FOSDEM’20 • Brussels, 2020-02-02

Tools and Mechanisms to Debug BPF Programs

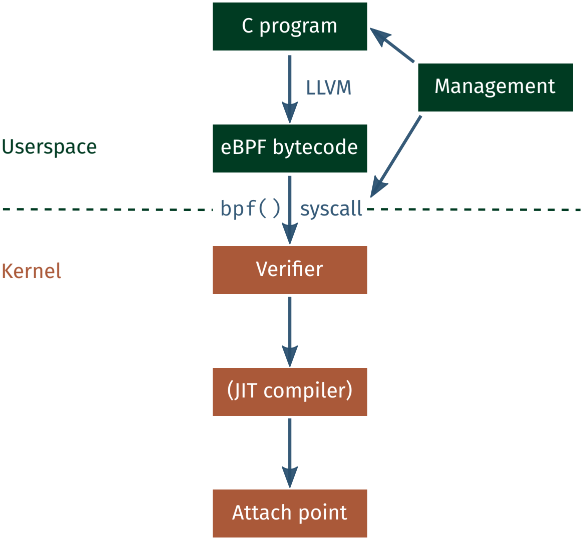
Quentin Monnet
@qeole

eBPF Programming

__e__xtended __B__erkeley __P__acket __F__ilter:

- User-written programs, usually compiled from C (or Go, Rust, Lua...) with clang/LLVM, to assembly-like bytecode
- Programs are injected into the kernel with the `bpf()` system call
- Verifier: programs terminate, are safe
- In-kernel interpreter, JIT (Just-in-Time) compiler
- Once loaded, programs can be attached to a hook in the kernel
- 64-bit instructions, 11 registers, 512 B stack, not Turing-complete
- Additional features: “maps”, kernel helper functions, BTF, ...

eBPF Workflow



eBPF Use Cases

Main use cases:

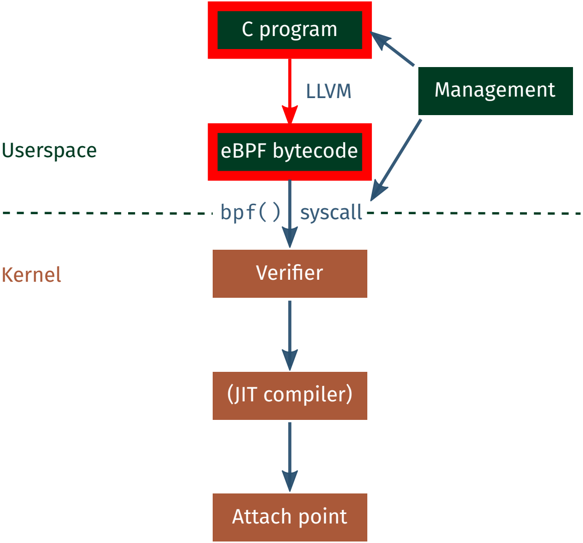
- Networking (tc, XDP: driver-level hook)
- Tracing, monitoring (think DTrace)
- Socket filtering (cgroups)
- Security (LSM, work in progress)
- And more!

Outline

- (Reminder on eBPF… DONE)
- The tools to inspect eBPF objects, at each step of the workflow
- Getting familiar with bpftool
- Next steps for BPF introspection and debugging

Inspecting BPF Objects

eBPF Workflow



Compile Time

Objective:

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

Compile Time: Compile and Dump

- Compile with clang/LLVM (or gcc, but fewer BPF features supported):

```
$ clang -O2 -emit-llvm -c sample.c -o - | \
      llc -march=bpf -mcpu=probe -filetype=obj -o sample.o
```

- Dump instructions from object file with **llvm-objdump** (v4.0+) (prior to kernel injection, relocation, rewrites)

```
$ llvm-objdump -d -r -print-imm-hex sample.o

sample.o:       file format ELF64-BPF

Disassembly of section .text:
func:
       0:       b7 00 00 00 00 00 00 00         r0 = 0
       1:       95 00 00 00 00 00 00 00         exit
```

