Rust in the Kernel

via eBPF
Dave

Software Engineer @ Red Hat
Michal

Software Engineer @ Deepfence
vadorovsky @ Github, Discord, Twitter
There is a race to re-write the Kernel in BPF and Rust... We say, why not both?
eBPF
eBPF is eating the kernel

- **IR decoding** can be done with eBPF, replacing the lirc daemon.

- **HID-bpf** is being actively worked on.

- **LSM** could replace selinux/apparmor.

- Prediction: More subsystems will follow.

- BPF kernel-space is heavily dominated by C, although Rust is an alternative.
Aya
C -> eBPF compilation (the old, still the most popular way)

Source code
fw_kern.c

clang
--target=bpf -c

eBPF object file
fw_kern.o

One big source file which #includes the other ones.

-c means skipping the linking phase.

ld and lld can't link eBPF
Linking with bpftool (via libbpf)

Inputs (eBPF object files)

- **Object file**
  - lib.o

- **Object file**
  - fw_prog.o

- **eBPF object file**
  - fw.o

- **bpftool gen**
  - fw.o
  - fw_prog.o
  - lib.o
Rust compilation

Library crate

Source code
dep/src/lib.rs

rustc
--crate-type
lib

Rust library
libdep.rlib

Binary crate

Source code
app/src/main.rs

rustc
--crate-type bin
--extern-dep=libdep.rlib

Linker (ld / lld)
app*.o
libdep.rlib

Executable
app

Rust library
libdep.rlib
Rust -> eBPF compilation

### Library crate

**Source code**
fw-common/src/lib.rs

**Rustc**
--crate-type lib

**Rust library**
fw_common.rlib

### Binary crate

**Source code**
fw-ebpf/src/main.rs

**Rustc**
--crate-type bin
--extern-dep=fw_common.rlib

**Linker (bpf-linker)**
app*.o
fw_common.rlib

**eBPF object file**
fw

**Rust library**
fw_common.rlib
bpf-linker

Inputs (LLVM bitcode)

- **Bitcode file** `.bc`
- **Object file (with embedded bc)** `.o`
- **.rlib archive** contains object file
- **.a archive** contains object file

```
  bpf-linker
  --output fw
  -- fw.(o|bc)
  fw_common.(o|bc)
```

**eBPF object file** `fw`
The Aya Ecosystem

user-land
- aya
- aya-log

kernel-land
- aya-bpf
- aya-log-ebpf

tools
- aya-template
- aya-tool

compilation
- bpf-linker
- rustc
Features

- CO:RE support (in userspace only)
- `aya-log` using perf buffers for logging
- async support (in the userspace) with Tokio and async-std
- Used by Deepfence, Exein, Parca and Red Hat.
Type Safety

```c
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
SEC("tracepoint")
int is_it_tracepoint_xdp_or_classifier(struct __sk_buff *skb) {
    return XDP_PASS;
}
```

```rust
#[xdp(name = "incorrect_xdp")]
pub fn incorrect_xdp(ctx: SkBuffContext) -> u32 {
    xdp_action::XDP_PASS
}
```

```bash
$ clang -O2 -emit-llvm -c incorrect_xdp.c -o - | llc
$ clang -O2 -emit-llvm -c incorrect_xdp.c -o - | llc
```

```bash
$ cargo xtask build-ebpf
 [...] 
error[E0308]: mismatched types
 --> src/main.rs:7:1
  7 | #[xdp(name = "incorrect_xdp")]
  | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ expected struct `SkBuffContext`, found struct `XdpContext`
8 | pub fn incorrect_xdp(ctx: SkBuffContext) -> u32 {
 [...] 
```
Error Handling

```c
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __uint(max_entries, 1024);
    __type(key, pid_t);
    __type(value, u32);
} pids SEC(".maps");

SEC("fentry/kernel_clone")
int BPF_PROG(kernel_clone, struct kernel_clone_args *args) {
    /* Get the pid */
    pid_t pid = bpf_get_current_pid_tgid() >> 32;
    /* Save the pid in map */
    u32 val = 0;
    int err = bpf_map_update_elem(&pids, &pid, &val, 0);
    if (err < 0)
        return err;
    return 0;
}
```

```rust
#[map(name = "pids")]
static mut PIDS: HashMap<u32, u32> = HashMap::<u32, u32>::with_max_entries(1024, 0);

#[fentry(name = "kernel_clone")]
pub fn kernel_clone(ctx: FEntryContext) -> u32 {
    match unsafe { try_kernel_clone(ctx) } {
        Ok(ret) => ret,
        Err(_) => 1,
    }
}

fn try_kernel_clone(ctx: FEntryContext) -> Result<u32, c_long> {
    // Get the pid
    let pid = ctx.pid();
    // Save the pid in map.
    unsafe { PIDS.insert(&pid, &0, 0)? };  
    Ok(0)
}
```
Common Code/Structs

**Others:**

Manually copy/paste struct definition from kernel code to userspace code (no code sharing).

Use a shared header files to share structs between userspace/kernel. Hard to integrate a linker.

Use BTF to inspect struct definitions from ELF file and generate code for userspace (no code sharing).

**Aya:**

Use a “common” crate to share structs and code between kernel and userspace.
Minimal build dependencies

C

- LLVM, clang
- libbpf
- make

Rust

- rustup (which installs rustc and cargo)
- bpf-linker
Starting a new project

- Getting an inspiration from [https://github.com/libbpf/libbpf-bootstrap/](https://github.com/libbpf/libbpf-bootstrap/)
- Your repo and Makefile have to be handcrafted

Rust
Shared Challenges for Aya and Rust in the Kernel
BPF Type Format (BTF)

A compact debug info format

Since 5.xx kernel modules can provide BTF

Required for CO:RE

- The rustc bpf target and LLVM need work to reliably generate BTF

- There are LLVM intrinsics we must use to be able to generate the necessary relocations for CO:RE in kernel-space.

- BTF has been designed to represent C structures-idioms. Can we cleanly map Rust to this?
Working with Kernel types isn’t fun.
Many anonymous unions.
Macros don’t get expanded

How can we make this safer/better?
Questions?
Thanks!