An intro to using eBPF to filter packets in the Linux kernel

Learn about using Extended BPF, an enhancement to the original Berkeley Packet Filter, to filter packets in the Linux kernel.

In 1992, Steven McCanne and Van Jacobson from Lawrence Berkeley Laboratory proposed a solution for BSD Unix systems for minimizing unwanted network packet copies to user space by implementing an in-kernel packet filter known as Berkeley Packet Filter (BPF). In 1997, it was introduced in Linux kernel version 2.1.75.

BPF's purpose was to filter all unwanted packets as early as possible, so the filtering mechanism had to be shifted from user space utilities like **tcpdump** to the in-kernel virtual machine. It sends a group of assembly-like instructions for filtering necessary packets from user space to kernel by a system call **bpf()**. The kernel statically analyzes the programs before loading them and makes sure they don't hang or harm a running system.

The BPF machine

More Linux resources

- Linux commands cheat sheet (https://developers.redhat.com/cheat-sheets/linux-commands-cheat-sheet/?intcmp=70160000000h1jYAAQ&utm_source=intcallout&utm_campaign=linuxcontent)
- Advanced Linux commands cheat sheet (https://developers.redhat.com/cheat-sheets/advanced-linux-commands/?intcmp=70160000000h1jYAAQ&utm_source=intcallout&utm_campaign=linuxcontent)
- Free online course: RHEL Technical Overview (https://www.redhat.com/en/services/training/rh024-red-hat-linux-technical-overview?intcmp=70160000000h1jYAAQ&utm_source=intcallout&utm_campaign=linuxcontent)
- Linux networking cheat sheet (https://opensource.com/downloads/cheat-sheet-networking?intcmp=70160000000h1jYAAQ&utm_source=intcallout&utm_campaign=linuxcontent)
- SELinux cheat sheet (https://opensource.com/downloads/cheat-sheet-selinux?intcmp=70160000000h1jYAAQ&utm_source=intcallout&utm_campaign=linuxcontent)
- Linux common commands cheat sheet (https://opensource.com/downloads/linux-common-commands-cheat-sheet?intcmp=70160000000h1jYAAQ&utm_source=intcallout&utm_campaign=linuxcontent)
- What are Linux containers? (https://opensource.com/resources/what-are-linux-

containers?intcmp=70160000000h1jYAAQ&utm_source=intcallout&utm_campaign=linuxcontent)

- Our latest Linux articles (https://opensource.com/tags/linux?intcmp=70160000000h1jYAAQ&utm_source=intcallout&utm_campaign=linuxcontent)

The BPF machine abstraction consists of (http://www.tcpdump.org/papers/) an accumulator, an index register (x), a scratch memory store, and an implicit program counter. It has a small set of arithmetic, logical, and jump instructions. The accumulator is used for arithmetic operations, while the index register provides offsets into the packet or the scratch memory areas. Here's an example of a small BPF program written in BPF bytecode:

```
ldh     [12]
jeq     #ETHERTYPE_IP, l1, l2
l1:     ret     #TRUE
l2:     ret     #0
```

The **ldh** instruction loads a half-word (16-bit) value in the accumulator from offset 12 in the Ethernet packet, which is an Ethernet-type field. If it is not an IP packet, **0** will be returned, and the packet will be rejected.

BPF just-in-time compiler

A just-in-time (JIT) compiler was introduced into the kernel (https://lwn.net/Articles/437884/) in 2011 to speed up BPF

bytecode execution. This compiler translates BPF bytecode into a host system's assembly code. Such a compiler exists for x86-64, SPARC, PowerPC, ARM, ARM64, MIPS, and System 390 and can be enabled through **CONFIG_BPF_JIT**.

eBPF machine

Extended BPF (eBPF) is an enhancement over BPF (which is now called cBPF, which stands for classical BPF) with more resources (https://www.kernel.org/doc/Documentation/networking/filter.txt), such as 10 registers and 1-8 byte load/store instructions. Whereas BPF has forward jumps, eBPF has both backwards and forwards jumps, so there can be a loop, which, of course, the kernel ensures terminates properly. It also includes a global data store called **maps**, and this maps state persists between events. Therefore eBPF can also be used for aggregating statistics of events. Further, an eBPF program can be written in C-like functions, which can be compiled using a GNU Compiler Collection (GCC)/LLVM compiler. eBPF has been designed to be JIT'ed with one-to-one mapping, so it can generate very optimized code that performs as fast as natively compiled code.

eBPF and tracing review

Upstream kernel development

Traditional built-in tracers in Linux are used in a post-process

manner, where they dump fixed-event details, then userspace tools like **perf** or **trace-cmd** can post processes to get required information (e.g., **perf stat**); however, eBPF can prepare user information in kernel context and transfer only needed information to user space. So far, support of **kprobes**, **tracepoints**, and **perf_events** filtering using eBPF have been implemented in the upstream kernel. They have been supported with **Arch x86-64**, **AArch64**, **S390x**, **PowerPC 64**, and **SPARC64**.

For more information, look at these Linux kernel files:

- kernel/bpf/ (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/bpf)
- kernel/trace/bpf_trace.c (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/trace/bpf_trace.c)
- kernel/events/core.c (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/events/core.c)

Userspace development

Userspace tools have been developed for both the in-kernel tree and the out-of-kernel tree. Take a look at these files and directories for more on the upstream kernel for eBPF use:

- tools/lib/bpf (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/lib/bpf)

- tools/perf/util/bpf-loader.c (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/perf/util/bpf-loader.c)
- samples/bpf/ (https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/samples/bpf)

BCC (https://github.com/iovisor/bcc) is another out-of-kernel tree tool that has efficient kernel tracing programs for specific usage (such as **funccount**, which counts functions matching a pattern).

