Dodging deadly signal using eBPF

Esteban Blanc

EPITA

November 06, 2021

Context

I was reading `man 7 signal`:

> `The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.`

Does it still holds?

This was writen at least in 2004, a dark time without modern fancy features. . .
This might not be true anymore, let's see

Usual way of catching signals - signal

```c
void sig_handler(int signum) {
    printf("Catched a SIGINT\n");
    _exit(1);
}
int main (void) {
    signal(SIGINT, sig_handler);
    sleep(2);
    return 0;
}
```

Defects

No SIGKILL, no SIGSTOP

Usual way of catching signals - sigaction

```c
void sig_handler(int signum) {
    printf("Catched a SIGINT\n");
    _exit(1);
}
int main (void) {
    struct sigaction s = {.sa_handler = sig_handler};
    if (sigaction(SIGINT, &s, NULL) < 0) {
        return 2;
    }
    sleep(2);
    return 0;
}
```

Defects

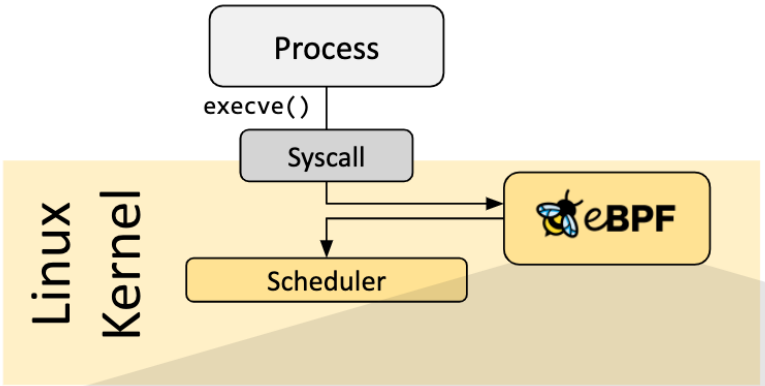
No SIGKILL, no SIGSTOP

eBPF

But I want to catch SIGKILL and SIGSTOP :(...

...eBPF to the rescue

How does eBPF works



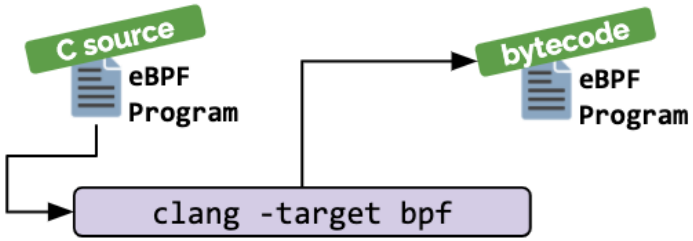

```c
int syscall__ret_execve(struct pt_regs *ctx)
{
        struct comm_event event = {
                .pid = bpf_get_current_pid_tgid() >> 32,
                .type = TYPE_RETURN,
        };

        bpf_get_current_comm(&event.comm, sizeof(event.comm));
        comm_events.perf_submit(ctx, &event, sizeof(event));

        return 0;
}
```

How are eBPF programs written?