- If `-g` is passed to clang, `llvm-objdump -S` can dump the original C code

Compile Time, in Two Steps: eBPF Assembly

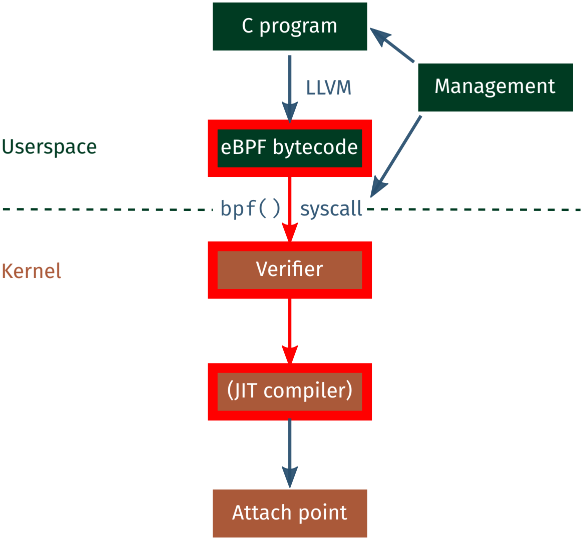
- Compile from C to eBPF assembly file

```
$ clang -target bpf -S -o sample.S sample.c
$ cat sample.S
            .text
            .globl  func                   # -- Begin function func
            .p2align        3
    func:                                  # @func
    # %bb.0:
            r0 = 0
            exit
                                           # -- End function
```

- ... Hack...
- Then compile from assembly to eBPF bytecode (LLVM v6.0+)

```
$ clang -target bpf -c -o sample.o sample.S
```

eBPF Workflow



Load Time

Objective:

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

Resources:

- **libbpf** / **`bpftool`** / **`ip`** / **`tc`** / **bcc**: load or list programs, manage objects
- Output from verifier logs, libbpf, kernel logs, extack messages
- Documentation (`filter.txt`, Cilium guide)

The Kernel eBPF Verifier: Checking Programs for Safety

The verifier performs checks on control flow graph and individual insns:

- Erroneous syntax (unknown or incorrect usage for the instruction)
- Too many instructions or maps or branches
- Back edges (i.e. loops, not bounded) in the control flow graph
- Unreachable instructions
- Jump out of range
- Out of bounds memory access
- Access to forbidden context fields (read or write)
- Reading access to non-initialized memory (stack or registers)
- Use of forbidden helpers for the current type of program
- Use of GPL helpers in non-GPL program (mostly tracing)
- `R0` not initialized before exiting the program
- Memory access with incorrect alignment
- Missing check on result from `map_lookup_elem()` before accessing map element
- ...

The Kernel eBPF Verifier: Example message

Possible out-of-bound access to packet data (no check on packet length):

```
# ip link set dev eth0 xdp object sample.o

Prog section 'action' rejected: Permission denied (13)!
 - Type:         6
 - Instructions: 41 (0 over limit)
 - License:      GPL

Verifier analysis:

0: (bf) r2 = r1
1: (7b) *(u64 *)(r10 -16) = r1
2: (79) r1 = *(u64 *)(r10 -16)
3: (61) r1 = *(u32 *)(r1 +76)
invalid bpf_context access off=76 size=4

Error fetching program/map!
```

Problem: error messages good for developers, but cryptic for newcomers

Make Sure to Get Verifier Information

Still, we do want the messages!

- Use debug flags when available
  - Debug buffer for verifier logs (pass to `bpf()`)
  - Debug flag for libbpf
  - Activate both in bpftool with `--debug`
- Interpret information:
  - Search the docs, `Documentation/networking/filter.txt`, Cilium guide
  - Read kernel code
  - To do: some kind of documentation/FAQ detailing the errors?

Program is Loaded: Introspection

We have passed the verifier! The program is loaded in the kernel

- For map and program introspection: **bpftool**
    - List maps and programs
    - Load a program, pin it
    - Dump program instructions (eBPF or JIT-ed)
    - Dump and edit map contents
    - etc.

