HOWTO: design kernel extensions with BPF struct_ops

Alexei Starovoitov

KERNEL
ENGINEERING
Meta

Motivation

- The kernel is not a place to implement a policy
- Examples of policies
    - TCP congestion control
    - OOM detection and selection
    - packet scheduling in networking
    - task selection and placement
    - security policies
    - various heuristics
- Solution: implement a policy as a kernel module or as a BPF program

Motivation

- The kernel have no room for uapi mistakes
- We make mistakes all the time

Old way of extending kernel with BPF

- Add new program type to uapi/bpf.h for every use case
    - Grew to 32 types over the years
- Sprinkle hooks in the kernel
- Add specific helpers

- uapi mistakes are forever
    - 6 out of 32 have zero users :(
- root cause
    - each program type means unique and specific bpf program context as only input argument

Lesson learned

- No new prog types
    - Though people still send patches to add them
- No new helpers
    - This decision caused fights between developers

non-uapi alternative

- struct_ops
- kfuncs

Kernel "struct *_ops"

- collection of callbacks
- typically named "struct foo_ops"
- Example: file_ops, inode_ops, vm_ops, net_device_ops, **tcp_congestion_ops**, ...
- It's an interface definition
- modules provide implementation for the interface

- In object oriented terminology "struct *_ops" describes an abstract class where callbacks are pure virtual functions

TCP congestion control

It's an example of good interface design.

```
struct tcp_congestion_ops {
        /* return slow start threshold (required) */
        u32 (*ssthresh)(struct sock *sk);

        /* do new cwnd calculation (required) */
        void (*cong_avoid)(struct sock *sk, u32 ack, u32 acked);

        /* call before changing ca_state (optional) */
        void (*set_state)(struct sock *sk, u8 new_state);

        /* call when cwnd event occurs (optional) */
        void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
        char                    name[TCP_CA_NAME_MAX];
        struct module           *owner;

        void (*init)(struct sock *sk);
        void (*release)(struct sock *sk);
};
```

TCP congestion control

- callbacks invoked by TCP networking stack
- lots of congestion control algorithms implemented as built-in and as kernel modules

net/ipv4/tcp_bbr.c: return tcp_register_congestion_control(&tcp_bbr_cong_ops);

net/ipv4/tcp_bic.c: return tcp_register_congestion_control(&bictcp);

net/ipv4/tcp_cubic.c: return tcp_register_congestion_control(&**cubictcp**);

net/ipv4/tcp_dctcp.c: return tcp_register_congestion_control(&dctcp);

...

TCP congestion control

```
net/ipv4/tcp_cubic.c:

static struct tcp_congestion_ops cubictcp = {
        .init           = cubictcp_init,
        .ssthresh       = cubictcp_recalc_ssthresh,
        .cong_avoid     = cubictcp_cong_avoid,
        .set_state      = cubictcp_state,
        .undo_cwnd      = tcp_reno_undo_cwnd,
        .cwnd_event     = cubictcp_cwnd_event,
        .pkts_acked     = cubictcp_acked,
        .owner          = THIS_MODULE,
        .name           = "cubic",
};
```

TCP congestion control

```
static void cubictcp_cwnd_event(struct sock *sk, enum tcp_ca_event event)
{
        if (event == CA_EVENT_TX_START) {
                struct bictcp *ca = inet_csk_ca(sk);
                u32 now = tcp_jiffies32;
                s32 delta;

                delta = now - tcp_sk(sk)->lsndtime;

                /* We were application limited (idle) for a while.
                 * Shift epoch_start to keep cwnd growth to cubic curve.
                 */
                if (ca->epoch_start && delta > 0) {
                        ca->epoch_start += delta;
                        if (after(ca->epoch_start, now))
                                ca->epoch_start = now;
                }
                return;
        }
}
```

TCP congestion control in BPF

- tcp_congestion_ops was an inspiration for BPF struct_ops
- Designed by Martin KaFai Lau <martin.lau@kernel.org> 4+ years ago
- Goals:
    - callbacks don't need to change on the kernel side to call into BPF programs
    - BPF programs are indistinguishable from kernel modules implementing *_ops
    - do not impose any uapi restrictions on the kernel

TCP congestion control in BPF

tools/testing/selftests/bpf/progs/bpf_cubic.c:

```
SEC(".struct_ops")
struct tcp_congestion_ops cubic = {
        .init           = (void *)bpf_cubic_init,
        .ssthresh       = (void *)bpf_cubic_recalc_ssthresh,
        .cong_avoid     = (void *)bpf_cubic_cong_avoid,
        .set_state      = (void *)bpf_cubic_state,
        .undo_cwnd      = (void *)bpf_cubic_undo_cwnd,
        .cwnd_event     = (void *)bpf_cubic_cwnd_event,
        .pkts_acked     = (void *)bpf_cubic_acked,
        .name           = "bpf_cubic",
};
```