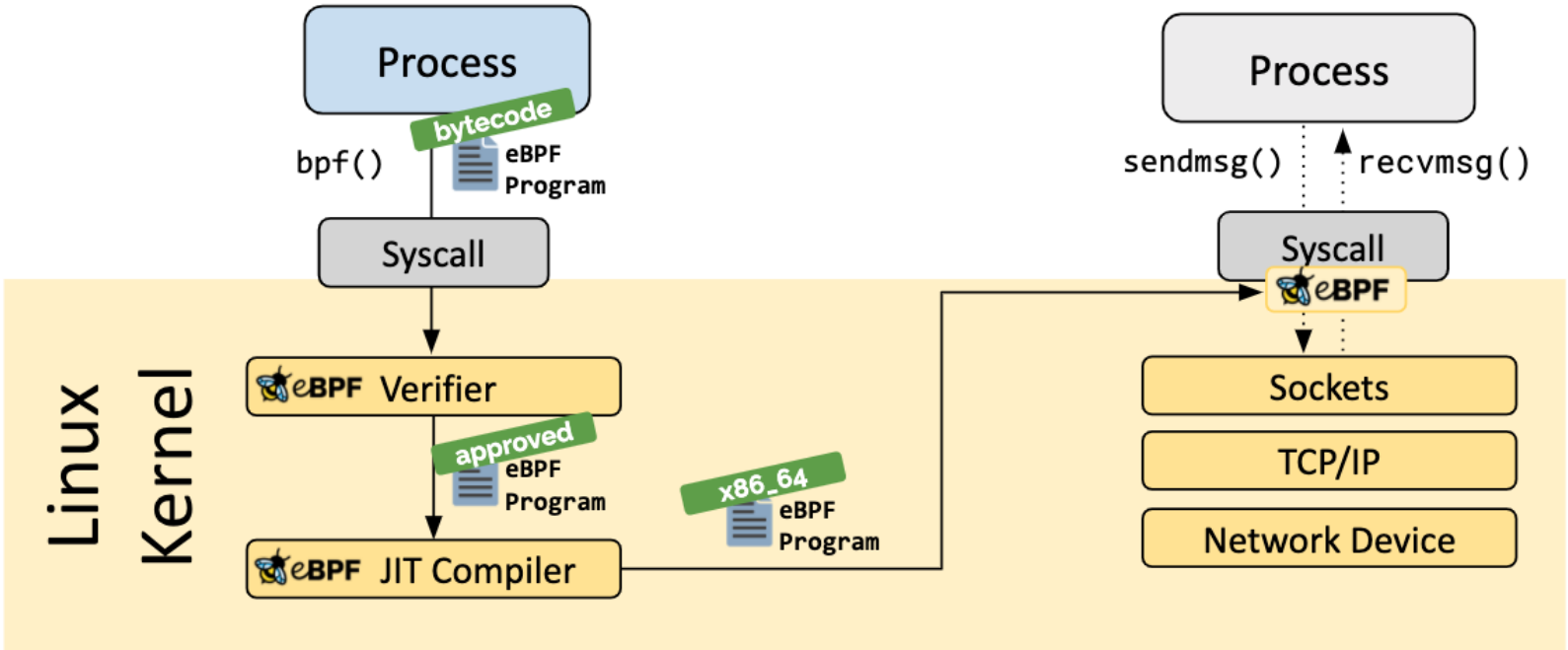
Framework like Cilium, bcc or bpftrace can be used



Verification step

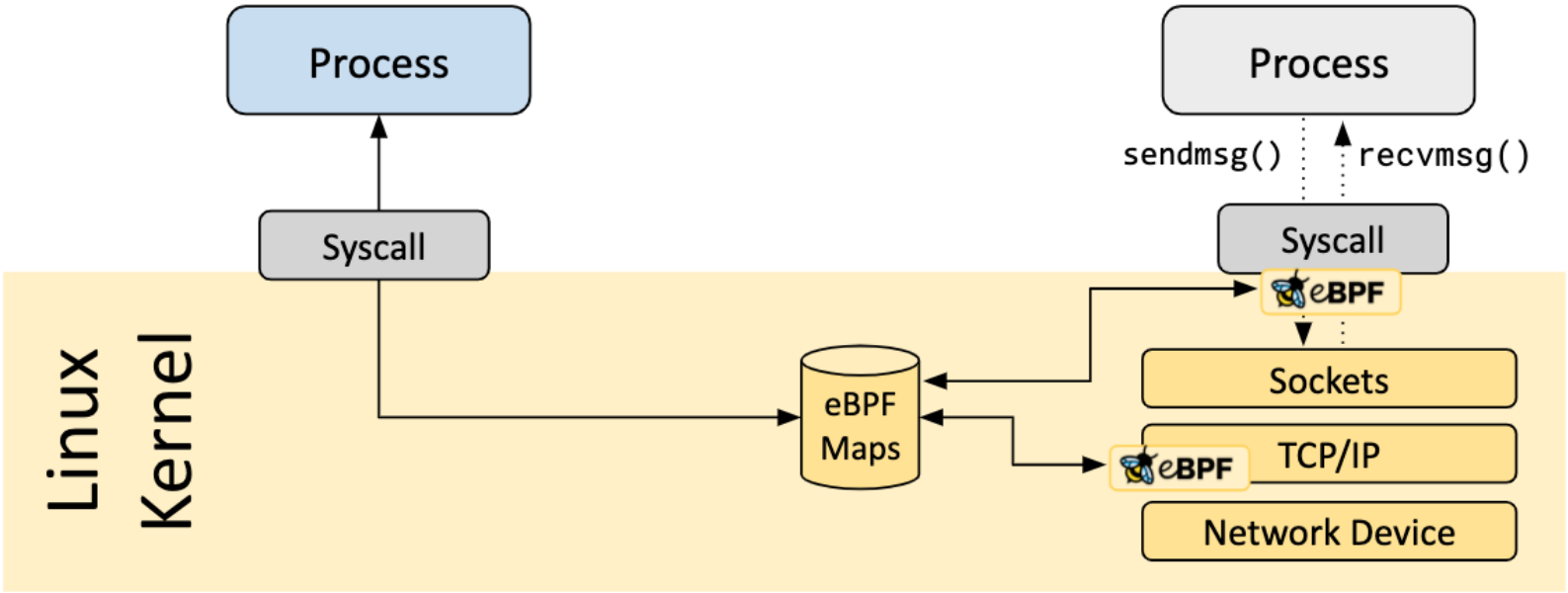
Check if the program is safe to use:

