eBPF Debugging Infrastructure

Current Techniques and Additional Proposals

Quentin Monnet
<quentin.monnet@netronome.com>
What do we want to debug, troubleshoot?

To achieve this:

• What debugging tools and methods are available?

• What is missing?
What to Debug: Many Levels

- Compile time
- Load time
- Runtime
- User space
- Kernel development
- User space programming
- Hardware
- Agilio SmartNIC
Objectives:

- Make sure the eBPF bytecode is generated as intended when compiling from C to eBPF

We have:

- LLVM backend: compilation
- llvm-objdump: dump generated bytecode
- eBPF assembly (LLVM): hack a sequence of instructions
Objectives:

- Load the program and make it pass the verifier, or understand why it is rejected

We have:

- `libbpf / ip / tc`: load or list programs
- `libbpf / bpftool` (and `tc` to some extent): eBPF object management
- Output from verifier logs, kernel logs, extack messages
- Documentation (filter.txt, Cilium guide)

What about:

- Checking what loads: `bpftool prog probe my_file.o` (work in progress, idea from Daniel)
- `man` pages (`bpf(2)` or `tc-bpf(8)` are badly outdated)
- Troubleshooting F.A.Q.? (e.g. some items already in filter.txt)
Objectives:

- Understand why a program does not run as intended, for example when processing network packets

We have:

- `bpftool`: introspection for maps / programs, object management. Readability improved with BTF.
- `bpf_trace_printk()`, perf events: print items, data.
- (Limited user space eBPF virtual machines)
- Hooks in binutils-gdb, but no simulator support.
- `tools/bpf/bpf_dbg.c` (cBPF)

What about:

- Debugger: break points, possibility to dump registers / stack / context?
  - Complete support in GDB?
  - Anything doable with LLDB? But how to pass packet data?
  - Extend `BPF_PROG_TEST_RUN` infrastructure? (idea: Daniel)
Objectives:

- Improve the eBPF architecture in the kernel, without breaking existing features

We have:

- Selftests: verifier, test programs
- Samples programs
- BPF_PROG_TEST_RUN infrastructure
- KASAN, syzkaller

What about:

- Having all JITs built-in, dump (then test) images for all architectures (idea: Daniel)
Objectives:

- Debug or enhance a program managing eBPF objects
- Generally improve eBPF support in the toolchain

We have:

- `strace`, `valgrind` support: tracing system calls, memory checks

What about:

- Probing kernel for features (with bpftool)? (idea: Daniel)
- Bytecode generation: ethtool n-tuples (in progress), libpcap?
Discussion

What do you feel is missing for debugging eBPF?
Backup: Dump All JIT Images

Kernel JITs: ARM64, ARM32, PowerPC64, s390, Sparc64, MIPS, x86_64, x86_32
Offload: NFP

Objectives:

- Test images for all architectures
- Find bugs or low hanging perf improvements

Idea (Daniel):

- All JIT built-in in the kernel
- Pass a flag to `bpf(PROG_LOAD, ...)` to JIT-compile for all arch
- Pass a flag to `bpf(OBJ_GET_INFO_BY_ID, ...)` to dump all images
- Simulate execution on several architectures
- Add tools/ to bootstrap VMs to test the images?
Member in union bpf_attr for bpf(BPF_PROG_TEST_RUN, attr, size):

struct { /* anonymous struct used by BPF_PROG_TEST_RUN command */
    __u32     prog_fd;
    __u32     retval;
    __u32     data_size_in;
    __u32     data_size_out;
    __aligned_u64     data_in;
    __aligned_u64     data_out;
    __u32     repeat;
    __u32     duration;
} test;

Fields data_out, data_out_size, retval, duration are filled by kernel

Idea:

- Add a field to pass break points (insn number, program entry point?)
- Add fields or buffer to dump internal state: register values, stack, data?
- Maybe a front-end loader? bpftool?
Example output:

```c
# bpftool kernel probe
/* System configuration */
#define HAVE_BPF_SYSCALL
#define UNPRIVILEGED_BPF_DISABLED 0
#define JIT_COMPILER_ENABLE 0
#define JIT_COMPILER_HARDEN 0
#define JIT_COMPILER_KALLSYMS 0
#define LINUX_VERSION_CODE 267008

/* eBPF program types */
#define HAVE_SOCKET_FILTER_PROG_TYPE
#define HAVE_KPROBE_PROG_TYPE
...
/* HAVE_STACK_MAP_TYPE is not set */

/* eBPF map types */
#define HAVE_HASH_MAP_TYPE
#define HAVE_ARRAY_MAP_TYPE
...

/* eBPF helper functions */
#define HAVE_BPF_MAP_LOOKUP_ELEM_HELPER
#define HAVE_BPF_MAP_UPDATE_ELEM_HELPER
...
/* HAVE_BPF_MSG_PUSH_DATA_HELPER is not set */
```
libpcap: patch the library or create an equivalent to use a similar syntax to produce eBPF programs

```bash
# tcpdump -d "port ssh"
(000) ldh [12]
(001) jeq #0x86dd   jt 2   jf 8
(002) ldb [20]
(003) jeq #0x6       jt 4   jf 19
(004) ldh [54]
(005) jeq #0x16      jt 18  jf 6
...
(019) ret #0
```

ethtool: implement a library to turn such rules into eBPF programs

```bash
# ethtool --config-ntuple eth0 flow-type tcp dst-port 22 action -1 <drop incoming SSH packets on a server>
```