Monitoring with eBPF

Nikola Zupancic

June 19, 2025

1 Progress

The program so far is *somewhat* capable of finding functions and their returns in *simple C programs that are stripped of debug info and symbols* (one program being a short 20 line program with a main and function foo, the other being some software that I wrote to share mouse and keyboard input across devices: github.com/c-ola/ioswitch).

It consists of a loader and a bpf program, as well as some short python scripts for finding symbols and analysis.

1.1 uprobe bpf

This is the bpf program that gets attached to the entry of a function, a nearly equivalent one also exists for the return, however it differs in using BPF_KRETPROBE as a macro, and has an extra return value argument.

The program finds the pid, calltime, instruction pointer, and base address of the .text section map (this behaviour differed between arm and x86_64, arm did not need the base address). It then writes the data to the ringbuf rb, which is read in the loader program.

```
struct {
    __uint(type, BPF_MAP_TYPE_RINGBUF);
    __uint(max_entries, 256 * 1024);
} rb SEC(".maps");

SEC("uprobe/entry_uprobe")
int BPF_KPROBE(uprobe_entry) {
    u64 start = bpf_ktime_get_ns();
    u64 ip = bpf_get_func_ip(ctx);
    pid_t pid = bpf_get_current_pid_tgid() >> 32;
    struct task_struct* task = (struct
        task_struct*)bpf_get_current_task();
    struct mm_struct* mm;
    bpf_probe_read_kernel(&mm, sizeof(struct mm_struct*), &task->mm);
    unsigned long base_code_addr;
    bpf_probe_read_kernel(&base_code_addr, sizeof(unsigned long),
        &mm->start_code);
    struct perf_data d = {
        .pid = pid,
        .ip = ip,
        .call_time = start,
        .base_code_addr = base_code_addr,
    };
    bpf_ringbuf_output(&rb, &d, sizeof(d), 0);
    bpf_printk("entry, start=%lu, rip=0x%lx, base_code_addr=0x%lx",
        start, ip, base_code_addr);
    return 0;
```

```
}
```

1.2 Loader

The loader program is inspired by some examples in the libbpf-bootsrap repo, following their general structure. It generates a skeleton header file from the compiled bpf object using bpftool in the build process.

```
bpftool gen skeleton build/uprobe.bpf.o > build/skel/uprobe.skel.h
```

This header file abstracts away alot of boilerplate in loading and attaching the bpf program. All that needs to be done is opening and loading the bpf program:

```
    skel = uprobe_bpf__open_and_load();
```

It can then be attached to the address of each symbol entry and return.

```
    const char* binary_name = elf_path;
    for (int i = 0; i < symbols.length; i++) {
        symbol* sym = symbols.values[i];
        uprobe_opts.retprobe = false;
        skel->links.uprobe_entry =
            bpf_program__attach_uprobe_opts(skel->progs.uprobe_entry, -1,
            binary_name, sym->addr, &uprobe_opts);
        if (!skel->links.uprobe_entry) {
            err = -errno;
            fprintf(stderr, "Failed to attach uprobe: %d\n", err);
            goto cleanup;
        }
        for (int j = 0; j < sym->num_returns; j++) {
            uprobe_opts.retprobe = true;
            unsigned long addr = sym->returns[j];
            skel->links.uprobe_ret = bpf_program__attach_uprobe_opts(
                skel->progs.uprobe_ret, -1, binary_name, addr,
                &uprobe_opts);
            if (!skel->links.uprobe_ret) {
                err = -errno;
                fprintf(stderr, "Failed to attach ret uprobe: %d\n", err);
                goto cleanup;
            }
        }
```

The data from the ringbuffer can then be handled by a callback function set by the following code.

```
    struct handle_ctx ctx;
    ctx.symbols = &symbols;
    struct ring_buffer *rb = NULL;
    rb = ring_buffer__new(bpf_map__fd(skel->maps.rb), handle_data, &ctx,
        NULL);
    if (!rb) {
        err = -1;
        fprintf(stderr, "Failed to create ring buffer\n");
        goto cleanup;
    }
    ctx.log_file = fopen("perf_log.log", "w");
```

```
int handle_data(void* vctx, void* dat, size_t dat_sz){
    struct handle_ctx ctx = *(struct handle_ctx*)vctx;
    struct perf_data *d = dat;
#ifdef __x86_64__
    long unsigned long addr = d->ip - d->base_code_addr +
        ctx.symbols->offset;
#else
    long unsigned long addr = d->ip - d->base_code_addr;
#endif
    int is_ret = 0;
    const char* name = get_symbol_name(ctx.symbols, addr, &is_ret);
    fprintf(ctx.log_file, is_ret ? "ret: " : "enter: ");
    fprintf(ctx.log_file, "pid=%d, name=%s, t=%llu, addr=%llx\n",
        d->pid, name, d->call_time, addr);
    printf(is_ret ? "ret: " : "enter: ");
    printf("pid=%d, name=%s, t=%llu, addr=%llx\n", d->pid, name,
        d->call_time, addr);
    return 0;
}
```

1.3 Symbol Data

The above process all relies on getting a ”symbols.json” file from the program. This is done through getting a symbol map from the linker with *-Wl,-Map=output.map* added to gcc args, and reading the compiled elf to try and get data about the elf sections. After pattern matching with regex on the map, the elf can be disassembled at each symbol address to find all returns before the next symbol.

I think that this process can be changed by using the binutil nm to find symbols from the unstripped binary, which can also likely be configured to give the size of each symbol, and thus the address of the return of the function (still have to look into this more).

1.4 Analysis

A sample perf_log.log can look like this.

```
...
enter: pid=7040, name=main, t=23616578499119, addr=814
enter: pid=7040, name=foo, t=23617579079046, addr=7d4
ret: pid=7040, name=foo, t=23617579372752, addr=810
enter: pid=7040, name=foo, t=23617579399002, addr=7d4
ret: pid=7040, name=foo, t=23617579472209, addr=810
enter: pid=7040, name=foo, t=23617579540459, addr=7d4
ret: pid=7040, name=foo, t=23617579647500, addr=810
...
...
...
ret: pid=7040, name=main, t=23617581723569, addr=890
...
```

The output of the analysis is simple at the moment. A graph visualization would be nice for this. I also think _start should not necessarily be traced by this program, however it does get caught by the symbol.json generation script. It could be manually ignored in the script.

```
call to _start took 0.130082ms
call to foo took 0.293706ms
call to foo took 0.073207ms
call to foo took 0.107041ms
call to foo took 0.07175ms
call to foo took 0.077583ms
call to foo took 0.07875ms
call to foo took 0.191332ms
call to foo took 0.076708ms
call to foo took 0.111416ms
call to foo took 0.160415ms
call to foo took 0.069708ms
call to foo took 0.068541ms
call to foo took 0.099749ms
call to foo took 0.087207ms
call to foo took 0.085166ms
call to foo took 0.068833ms
call to foo took 0.070582ms
call to foo took 0.081958ms
call to foo took 0.079916ms
call to foo took 0.085167ms
call to main took 1003.22445ms
1005.393267ms
```