TCP congestion control in BPF

```c
SEC("struct_ops")
void BPF_PROG(bpf_cubic_cwnd_event, struct sock *sk, enum tcp_ca_event event)
{
	if (event == CA_EVENT_TX_START) {
		struct bpf_bictcp *ca = inet_csk_ca(sk);
		__u32 now = tcp_jiffies32;
		__s32 delta;

		delta = now - tcp_sk(sk)->lsndtime;

		/* We were application limited (idle) for a while.
		 * Shift epoch_start to keep cwnd growth to cubic curve.
		 */
		if (ca->epoch_start && delta > 0) {
			ca->epoch_start += delta;
			if (after(ca->epoch_start, now))
				ca->epoch_start = now;
		}
		return;
	}
}
```

TCP congestion control in BPF vs kernel

```
SEC("struct_ops")
void BPF_PROG(bpf_cubic_cwnd_event, struct sock *sk, enum tcp_ca_event
event)
{
	if (event == CA_EVENT_TX_START) {
		struct bpf_bictcp *ca = inet_csk_ca(sk);
		__u32 now = tcp_jiffies32;
		__s32 delta;

		delta = now - tcp_sk(sk)->lsndtime;

		/* We were application limited (idle) for a while.
		 * Shift epoch_start to keep cwnd growth to cubic curve.
		 */
		if (ca->epoch_start && delta > 0) {
			ca->epoch_start += delta;
			if (after(ca->epoch_start, now))
				ca->epoch_start = now;
		}
		return;
	}
}
```

```
static void cubictcp_cwnd_event(struct sock *sk, enum tcp_ca_event event)
{
	if (event == CA_EVENT_TX_START) {
		struct bictcp *ca = inet_csk_ca(sk);
		u32 now = tcp_jiffies32;
		s32 delta;

		delta = now - tcp_sk(sk)->lsndtime;

		/* We were application limited (idle) for a while.
		 * Shift epoch_start to keep cwnd growth to cubic curve.
		 */
		if (ca->epoch_start && delta > 0) {
			ca->epoch_start += delta;
			if (after(ca->epoch_start, now))
				ca->epoch_start = now;
		}
		return;
	}
}
```

TCP congestion control in BPF vs kernel

```c
SEC("struct_ops")
void BPF_PROG(bpf_cubic_cwnd_event, struct sock *sk, enum tcp_ca_event
event)
{
        if (event == CA_EVENT_TX_START) {
                struct bpf_bictcp *ca = inet_csk_ca(sk);
                __u32 now = tcp_jiffies32;
                __s32 delta;

                delta = now - tcp_sk(sk)->lsndtime;

                /* We were application limited (idle) for a while.
                 * Shift epoch_start to keep cwnd growth to cubic curve.
                 */
                if (ca->epoch_start && delta > 0) {
                        ca->epoch_start += delta;
                        if (after(ca->epoch_start, now))
                                ca->epoch_start = now;
                }
                return;
        }
}
```

```c
static void cubictcp_cwnd_event(struct sock *sk, enum tcp_ca_event event)
{
        if (event == CA_EVENT_TX_START) {
                struct bictcp *ca = inet_csk_ca(sk);
                u32 now = tcp_jiffies32;
                s32 delta;

                delta = now - tcp_sk(sk)->lsndtime;

                /* We were application limited (idle) for a while.
                 * Shift epoch_start to keep cwnd growth to cubic curve.
                 */
                if (ca->epoch_start && delta > 0) {
                        ca->epoch_start += delta;
                        if (after(ca->epoch_start, now))
                                ca->epoch_start = now;
                }
                return;
        }
}
```

Same speed!

compiles to BPF ISA and JITed to native

compiles to native

Why implement TCP congestion control in BPF instead of kernel module?