We will come back to bpftool later

BTF: *BPF Type Format*

BTF objects embed debug information on programs and maps
They are also use internally by the kernel for some advanced BPF features

- Embed BTF information when compiling programs:
  Compile with LLVM v8+, use `-g` flag
- For maps, some wrapping needed in the C source code

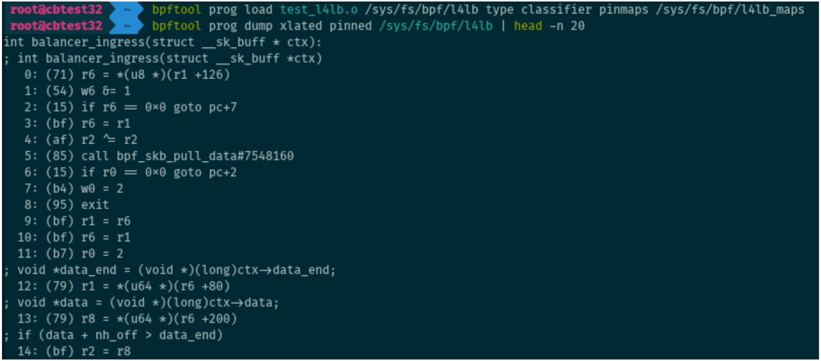
```
struct my_value { int x, y, z; };

struct {
        int type;
        int max_entries;
        int *key;
        struct my_value *value;
} btf_map SEC(".maps") = {
        .type = BPF_MAP_TYPE_ARRAY,
        .max_entries = 16,
};
```

(See kernel commit abd29c931459)

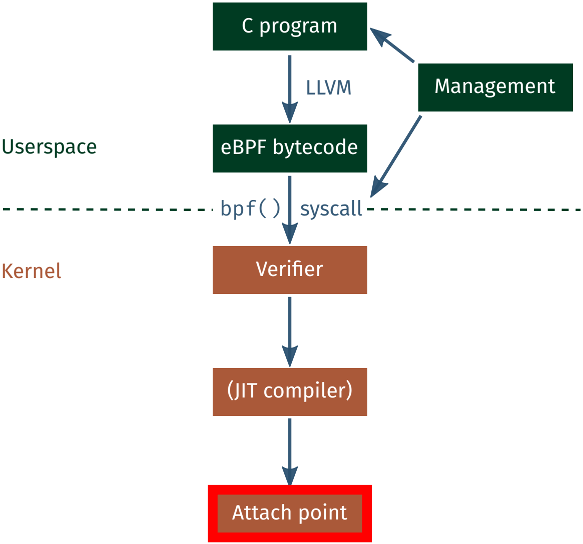
BTF: *BPF Type Format*

Exemple: Program dump from kernel, with C source code



```
root@cbtest32  ~  bpftool prog load test_l4lb.o /sys/fs/bpf/l4lb type classifier pinmaps /sys/fs/bpf/l4lb_maps
root@cbtest32  ~  bpftool prog dump xlated pinned /sys/fs/bpf/l4lb | head -n 20
int balancer_ingress(struct __sk_buff * ctx):
; int balancer_ingress(struct __sk_buff *ctx)
   0: (71) r6 = *(u8 *)(r1 +126)
   1: (54) w6 &= 1
   2: (15) if r6 == 0x0 goto pc+7
   3: (bf) r6 = r1
   4: (af) r2 ^= r2
   5: (85) call bpf_skb_pull_data#7548160
   6: (15) if r0 == 0x0 goto pc+2
   7: (b4) w0 = 2
   8: (95) exit
   9: (bf) r1 = r6
  10: (bf) r6 = r1
  11: (b7) r0 = 2
; void *data_end = (void *)(long)ctx->data_end;
  12: (79) r1 = *(u64 *)(r6 +80)
; void *data = (void *)(long)ctx->data;
  13: (79) r8 = *(u64 *)(r6 +200)
; if (data + nh_off > data_end)
  14: (bf) r2 = r8
```

eBPF Workflow



Runtime

Objective:

- Understand why a program does not run as intended

Several solutions

Debugging at Runtime with bpf_trace_printk()

