bpf-bindsnoop.py

Christian Cordiviola

November 23, 2021

Introduction to eBPF

eBPF, as Extended Berkeley Packet Filter, is a kernel technology (for Linux 4.x and later) that can run sandboxed programs in an OS kernel. The OS guarantees safety and execution efficiency as if natively compiled, using a JIT (Just-in-Time) compiler and a verification engine. Usage of eBPF today includes:

- Security
- Tracing
- Networking
- Kernel Observability & Monitoring

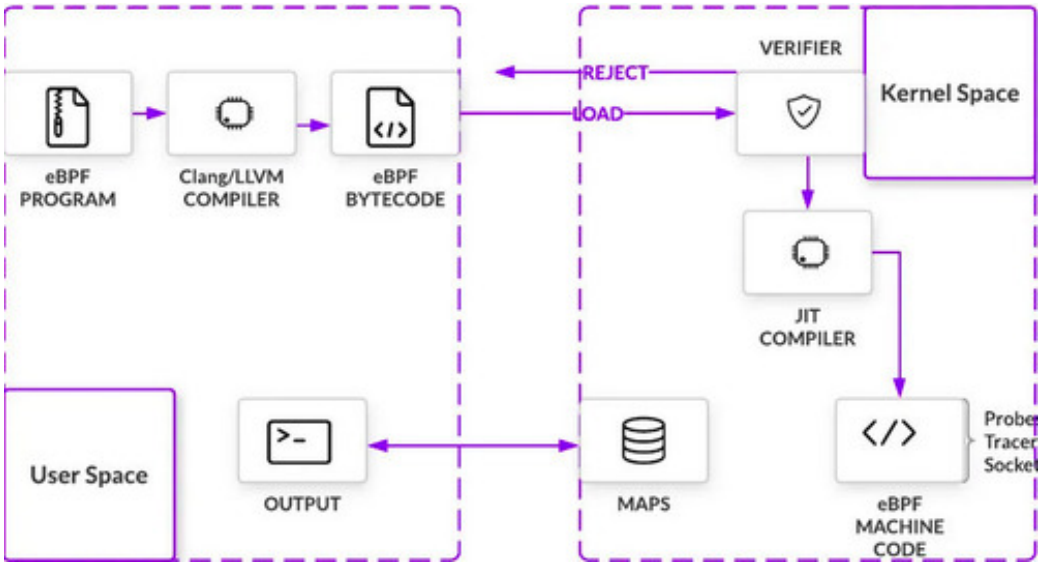
eBPF allows regular userspace applications to package logic to be executed within the Linux kernel as bytecode. These are called eBPF programs, written in C, and are produced by eBPF compiler toolchain called BCC (BPF Compiler Collection).
However, before being loaded into the kernel, an eBPF program must pass a set of checks. The verification engine executes the eBPF program within a virtual machine, allowing the verifier to traverse the potential execution paths the program may take when executed within the kernel, making sure it does run to completion without any loops, which would cause kernel lockup. Only if all checks pass, the eBPF program is loaded and compiled into the kernel and starts waiting for the right hook: once triggered, the bytecode executes.

Using eBPF

eBPF programs execute in an event-driven environment: they are triggered by kernel hooks. Hook locations include:

- System Calls
- Function entry/exit
- Network Events
- Kprobes and uprobes

When eBPF programs are triggered at their hook points, they can call helper functions that can perform a wide variety of tasks. Helpers, however, must be defined by the kernel, meaning there is a whitelist of calls eBPF programs can make. Additionally, eBPF programs make use of eBPF maps to keep state between invocations and to share data with user-space applications. The kernel expects all eBPF programs to be loaded as bytecode, which is created by using higher-level languages. The most popular toolchain for writing and debugging eBPF programs is BCC, based on LLVM and CLANG.



After verification, the bytecode is JIT (Just-in-Time) compiled into native machine code. eBPF is 64-bit encoded with 11 total registers, allowing it to map the hardware for x86_64, ARM and arm64 architecture among others. The important takeaway is that eBPF unlocks access to kernel level events, without the typical restrictions found when changing kernel code directly.

Summarizing, eBPF works by:

- Compiling eBPF programs into bytecode
- Verifying programs execute safely in a VM before being loaded at the hook point
- Attaching programs to hook points within the kernel that are triggered by specific events
- Compiling at runtime for maximum efficiency
- Calling helper functions to manipulate dat when a program is triggered
- Using maps (key-value pairs) to share data between user and kernel space

bindsnoop.py

The tool utilizes eBPF to trace all IPv4 and IPv6 bind() attempts, even if they fail or the socket is not willing to accept(), by attaching an eBPF program to the kernel hooks **inet_bind()** and **inet6_bind**. This is known as attaching a kprobe (kernel probe) to the hooks.