- Safety
    - If it loads it won't crash the kernel
- Portability
    - Doesn't depend on the kernel version
    - Compile once and load BPF programs on many servers running different kernel versions
- Debuggability and observability
    - BPF programs are compiled with source code embedded in
    - GPL license is enforced
    - bpftool can profile and examine loaded programs

Existing and upcoming struct_ops users

- tcp_congestion_ops
- hid_bpf_ops
  - HID drivers
- sched_ext_ops
  - task scheduler
- Qdisc_ops
  - Network queuing discipline (when fq, fq_codel, pfifo, htb is not enough)
- fuse-bpf
- OOM-bpf

Interaction between kernel and BPF code

- Kernel C code is compiled into native CPU ISA with native calling convention
- BPF C code is compiled into BPF ISA with BPF calling convention
    - JIT translate BPF ISA into native ISA
    - calling from/to kernel/BPF requires conversion of arguments/return value
- BPF calling convention prescribes that arguments are passed in R1-R5 and return value in R0
    - LLVM and GCC compile C to BPF ISA this way

Comparison of calling conventions

	BPF	x86	Arm64	Risc-V
Arg 1	r1	rdi	x0	a0
Arg 2	r2	rsi	x1	a1
Arg 3	r3	rdx	x2	a2
Arg 4	r4	rcx	x3	a3
Arg 5	r5	r8	x4	a4
Arg 6		r9	x5	a5
Return	r0	rax	x0	a0

No extra cost to/from BPF on x86

	BPF	x86	Arm64	Risc-V
Arg 1	r1	rdi	x0	a0
Arg 2	r2	rsi	x1	a1
Arg 3	r3	rdx	x2	a2
Arg 4	r4	rcx	x3	a3
Arg 5	r5	r8	x4	a4
Arg 6		r9	x5	a5
Return	r0	rax	x0	a0

One to one mapping of BPF registers to x86 registers

One extra mov to return from BPF to Arm64

	BPF	x86	Arm64	Risc-V
Arg 1	r1	rdi	**x0**	a0
Arg 2	r2	rsi	x1	a1
Arg 3	r3	rdx	x2	a2
Arg 4	r4	rcx	x3	a3
Arg 5	r5	r8	x4	a4
Arg 6		r9	x5	a5
Return	r0	rax	**x0**	a0

- The first function argument and return value are in the same register
- JIT has to map BPF R1 and R0 to two different registers and add an extra copy after the CALL instruction
  - R1 is mapped to x0
  - R0 is mapped to x7

Ways of calling into the kernel

- helpers
- kfuncs

BPF helpers: compiler translate calling conventions

```
// kernel/bpf/helpers.c:
BPF_CALL_2(bpf_map_lookup_elem, struct bpf_map *, map, void *, key)
{
	return (unsigned long) map->ops->map_lookup_elem(map, key);
}

// macro magic expands into:
static inline u64 ____bpf_map_lookup_elem(struct bpf_map * map, void * key)
{
	return (unsigned long) map->ops->map_lookup_elem(map, key);
}
u64 bpf_map_lookup_elem(u64 map, u64 key, u64 r3, u64 r4, u64 r5) // BPF program calls this function
{
	return ____bpf_map_lookup_elem((struct bpf_map *)map, (void *)key);
}

// and compiled to:
(gdb) disassemble bpf_map_lookup_elem
Dump of assembler code for function bpf_map_lookup_elem:
   0xffffffff811f40c0  <+0>:  endbr64
   0xffffffff811f40c4  <+4>:  call   0xffffffff8105b0f0 <__fentry__>
   0xffffffff811f40c9  <+9>:  mov    (%rdi),%rax
   0xffffffff811f40cc <+12>:  jmp    *0x60(%rax)
End of assembler dump.
```

All arguments and return value are in correct registers. No extra copies.

BPF helpers ... no more

- Helper's disadvantage:
    - helpers have hard coded IDs in uapi/bpf.h
    - kernel modules cannot add them
    - subsystems cannot easily introduce them

- Solution:
    - stop adding helpers and introduce kfunc mechanism
    - kfunc is an unstable interface between BPF programs and the kernel
    - kernel modules can define their own kfuncs

git grep "FN(" include/uapi/linux/bpf.h	211 helpers
git grep '^__bpf_kfunc\>'	175 kfuncs

BPF kfuncs internals

- kfuncs rely on BPF Type Format (BTF)
- function prototype is converted to btf_func_model

```
struct btf_func_model {
        u8 ret_size;
        u8 ret_flags;
        u8 nr_args;
        u8 arg_size[MAX_BPF_FUNC_ARGS];
        u8 arg_flags[MAX_BPF_FUNC_ARGS];
};
```

- JITs use btf_func_model to translate BPF calling convention to native
    - nop on x86-64 because
        - all BPF registers are mapped 1-1 to x86 registers and
        - type promotion rules are the same (unlike risc-v)
    - not easy on x86-32

Calling kfuncs on x86-32

i386 kernel is compiled with "-mregparm=3":

The first three args of a function will be considered for putting into the 32bit register EAX, EDX, and ECX.