- eBPF helper `bpf_trace_printk()`
  Prints to `/sys/kernel/debug/tracing/trace`
  Example snippet:

  ```
  const char fmt[] = ”First four bytes of packet: %x\n”;
  bpf_trace_printk(fmt, sizeof(fmt), *(uint32_t *)data);
  ```

Debugging at Runtime with Perf Events

- “Perf event arrays”, more efficient than `bpf_trace_printk()`
  Example: dump data from packet

```
struct bpf_map_def SEC(”maps”) pa = {
        .type = BPF_MAP_TYPE_PERF_EVENT_ARRAY,
        .key_size = sizeof(int),
        .value_size = sizeof(int),
        .max_entries = 64,
};

int xdp_prog1(struct xdp_md *xdp)
{
        int key = 0;
        bpf_perf_event_output(xdp, &pa, 0x20ffffffffULL, &key, 0);
        return XDP_PASS;
}
```

- Contrary to `bpf_trace_printk()`, can be used with hardware offload

Debug BPF with BPF

BPF can be used for tracing, and comes to the rescue

- Possible to attach tracing BPF programs at entry and exit of a networking BPF program (Linux 5.5)
  - E.g. get packet data in input and/or output of the program
  - See `tools/testing/selftests/bpf/progs/test_xdp_bpf2bpf.c` and related
  - Not sure if compatible with tracing programs?
- Use bcc or bpftrace to examine what happens in the kernel (can also be used at verification time to follow verification steps)

Testing Programs: BPF_PROG_TEST_RUN

`BPF_PROG_TEST_RUN` subcommand for the `bpf()` system call

- Manually run a program with given input data and context
- Output data and context are retrieved

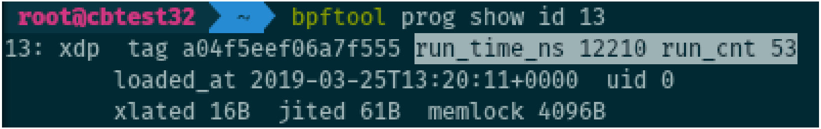
Limitations:

- Not available for all programs (mostly networking for now)
- Tracing: How to check kernel data structures are changed?
- Some BPF helpers hard to support (`bpf_redirect()` etc.)
- Non-root accessibility would be nice?
- (Proposal on the topic for next Netdev conference in March 2020)

Statistics for Programs

Statistics for BPF programs: completion time and number of runs

- Activate (slight overhead) with:
  ```
  # sysctl -w kernel.bpf_stats_enabled=1
  ```
- Displayed by e.g. bpftool:



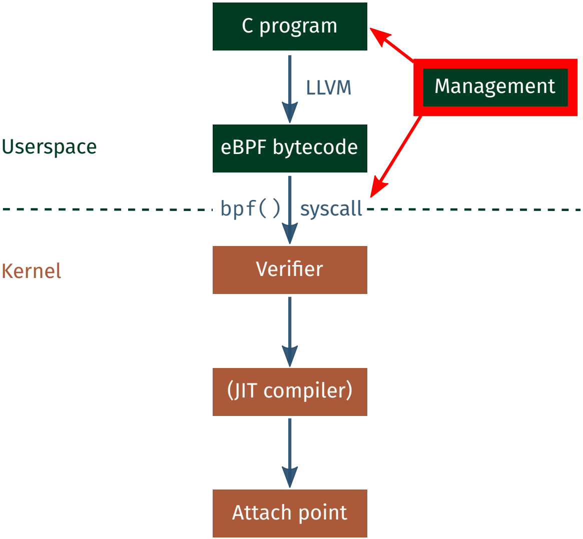
```
root@cbtest32  ~  bpftool prog show id 13
13: xdp  tag a04f5eef06a7f555 run_time_ns 12210 run_cnt 53
        loaded_at 2019-03-25T13:20:11+0000  uid 0
        xlated 16B  jited 61B  memlock 4096B
```

Debugging at Runtime: Miscellaneous

Additional tools that might be of use:

- **Perf** has support for annotating JIT-ed BPF programs (e.g. `perf top`)
- User space BPF machines: **uBPF**, rbpf
  (Features missing, no verifier, but can run with debugger)
- `tools/bpf/bpf_dbg.c` (legacy cBPF only)

eBPF Workflow



User Space Programming

Objectives:

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

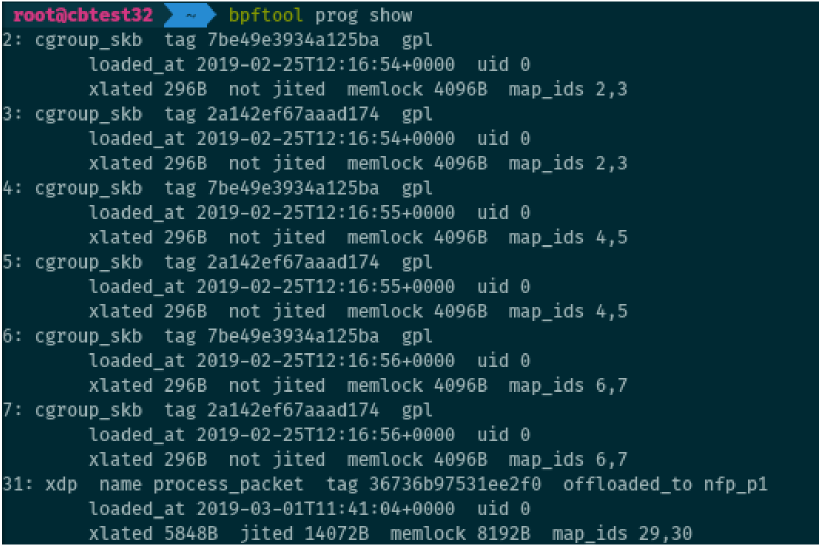
Solutions:

- We can rely on existing frameworks (bcc, bpftrace, libkefir...)
- Libraries for managing eBPF programs: **libbpf** (kernel tree, `tools/lib/bpf`), libbcc (bcc tools)
- Probe BPF-related kernel features with bpftool
- **strace**: support for `bpf()` system call
  ```
  strace -e bpf ip link set dev nfp_p0 xdpoffload obj prog.o
  ```
- **valgrind**: support for `bpf()` system call
  ```
  valgrind bpftool prog show
  ```

Getting Familiar With Bpftool

Bpftool: List Programs

List all BPF programs loaded on the system



```
root@cbtest32  ~  bpftool prog show
2: cgroup_skb  tag 7be49e3934a125ba  gpl
        loaded_at 2019-02-25T12:16:54+0000  uid 0
        xlated 296B  not jited  memlock 4096B  map_ids 2,3
3: cgroup_skb  tag 2a142ef67aaad174  gpl
        loaded_at 2019-02-25T12:16:54+0000  uid 0
        xlated 296B  not jited  memlock 4096B  map_ids 2,3
4: cgroup_skb  tag 7be49e3934a125ba  gpl
        loaded_at 2019-02-25T12:16:55+0000  uid 0
        xlated 296B  not jited  memlock 4096B  map_ids 4,5
5: cgroup_skb  tag 2a142ef67aaad174  gpl
        loaded_at 2019-02-25T12:16:55+0000  uid 0
        xlated 296B  not jited  memlock 4096B  map_ids 4,5
6: cgroup_skb  tag 7be49e3934a125ba  gpl
        loaded_at 2019-02-25T12:16:56+0000  uid 0
        xlated 296B  not jited  memlock 4096B  map_ids 6,7
7: cgroup_skb  tag 2a142ef67aaad174  gpl
        loaded_at 2019-02-25T12:16:56+0000  uid 0
        xlated 296B  not jited  memlock 4096B  map_ids 6,7
31: xdp  name process_packet  tag 36736b97531ee2f0  offloaded_to nfp_p1
        loaded_at 2019-03-01T11:41:04+0000  uid 0
        xlated 5848B  jited 14072B  memlock 8192B  map_ids 29,30
```

Bpftool: Dump Programs

Dump kernel-translated instructions

```
# bpftool prog dump xlated id 4
    0: (b7) r0 = 0
    1: (95) exit
```

