       OPTIONS := { {-j|--json} [{-p|--pretty}] | {-d|--debug} |
                    {-V|--version} }
```

bpftool

- List loaded programs

```
$ bpftool prog
348: tracepoint  name sched_tracer  tag 3051de4551f07909  gpl
loaded_at 2024-08-06T15:43:11+0200  uid 0
xlated 376B  jited 215B  memlock 4096B  map_ids 146,148
btf_id 545
```

- Load (and possibly attach) a program

```
$ mkdir /sys/fs/bpf/myprog
$ bpftool prog loadall trace_execve.bpf.o /sys/fs/bpf/myprog [loadall]
```

- Unload a program

```
$ rm -rf /sys/fs/bpf/myprog
```

bpftool

- Dump a loaded program

```
$ bpftool prog dump xlated id 348
int sched_tracer(struct sched_switch_args * ctx):
; int sched_tracer(struct sched_switch_args *ctx)
  0: (bf) r4 = r1
  1: (b7) r1 = 0
; __u32 key = 0;
  2: (63) *(u32 *)(r10 -4) = r1
; char fmt[] = "Old task was %s, new task is %s\n";
  3: (73) *(u8 *)(r10 -8) = r1
  4: (18) r1 = 0xa7325207369206b
  6: (7b) *(u64 *)(r10 -16) = r1
  7: (18) r1 = 0x7361742077656e20
[...]
```

- Dump eBPF program logs

```
$ bpftool prog tracelog
kworker/u80:0-11  [013] d..41  1796.003605: bpf_trace_printk: Old task was kworker/u80:0, new task is swapper/13
<idle>-0          [013] d..41  1796.003609: bpf_trace_printk: Old task was swapper/13, new task is kworker/u80:0
sudo-18640        [010] d..41  1796.003613: bpf_trace_printk: Old task was sudo, new task is swapper/10
<idle>-0          [010] d..41  1796.003617: bpf_trace_printk: Old task was swapper/10, new task is sudo
[...]
```

bpftool

- List created maps

```
$ bpftool map
80: array  name counter_map  flags 0x0
    key 4B  value 8B  max_entries 1  memlock 256B
    btf_id 421
82: array  name .rodata.str1.1  flags 0x80
    key 4B  value 33B  max_entries 1  memlock 288B
    frozen
96: array  name libbpf_global  flags 0x0
    key 4B  value 32B  max_entries 1  memlock 280B
[...]
```

- Show a map content

```
$ sudo bpftool map dump id 80
[{
  "key": 0,
  "value": 4877514
  }
]
```

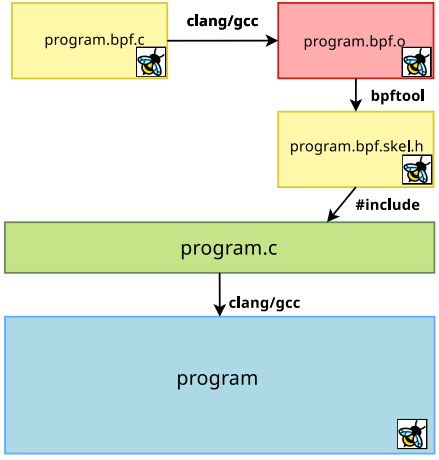
Manipulating attached programs

- We need to interact with attached programs:
  - Retrieve logs
  - Read maps
  - Modify maps
- Keeping using `bpftool` is unhandy, we may rather prefer to develop our own programs dedicated to our eBPF-based feature
- Contrarily to eBPF programs, we can use a wider variety of languages/frameworks to write those:
  - C: `libbpf`
  - Go: `ebpf-go`, `libbpfgo`
  - Rust: `libbpf-rs`, `redbpf`, `aya`
    - The eBPF program can also be written in Rust

libbpf

- easy-to-use C library to ease eBPF tooling writing:
  - eBPF program loader
  - high and low levels APIs for userspace
  - Wrapper APIs to call bpf helpers in eBPF programs
  - CO-RE
  - Supports bpftool skeletons
- Sources are maintained in the kernel source tree, see `tools/lib/bpf/`

libbpf and bpftool




```
$ bpftool gen skeleton simple_filter.bpf.o name simple_filter > simple_filter.bpf.skel.h
$ gcc simple_filter.c -o simple_filter. -lbpf
```

Writing our userspace program

```c
#include <bpf/libbpf.h>
#include <unistd.h>
#include <signal.h>
#include <net/if.h>
#include "simple_filter.bpf.skel.h"

static bool quit = false;

void sigint(int unused)
{
        quit = true;
}

int main(int argc, char *argv[])
{
        int ifindex, key=0, count, ret, prog_fd;
        struct simple_filter *skel;

        signal(SIGINT, sigint);

        [...]
```

Writing our userspace program

```
        [...]

        skel = simple_filter__open_and_load();
        if (!skel)
                exit(EXIT_FAILURE);

        prog_fd = bpf_program__fd(skel->progs.drop_icmp);
        ifindex = if_nametoindex("lo");
        ret = bpf_xdp_attach(ifindex, prog_fd, 0, NULL);

        while(!quit){
                ret = bpf_map__lookup_elem(skel->maps.drop_count, &key, sizeof(int),
                                &count, sizeof(int), 0);
                if (!ret)
                        fprintf(stdout, "%d packets dropped\n", count);
                sleep(2);
        }
        bpf_xdp_detach(ifindex, 0, NULL);
        simple_filter__destroy(skel);

        return 0;
}
```

Showtime

Resources

- https://github.com/Tropicao/ebpf_simple_filter.git
- The official eBPF documentaion
- Bootlin "Debugging, Tracing, and Profiling" training
- Kernel tests: tools/testing/selftests/bpf/
- Learning eBPF, Liz Rice

Questions?

Alexis Lothoré

alexis.lothore@bootlin.com

Bootlin is hiring!
We also have internships available for 2025
More on bootlin.com and at our booth