Two 32bit registers are used to pass a 64bit arg.

void foo(u32 a, u32 b, u32 c, u32 d):

- u32 a: EAX
- u32 b: EDX
- u32 c: ECX
- u32 d: stack

void foo(u64 a, u32 b, u32 c):

- u64 a: EAX (lo32) EDX (hi32)
- u32 b: ECX
- u32 c: stack

Any kernel function can be a kfunc

```
#define __bpf_kfunc __used __retain noinline

__bpf_kfunc void bpf_rcu_read_lock(void)
{
        rcu_read_lock();
}

BTF_KFUNCS_START(common_btf_ids)
BTF_ID_FLAGS(func, bpf_rcu_read_lock)
BTF_KFUNCS_END(common_btf_ids)
```

- Unlike helpers there is no extra code from BPF_CALL_N() macros that convert calling convention
- JITs generate translation code
- A lot more efficient on 32-bit architectures

Ways of calling into BPF program

- prog->bpf_func(ctx, ...);
    - all networking hooks are done this way
- tracing style
    - kprobe, fentry, tracepoint
- struct_ops

Old way of calling into BPF program

```
struct xdp_buff xdp;
struct bpf_prog *prog;
u32 ret;

// store all arguments that needs to be passed to BPF prog in the "context" structure
xdp_init_buff(&xdp, ...);
xdp_prepare_buff(&xdp, hard_start, data, ...);

prog = // fetch the prog pointer from somewhere

// call it with a single "context" argument
ret = prog->bpf_func(&xdp, prog->insnsi /* for interpreter */);

switch (ret) {
case XDP_PASS:
 ..
```

Disadvantages of old way of calling into BPF

- "context" structure is uapi
    - think twice of every field and ways to extend
- plenty of boiler plate code to pack arguments into "context" struct

struct_ops way of calling

```
static inline void tcp_ca_event(struct sock *sk,
                                const enum tcp_ca_event event)
{
        const struct inet_connection_sock *icsk = inet_csk(sk);

        if (icsk->icsk_ca_ops->cwnd_event)
                icsk->icsk_ca_ops->cwnd_event(sk, event);
}

net/ipv4/tcp_input.c: tcp_ca_event(sk, CA_EVENT_ECN_IS_CE);
net/ipv4/tcp_input.c: tcp_ca_event(sk, CA_EVENT_ECN_NO_CE);
net/ipv4/tcp_input.c: tcp_ca_event(sk, CA_EVENT_LOSS);
net/ipv4/tcp_input.c: tcp_ca_event(sk, CA_EVENT_COMPLETE_CWR);
net/ipv4/tcp_output.c: tcp_ca_event(sk, CA_EVENT_CWND_RESTART);
net/ipv4/tcp_output.c: tcp_ca_event(sk, CA_EVENT_TX_START);
```

struct_ops way of calling

- Just a normal C code
- BPF struct_ops mechanism generates trampoline to call

```
void (*cwnd_event)(struct sock *sk, enum tcp_ca_event ev);
```

- sk in %rdi is stored to stack
- ev in %rsi is stored to stack
- calls JITed bpf prog **directly**

```
SEC("struct_ops")
void BPF_PROG(bpf_cubic_cwnd_event, struct sock *sk, enum tcp_ca_event event)
// access 'sk' from BPF program is a read from stack
// while the verifier enforces types
```

struct_ops way of calling

- BPF struct_ops mechanism populates

```
struct tcp_congestion_ops {
  .cwnd_event = // pointer to trampoline
} cubic;
```

- Kernel calls native ops callback
    - pass arguments in registers + indirect call
- Kernel calls BPF struct_ops callback
    - pass arguments on stack + indirect call + direct call

struct_ops way of calling

- No kernel side changes
- No uapi contract

How to design kernel extension

- Forget about BPF
- Think of clean abstract interface
  - A set of callbacks is an interface from kernel to kernel module (or BPF struct_ops programs)
  - export_symbol is an interface from kernel module back into the kernel
  - kfuncs is an equivalent for BPF programs
    - It's more tightly controlled due to type enforcement
- Register new ops
  - register_bpf_struct_ops(&bpf_tcp_congestion_ops, tcp_congestion_ops);

BPF mission

or why we're still passionate about this code

- To innovate
    - helpers, struct_ops, kfuncs development satisfies our thirst for innovation

- To enable others to innovate
    - It's a joy to see how struct_ops enabled hid-bpf and sched-ext

- To challenge what's possible
    - When everyone says "It's impossible"
      we reply "The whole thing maybe impossible, but this part is doable".