

BPF and Linux Tracing Infrastructure

Elena Zannoni
elena.zannoni@oracle.com

Overview

- BPF main concepts and elements
- BPF and tracing
- BPF and Perf integration
- A few examples

BPF Concepts

BPF/cBPF/eBPF (Berkeley Packet Filters)

- Infrastructure that allows user defined programs to execute in kernel space.
- Programs written in C and translated into BPF instructions using Clang/LLVM, loaded in kernel and executed
- LLVM backend available to compile BPF programs
- 10 64-bit registers
- ~90 instructions implemented, including “bpf_call” for calling helper kernel functions from BPF programs
- Safety checks are performed by BPF program verifier in kernel
- Kernel has arm64, arm, mips, powerpc, s390, sparc x86 JITs
- Due to its history, you will find references to cBPF (classic), eBPF (extended), now simply called BPF

General Idea

- Load program into kernel and execute in kernel space
- There is a userspace program that does the housekeeping: compile the bpf program, load it, etc

BPF Programs

- Used by tracing and networking systems in kernel
- Different types of programs. Type determines how to interpret the context argument (mainly). Correspond to areas of BPF use in kernel
 - BPF_PROG_TYPE_SOCKET_FILTER
 - BPF_PROG_TYPE_SCHED_CLS
 - BPF_PROG_TYPE_SCHED_ACT
 - BPF_PROG_TYPE_XDP
 - BPF_PROG_TYPE_KPROBE
 - BPF_PROG_TYPE_TRACEPOINT
 - BPF_PROG_TYPE_PERF_EVENT (not yet in)
- BPF_PROG_RUN(ctx, prog): kernel macro that executes the program instructions. Has 2 arguments: pointer to context, array of bpf program instructions

Context

- Each BPF program is run within a context (ctx argument)
- Context is stored at start of BPF program (callee saved)
- Context may be used when calling helper functions, as their first argument
- Context provides data on which the BPF program operates:
 - (k)probes: it is the register set
 - Tracepoints: it is the format string
 - Networking filters: it is the socket buffer

BPF Helper Functions

- A BPF program can call certain helper functions.
- Helper Functions must be known: enum `bpf_func_id` values in `include/uapi/linux/bpf.h`
- Verifier uses info about each function to check safety of BPF calls
- Signature:
 - `u64 bpf_helper_function (u64 r1, u64 r2, u64 r3, u64 r4, u64 r5)`

BPF Helper Functions

- Map operations
 - `bpf_map_lookup_elem`
 - `bpf_map_update_elem`
 - `bpf_map_delete_elem`
 - [...]
- Tracing
 - `bpf_probe_read`
 - `bpf_trace_printk`
 - `bpf_ktime_get_ns`
 - [...]
- Networking
 - `bpf_skb_store_bytes`
 - `bpf_l3_csum_replace`
 - `bpf_l4_csum_replace`
 - [...]

BPF Safety

- Max 4096 instructions per program
- Stage 1 reject program if:
 - Loops and cyclic flow structure
 - Unreachable instructions
 - Bad jumps
- Stage 2 Static code analyzer:
 - Evaluate each path/instruction while keeping track of regs and stack states
 - Arguments validity in calls
- Non root usage: only for BPF programs of type `BPF_PROG_TYPE_SOCKET_FILTER`
- “Pointer verification” tests to avoid passing pointers back to userspace

Examples of Safety Checks/Errors

- Rn is invalid :invalid reg number
- Rn !read_ok :cannot read source op from register
- frame pointer is read only : cannot write into reg
- invalid access to map value, value_size=%d off=%d size=%d
- invalid bpf_context access off=%d size=%d
- invalid stack off=%d size=%d
- BPF_XADD uses reserved fields
- unsupported arg_type %d
- jump out of range from insn %d to %d
- back-edge from insn %d to %d
- unreachable insn %d
- BPF program is too large. Processed %d insn
- Reg pointer arithmetic prohibited
- R0 leaks addr as return value
- [...more...]

BPF Maps

- A map is generic memory allocated
- Transfer data from BPF programs to userspace or to kernel or vice versa; share data among many BPF programs
- A map is identified by a file descriptor returned by a `bpf()` system call in a userspace program that creates the map
- Attributes of a map: max elements, size of key, size of value
- Some types of maps: `BPF_MAP_TYPE_ARRAY`, `BPF_MAP_TYPE_HASH`, `BPF_MAP_TYPE_PROG_ARRAY`, `BPF_MAP_TYPE_PERF_EVENT_ARRAY`, `BPF_MAP_TYPE_STACK_TRACE`, `BPF_MAP_TYPE_CGROUP_ARRAY`,.....
- Maps operations (only specific ones allowed):
 - by user level programs (via `bpf()` syscall) or
 - by BPF programs via helper functions (which match the `bpf()` semantic)
- To close a map, call `close()` on the descriptor
- Maps (and BPF) can be persistent across termination of the process that created the map

bpf() System Call

- Single system call to operate both on maps and BPF programs
- Different types of arguments and behavior depending on the type of call determined by flag argument:
 - BPF_PROG_LOAD: verify and load a BPF program
 - BPF_MAP_CREATE: creates a new map
 - BPF_MAP_LOOKUP_ELEM: find element by key, return value
 - BPF_MAP_UPDATE_ELEM: find element by key, change value
 - BPF_MAP_DELETE_ELEM: find element by key, delete it
 - BPF_MAP_GET_NEXT_KEY: find element by key, return key of next element
 - BPF_OBJ_PIN, BPF_OBJ_GET: create persistent program (missing from the man page)