Dump JIT-ed instructions

```
# bpftool prog dump jited id 4
   0:   push   %rbp
   1:   mov    %rsp,%rbp
   4:   sub    $0x28,%rsp
   b:   sub    $0x28,%rbp
   f:   mov    %rbx,0x0(%rbp)
  13:   mov    %r13,0x8(%rbp)
  [...]
  33:   mov    0x18(%rbp),%r15
  37:   add    $0x28,%rbp
  3b:   leaveq
  3c:   retq
```

Bpftool: Load, Attach Programs

Load a program:

```
# bpftool prog load <program> <pinned_path>
```

Attach to socket:

```
# bpftool prog attach <program> <attach type> <target map>
```

Or to cgroups:

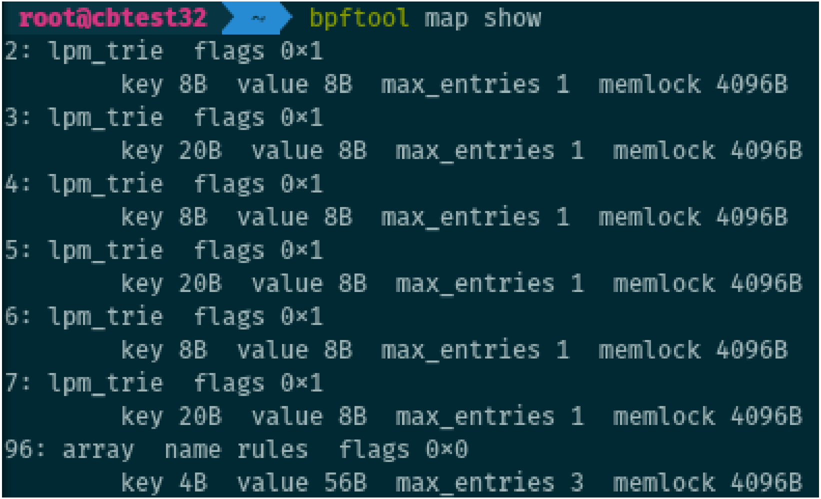
```
# bpftool cgroup attach <cgroup> <attach type> <program> [flags]
```

Or to tc, XDP:

```
# bpftool net attach <attach type> <program> <interface>
```

Bpftool: Show Maps

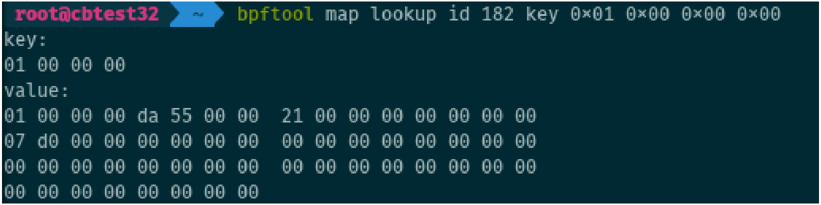
List all maps loaded on the system:



```
root@cbtest32  ~  bpftool map show
2: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
3: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
4: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
5: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
6: lpm_trie  flags 0x1
        key 8B  value 8B  max_entries 1  memlock 4096B
7: lpm_trie  flags 0x1
        key 20B  value 8B  max_entries 1  memlock 4096B
96: array  name rules  flags 0x0
        key 4B  value 56B  max_entries 3  memlock 4096B
```

Bpftool: Basic Map Operations

Retrieve first entry of array map (note: host endianness for the key):



```
root@cbtest32 ~ bpftool map lookup id 182 key 0x01 0x00 0x00 0x00
key:
01 00 00 00
value:
01 00 00 00 da 55 00 00  21 00 00 00 00 00 00 00
07 d0 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
```

Or dump all entries of a given map:

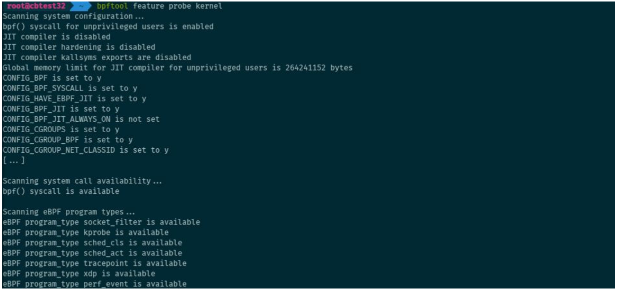
```
# bpftool map dump id 182
```

Update a map entry (even works for prog array maps used for tail calls)

```
# bpftool map update id 182 key 3 0 0 0 value 1 1 168 192
```