Perf has also a BPF interface that can be used to load eBPF objects into kernel.

eBPF tracing: User space to kernel space flow

BPF system call and BPF maps are two useful entities that can interact with the eBPF kernel.

BPF system call

A user can interact with the eBPF kernel using a **bpf()** system call whose prototype is:

```
int bpf(int cmd, union bpf_attr *attr, unsigned int siz
```

Following is a summary of those arguments; see man page

BPF (http://man7.org/linux/man-pages/man2/bpf.2.html) for more details.

- **cmd** can be any defined **enum bpf_cmd**, which tell the kernel about management of the map area (e.g., creating, updating, deleting, finding an element, attaching or detaching a program, etc.).
- **attr** can be a user-defined structure that can be used by its respective command.
- **size** is the size of **attr**.

BPF maps

eBPF tracing calculates the stats in the kernel domain itself. We will need some type of memory/data structure within the kernel to create such stats. Maps are a generic data structure that store different types of data in the form of key-value pairs. They allow sharing of data between eBPF kernel programs, and between kernel and user-space applications.

Important attributes for maps include:

- Type (map_type)
- Maximum number of elements (max_entries)
- Key size in bytes (key_size)
- Value size in bytes (value_size)

There are different types of maps (e.g., hash, array, program

array, etc.), that are chosen based on the use or need. For example, if the key is a string (or not from an integer series), then a hash map can be used for faster lookup; however, if the key is like an index, then an array map will provide the fastest lookup method.

A key cannot be bigger than **key_size**, it cannot store a value bigger than **value_size**, and **max_entries** is the maximum number of key-value pairs that can be stored within a map.

Here are two important commands to note:

- **BPF_PROG_LOAD:** Following are important attributes for this program:

    prog_type: program types useful for tracing:

        BPF_PROG_TYPE_KPROBE
        BPF_PROG_TYPE_TRACEPOINT
        BPF_PROG_TYPE_PERF_EVENT

    insns: a pointer to **struct bpf_insn** that has BPF instructions to be executed by an in-kernel BPF virtual machine

    insn_cnt: total number of instructions present at **insns**

    license:string, which must be GPL-compatible to

call helper functions marked **gpl_only**

kern_version: version of kernel tree

- **BPF_MAP_CREATE:** Accepts attributes as discussed in the BPF maps section, creates a new map, then returns a new file descriptor that refers to the map. Returned **map_fd** can be used for lookup or to update map elements with commands such as **BPF_MAP_LOOKUP_ELEM**, **BPF_MAP_UPDATE_ELEM**, **BPF_MAP_DELETE_ELEM**, **or** **BPF_MAP_GET_NEXT_KEY**. These map-manipulation commands accept an attribute with **map_fd**, key, and value.

To understand how it works, look up this standalone eBPF demo (https://github.com/pratyushanand/learn-bpf) on GitHub; it does not need any other eBPF library code. It has a small library to load different sections of BPF kernel code (**bpf_load.c** (https://github.com/pratyushanand/learn-bpf/blob/master/bpf_load.c)) and then wrapper functions on top of the **bpf()** system call (**bpf.c** (https://github.com/pratyushanand/learn-bpf/blob/master/bpf/bpf.c)) to manipulate the map and load kernel BPF code. When we compile this code, we get two executables: **memcpy_kprobe** and **memcpy_stat**.

- **memcpy_kprobe:** For each application, we have one **_kern** file and another **_user** file. The **_kern** file has a function, **int bpf_prog1(struct pt_regs *ctx)** (https://github.com/pratyushanand/learn-bpf/blob/master/memcpy_kprobe_kern.c#L9). This function is executed in the kernel so it can access kernel variables and functions. **Memcpy_kprobe_kern.c** (https://github.com/pratyushanand/learn-bpf/blob/master/memcpy_kprobe_kern.c) has three section mappings for program, license, and version, respectively. Data from these sections are part of attributes of the system call **bpf(BPF_PROG_LOAD,...)**, and then the kernel executes loaded BPF instructions as per the **prog_type** attribute. So, BPF code in **memcpy_kprobe_kern.c** will be executed when a **kprobe** instrumented at the entry of kernel **memcpy()** is hit. When this BPF code is executed, it will read the third argument of **memcpy()**, such as the size of the copy, and will then print one statement for **memcpy size** in trace buffer. **Memcpy_kprobe_user.c** (https://github.com/pratyushanand/learn-bpf/blob/master/memcpy_kprobe_user.c) loads the kernel program and keeps on reading the trace buffer to show what kernel eBPF program is writing into it.

- **memcpy_stat:** This prepares stats of **memcpy()** copy size in the kernel itself. **Memcpy_stat_kern.c** (https://github.com/pratyushanand/learn-bpf/blob/master/memcpy_stat_kern.c) has one more section, as **maps.**

bpf_prog1() reads **memcpy()** sizes and updates the map table. The corresponding userspace program **memcpy_stat_user.c** (https://github.com/pratyushanand/learn-bpf/blob/master/memcpy_stat_user.c) reads the map table every two seconds and prints stats on the console.

These simple examples explain how to write kernel eBPF code for kernel tracing and statistics preparation.

If you have used eBPF and have advice to share, please leave a note in the comments.