Connecting the Dots.... ...Usage Flows Examples

Generic Usage Flow

- From userspace program, load and run the bpf program, via the bpf() syscall, returns fd
- Cleanup/end: userspace program closes the fd corresponding to the bpf program
- BPF program can be specified in two ways:
 - Original Method: Write it directly using the BPF language as an array of instructions, and pass that to the bpf() syscall (all done in userspace program)
 - Better Method:
 - Write it using C, in a .c file. Use compiler directive in .c file to emit a **section** (will contain the program) with a specific name. Compile (with LLVM) into a .o file
 - The .o (Elf) file is then parsed by userspace program to find the section, the BPF instructions in it are passed to the bpf() syscall

Usage Flow with Maps

- If maps are used, they must be created by the userspace program and they must be associated with the BPF program.
- How?
- In same .c file used to specify the BPF program, use compiler directive to emit **section** called “maps” in the .o program.
- “maps” section contains map specification (type of map, size of element, size of key, max number of elements)
- “maps” section can contain multiple maps (easily parsed since all maps specifications are same size)
- In userspace program, parse .o file, to find “maps” section
- For each map listed in the section, create a new map, using bpf() syscall
- Then.....

Associating Maps with a BPF Program

-One missing piece of the puzzle: how does the running BPF program know where the map to operate on is?
- Remember:
 - A map is a file descriptor (returned by `bpf()` system call)
 - Map operations are done via calls to helper functions exclusively, from the BPF program
 - Map operations in `.c` kernel BPF program have address of map structure which is in its own “maps” section.
- Ultimately you want to have the fd of the map in call instruction field of the BPF program, prior to execution
- So process relocation of BPF program, by walking the instructions and inserting the fd of the map in the proper field of calls to helper functions

Elf Section Naming

- Names used for the BPF program sections, must follow convention:
 - “kprobe/<event name>”
 - “kretprobe/<event name>”
 - “tracepoint/<subsystem name>/<event name>”
 - “socket<name>”
 - “maps”: map structures (not actual maps!)
 - “license”: whether the BPF program is GPL
 - “version”: kernel version
 - “<subset of perf command line syntax>” (see later slides)
- Important because type of BPF program is determined by the section names

There is Some Good News...

- ... this is fairly complex... however...
- A lot of the workflow is already collected into a generic userspace program, **samples/bpf/bpf_load.c** and **bpflib.c** in the kernel tree.
- Also now integrated in libbpf available in **tools/lib/bpf/libbpf**
- Takes care of parsing the Elf file, creating and handling maps, adding the probe event, loading the BPF program, associating the program to the event (for tracing), etc
- For use with perf: **tools/perf/util/bpf_loader.c** collects the common workflow, and also takes care of calling llvm to do compilation.

BPF and Tracing

Tracing and BPF

- Goals:
 - Place probe at function “foo” in the kernel. When probe fires, execute program “bpf_prg”
 - Place probe at function “foo” in user executable “exec”. When probe fires, execute program “bpf_prg”
 - Attach “bpf_prg” program to static kernel tracepoint. When tracepoint is executed, execute program “bpf_prg”
 - Possibly, do all the above from within perf

Some Kernel Changes in Tracing for BPF

- ioctl command added to associate bpf program to kprobe event
PERF_EVENT_IOC_SET_BPF
- Added a field to tp_event field of the perf_event structure:
tp_event->prog
- tp_event is a structure defining what to do when the event happens
- This is set by the ioctl value above
- Here it means: call trace_call_bpf(prog, pt_regs)
- Which in turns calls BPF_PROG_RUN(prog, pt_regs /*ctx*/) with pt_regs as the context (see earlier slides)
- More....