Bpftool: Probe Kernel Features

Check what BPF-related features are available on the system,
List program types, map types, BPF helpers available:



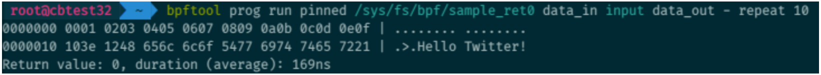
```
root@cbtest32  ~  bpftool feature probe kernel
Scanning system configuration...
bpf() syscall for unprivileged users is enabled
JIT compiler is disabled
JIT compiler hardening is disabled
JIT compiler kallsyms exports are disabled
Global memory limit for JIT compiler for unprivileged users is 264241152 bytes
CONFIG_BPF is set to y
CONFIG_BPF_SYSCALL is set to y
CONFIG_HAVE_EBPF_JIT is set to y
CONFIG_BPF_JIT is set to y
CONFIG_BPF_JIT_ALWAYS_ON is not set
CONFIG_CGROUPS is set to y
CONFIG_CGROUP_BPF is set to y
CONFIG_CGROUP_NET_CLASSID is set to y
[...]

Scanning system call availability...
bpf() syscall is available

Scanning eBPF program types...
eBPF program_type socket_filter is available
eBPF program_type kprobe is available
eBPF program_type sched_cls is available
eBPF program_type sched_act is available
eBPF program_type tracepoint is available
eBPF program_type xdp is available
eBPF program_type perf_event is available
```

Bpftool: Test Runs

Test-run programs with user-defined input data and context:



```
root@cbtest32  ~  bpftool prog run pinned /sys/fs/bpf/sample_ret0 data_in input data_out - repeat 10
0000000 0001 0203 0405 0607 0809 0a0b 0c0d 0e0f | ........ ........
0000010 103e 1248 656c 6c6f 5477 6974 7465 7221 | .>.Hello Twitter!
Return value: 0, duration (average): 169ns
```

More Bpftool!

Some other features:

- List programs per cgroup, per network interface, per tracing hook
- Can load several programs at once from single object file (`loadall`)
- Dump `bpf_trace_printk()` output: `bpftool tracelog`
- Dump data from event maps: `bpftool map event_pipe id 42`
- Generate skeleton header from `.o` file for management in user space
- Batch mode (`bpftool batch file <file>`)
- JSON support (`-j|--json` or `-p|--pretty`)
- Subcommand prefixes (`bpftool p d i 42`); Exhaustive bash completion
- And more!

See also https://twitter.com/qeole/status/1101450782841466880

Bpftool: Man Pages

More information:

- `man 8 bpftool`
- `man 8 bpftool-prog`, `man 8 bpftool-map`, etc.

Next Steps for eBPF Tooling and Debugging Facilities

Next Steps for eBPF Tooling

- BPF architecture
  - More modularity for easier debugging? (see BPF extension programs)
  - More informations on BPF objects from sysfs
  - Improvements for test runs
- Actual debugging process: Implement a step-by-step debugger
  - Run program in a VM, and freeze/unfreeze at each step?
  - Extend `BPF_PROG_TEST_RUN` interface?
  - Attach kprobes to every single instruction of program?
- Documentation
  - Update existing documentation
  - Create some troubleshooting guide/FAQ?

(Several of those ideas proposed for discussion at Netdev in March 2020)

Wrapping Up

eBPF programs do not run in user space: debugging is not trivial

But:

- Tooling is getting better and better: more tools, more complete
- We can dump insns at any stage of the process (**llvm-obdjump**, **bpftool**)
- We can print data (`bpf_trace_printk()`, perf event maps) at runtime
- We can do test runs in kernel, or to run in user space BPF frameworks
- BPF itself can be used to help debug verifier or other BPF programs

And hopefully more will come!

Thank you!

Questions?