BPF Guidelines

https://tinyurl.com/bpfguidelines2022

Why Guidelines?

To communicate high-level end-use best practices to newbies

- ebpf.io/projects lists all BPF-y projects (that meet stated requirements)
- ebpf.io/guidelines shows best practice projects (BSC selected)
- End-uses: observability, security, networking, application accelerators, etc.

Not about coding style: That should be documented by each project.

Observability

1. Familiarize yourself with traditional observability sources to avoid reinventing the wheel.
    a. For Linux: sources: /proc, netlink, etc. Tools: vmstat, iostat, pidstat, sar, etc.
    b. For Windows: Task Manager, PerfView, ETW, Xperf, etc.
2. Recommended requirements
    a. Linux 5.?+ (for BTF and CO-RE support)
    b. Windows TBD
3. Install bcc, try its tools, find some performance/debugging wins.
    a. Recommended to use the libbpf-tools versions (C-based)
4. Install bpftrace, try its one-liners, try custom one-liners and your own programs (see the reference guide). Learning BPF tracing this way is easy: It's like learning pseudocode that runs.
    a. If you develop your own bpftrace programs, use workload generators (including microbenchmarks and those you write yourself) to sanity check the bpftrace output. Emit a prime-number-ish number of events (e.g., 23000) and make sure your tool agrees. Common mistakes include tracing a fast path but missing a slow path. Make sure your workload generator can simulate a variety of things.
5. For front-end UIs, consider building upon:
    a. bcc tool output. This is like scraping the output of iostat(1).
    b. bpftrace -f json. Any bpftrace tool or one-liner (including your own from (4)) can be converted to emit json. This is like building a Python/Perl agent that uses a BPF library: The hot-path code runs in efficient BPF, and the low-frequency code (metric collection and reporting) runs in an easy-to-maintain higher-level language. This is likely suitable for most people.
6. If you make it this far, you can now explore developing a custom bcc program for your needs.
    a. libbpf-tools (BTF, CO-RE, C-based) is the recommended API.
    b. Python tools is the older original API. No longer recommended.

   c. Study the bcc CONTRIBUTING SCRIPTS checklist, even if you have no intention to ever contribute the tool. It has lots of good advice, such as

BPF Development

How to begin
Check for existence of docs
Bpf bootstrap
Look at selftests libbpf
Understand pros vs cons:

- BPF development is much faster than kernel module development

Understand high level roadmap: where we want to be

Security

1. Familiarize yourself with traditional security tools to avoid reinventing the wheel.
   a. For Linux: LSM, auditd
2. Recommended requirements
   a. Linux 5.x?
3. BPF unprivileged disabled by default
4. Install bcc
5. Study events to trace
   a. LSM hooks
   b. BsidesSF talk 2017
6. Study time of use attacks (TOU) and understanding bcc observabality tools aren't suitable (different set of requirements)
   a. Nabil's execsnoop rewrite
   b. Probing user memory is racy
   c. Using comm
   d. CNCF eBPF day security talk
7. Signed programs

Networking

1. Familiarize yourself with networking capabilities to avoid reinventing the wheel.
   a. XDP, Cilium, Katran
   b. IPv6, UDP, HTTP3 QUIC
   c. Get experience with: tc, qdiscs
   d. Design: understand options and what's right for you;
   understand scope: cgroup, socket, device, xdp software device
2. Recommended requirements

    a. Linux 5.x?
3. Install bcc
4. Have a quick win: XDP/Cilium/Katran?
5. Experiment with libbpf tc libraries

Performance Accelerators

1. Familiarize yourself with application/kernel internals and existing accelerators.
2. Recommended requirements
    a. Linux 5.x?
3. Study bmc-cache as an example accelerator (memcached)
4. Install bcc

Meta

1. Familiarize yourself with:
    a. Problem space, incl. Other solutions
    b. Use cases
    c. Technologies
    d. Design: what makes sense for your workload
2. OS requirements
3. Get a quick win
4. Recommended programming path …