Retrieving Data

- How to retrieve data collected during tracing
- Can read the `<tracefs>/trace_pipe` file from userspace as normal
- Can read memory `bpf_probe_read`
- Can retrieve registers values (they are the `ctx`)
- Can read from maps

How to specify a tracing event with BPF

- Write the `bpf_prg` in C.
 - Use compiler directive to emit an Elf section into the `.o` file:
 - “`kprobe/foo`” (or “`kretprobe/foo`”)
- OR
- “`tracepoint/foo/bar`”

This section will contain the BPF program instructions

- Compile the `.c` into a bpf program with LLVM (a `.o` file)

Tracepoints and BPF

- In userspace:
- Look for section named “tracepoint/foo/bar” in Elf file, where foo is the kernel subsystem, bar is the tracepoint name
- There will be a subdirectory for that tracepoint:
<tracefs>/events/foo/bar/*
- File <tracefs>/events/foo/**id** contains the tracepoint id (an integer)
- Create an event structure “attr” with type PERF_TYPE_TRACEPOINT and with config.id the value in <tracefs>/events/foo/id
- Load program bpf_prg instructions into kernel (use bpf_prog_load() library call), returns **fd**
- Create the event: **efd** = perf_event_open(&attr,)
- Enable the event: ioctl(**efd**, PERF_EVENT_IOC_ENABLE, 0);
- Attach prog to the event: ioctl(efd, PERF_EVENT_IOC_SET_BPF, **fd**);
- BPF prog will run when tracepoint is executed

Dynamic probes and BPF

- In userspace:
- Look for section named “kprobe/foo” in Elf file. If found:
- Create a new probe: `echo 'p:foo foo' >> <tracefs>/kprobe_events`
- There will be a subdirectory for that new probe:
`<tracefs>/events/kprobes/foo/*`
- File `<tracefs>/events/kprobes/foo/id` contains the probe id
- Create an event structure “attr” with `type = PERF_TYPE_KPROBE` and with `config.id` the value in `<tracefs>/events/kprobes/foo/id`
- Load program `bpf_prg` instructions into kernel (via `bpf(BPF_PROG_LOAD...)` call or library wrapper `bpf_prog_load()`), returns **fd**
- Create the event: `efd = perf_event_open(&attr,)`
- Enable the event: `ioctl(efd, PERF_EVENT_IOC_ENABLE, 0);`
- Attach prog to the event: `ioctl(efd, PERF_EVENT_IOC_SET_BPF, fd);`
- BPF prog will run when probe fires (via `BPF_PROG_RUN()` called by `kprobe_dispatcher`)

Uprobes and BPF

- BPF programs can specify user space probes
- Section name contains keyword “exec”:

```
SEC("exec=<filename>\n;"  
    "<progname>=foo")
```

Specify executable name, then name of the bpf program, and location of the probe point. Separate by a ‘;’

BPF and Perf Integration

- Use BPF programs to filter events recorded by perf
- Syntax:
 - Perf record --event bpf-prog.c <command>
- Introduces userspace library libbpf and perf utilities to do the complex housekeeping done in the bpf_load.c example file
- tools/lib/bpf/libbpf.c and tools/perf/util/bpf_loader.c
- Same mechanism of using section names to indicate specific info:
 - “maps”, “license”, “version” used as in previous case
 - Program names are the section names. No need to have “kprobes” or “kretprobes” prefixes.
- Plus more complex section names (see slide)

ELF Section Naming “Overloaded”

- Many types of complex strings allowed as sections names, mimic “perf probe” command syntax
- Tracepoint specification: “subsystem:tracepointname”
- Dynamic probe specification “progrname=kernfunc”
- Dynamic probe with up to 3 arguments “progrname=kernfunc arg1 arg2 arg3”
- “force=...”, same as -f, --force . Forcibly add events with existing name.
- “module=...” same as -m, --module=MODNAME module name in which perf-probe searches probe points
- “inlines=...”, same as --no-inlines . Search only for non-inlined functions.
- “exec=...”, same as -x, --exec=PATH Specify path to the executable or shared library file for user space tracing.
- Separated by ‘;’

BPF + tracepoint example (example.c)

```
SEC("raw_syscalls:sys_enter")
int func(void *ctx)
{
    u64 id = *((u64 *)(ctx + 8)); → get the ID value and filter on it.
    if (id == 1)
        return 1;
    return 0;
}
```

In the file: <tracefs>/events/raw_syscalls/sys_enter/format we have:

format:

```
field:unsigned short common_type;    offset:0;    size:2; signed:0;
field:unsigned char common_flags;    offset:2;    size:1; signed:0;
field:unsigned char common_preempt_count;    offset:3;    size:1; signed:0;
field:int common_pid;    offset:4;    size:4; signed:1;
field:long id;    offset:8;    size:8; signed:1;
field:unsigned long args[6];    offset:16;    size:48;    signed:0;
```

% perf record -e ./example.c cat ./myfile1 /dev/null → will report only the events when the syscall has ID 1 (write).

BPF + Dynamic probe example

```
SEC("_write=sys_write")
int _write(void *ctx)
{
    return 1;
}
```

Defining a probe called `_write`, at the function `sys_write` in the kernel.

Probe fires when `sys_write` is called

The name of the probe can be anything, as long as it matches the program name.

BPF + probe with arguments example

```
SEC("lock_page=__lock_page page->flags")
    int lock_page(struct pt_regs *ctx, int err, unsigned long
flags)
    {
        return 1;
    }
```

- The bpf program has ctx as first argument as usual
- Second argument is a flag to indicate valid access to pointer
- There can be max 3 arguments for the function
- bpf_prog_func (ctx, err, arg1, arg2, arg3)

Thanks!!
Questions??