Syntax: *kprobe__kernel_function_name*

kprobe__ is a special prefix that creates a kprobe (dynamic tracing of a kernel function call) for the kernel function name provided as the remainder. kprobes can also be used by declaring a normal C function, then using the Python *BPF.attach_kprobe()* to associate it with a kernel function. This is done by using python3-bpfcc, the Python wrapper package for BCC.
Arguments are specified on the function declaration:

*kprobe__kernel_function_name(struct pt_regs *ctx [, argument1 ...])*

The first argument is always struct pt_regs *, the remainder are the arguments to the function (they don't need to be specified, if you don't intend to use them).

```
88 int kprobe__inet_bind(struct pt_regs* ctx, struct socket* socket, const struct sockaddr* addr,
   int addrlen){
89
90     return inetbind(ctx,socket,addr,addrlen);
91
92 }
93
94 int kprobe__inet6_bind(struct pt_regs* ctx, struct socket* socket, const struct sockaddr* addr,
    int addrlen){
95
96     return inetbind(ctx,socket,addr,addrlen);
97
```

The eBPF program gathers data, such as:

- pid of binding process
- IPv4/v6 bound address
- bound port
- protocol (TCP/UDP only)

from the kernel call's arguments, the struct sock* and the struct sockaddr*, and fills the data_t structure with said data, then sends them to user space by writing the structure in an eBPF map.

```
17 //structure to send information to user space
18 struct data_t {
19
20     u64 ts;
21     u64 pid;
22     char comm[TASK_COMM_LEN];
23     u64 port;
24     u32 v4addr;
25     u64 proto;
26     u64 version;
27     u64 v6addr[2];
28
29 };
```

The handle:

```
34 static int inetbind(struct pt_regs* ctx, struct socket* sock, const struct sockaddr* addr, int addrlen){
35
36     //initialize data structures
37     struct data_t data;
38     __builtin_memset(&data,0,sizeof(data));
39     struct sock* sk = sock->sk;
40     struct inet_sock* inetsk = (struct inet_sock*) sk;
41
42     //get time
43     data.ts = bpf_ktime_get_ns() / 1000;
44
45     //get pid
46     data.pid = bpf_get_current_pid_tgid();
47
48     //get comm
49     bpf_get_current_comm(data.comm, TASK_COMM_LEN);
50
51     //get protocol with direct read workaround
52     u8 protocol = 0;
53     bpf_probe_read(&protocol,1,(u8*)&sk->sk_gso_max_segs - 3);
54
55     //TCP or UDP?
56     if(protocol == IPPROTO_TCP)
57         data.proto = 1;
58     else if(protocol == IPPROTO_UDP)
59         data.proto = 2;
60
61     //get socket IP family
62     u16 family = sk->__sk_common.skc_family;
63
64     //get IPv4 address and port
65     if(family == AF_INET){
66         struct sockaddr_in *in_addr = (struct sockaddr_in *)addr;
67         data.v4addr = in_addr->sin_addr.s_addr;
68         data.port = in_addr->sin_port;
69         data.port = ntohs(data.port);
70         data.version = 1;
71     }
72
73     //or get IPv6 address and port
74     else if(family == AF_INET6){
75         struct sockaddr_in6 *in6_addr = (struct sockaddr_in6 *)addr;
76         bpf_probe_read(data.v6addr,sizeof(data.v6addr),in6_addr->sin6_addr.s6_addr);
77         data.port = in6_addr->sin6_port;
78         data.port = ntohs(data.port);
79         data.version = 2;
80     }
81
82     //submit event to user space
83     events.perf_submit(ctx,&data, sizeof(data));
84     return 0;
85
86 };
```

This function is called whenever one of the hookpoints, **inet_bind()** or **inet6_bind()**, are triggered. Fills the data_t structure with the necessary information and submits it to the map, so it can be accessed from user-space.

The user-space side of the program, written in Python, initializes eBPF, then polls the map waiting for a new bind() event. Whenever said event happens, the data_t structure is retrieved, parsed and the output is built.

```
123 #initialize BPF
124 b = BPF(text = prog)
125 b["events"].open_perf_buffer(print_event)
126
127 #print headers
128 print("%-10s %-20s %-12s %-8s %-40s" % ("PID","COMM","PROTO","PORT","ADDR"))
129
130 #read events
131 while 1:
132     try:
133         b.perf_buffer_poll()
134     except KeyboardInterrupt:
135         exit()
```