- No crash
- No infinite loop
- Not too long, inferior to BPF_MAXINSNS
- No out of bounds access
- No undefined behavior



Communication between eBPF and userland

You need to be able to communicate with userland process
Done with eBPF Maps



Types of maps

- BPF_MAP_TYPE_HASH
- BPF_MAP_TYPE_ARRAY
- ...

Hook points

- kprobe: kernel function call
- kretprobes: kernel function return
- tracepoints: no stable ABI
- uprobe: kernel probe but for userland
- ...

Where to place our hook

- In the kernel function that send a signal to a process?
- In the `kill` syscall?

The kill syscall

man 3 kill:

```
int kill(pid_t pid, int sig);
The kill() system call can be used to send any signal
to any process group or process.
```

The plan

1. Attach to kill kprobe
2. Find a way to return early form the syscall
3. Enjoy!

Hooking to the kill syscall

We will use `bpftrace` for the POC phase

```
$ bpftrace -e "kprobe:__x64_sys_kill {
    printf(\"Kill called\n\")
  }"
```

It's demo time

It's demo time!

Blocking the signal

We are hooked!

eBPF provide an override method to abort the called function with a specified return value

Wait whaaaaaat??

Security concern

We are essentially aborting a syscall here!

Kernel configuration

`CONFIG_BPF_KPROBE_OVERRIDE`

Only function marked as `ALLOW_ERROR_INJECTION` can be override

Hooking to the kill syscall, override edition

```
# blocksig.sh

bpftrace -e "kprobe:__x64_sys_kill {
    if (arg1 == $1) {
        printf(\"Kill called\n\");
        override(0);
    }
}"
```

It's demo time again!

```
$ # Without blocksig.sh
$ ping skallwar.fr > /dev/null &
[1] 371628
$ kill -9 371628
[1]+  Killed     ping skallwar.fr > /dev/null

$ # With blocksig.sh
$ ping skallwar.fr > /dev/null &
[1] 315629
$ sudo ./blocksig.sh 315629 &
Attaching 1 probe...
$ kill -9 315629
Signal blocked for 315629
```

Sadly it's still demo time...

```
$ ping 8.8.8.8 > /dev/null &
[1] 62705
$ sudo ./blocksig.sh 62705 > /dev/null &
[2] 62777
$ killall bpftrace
[2]+  Killed     ./blocksig.sh 62705
$ kill -9 62705
[1]+  Killed     ping 8.8.8.8 > /dev/null
```

Fatal flaw

It will not work like this

```
16727 /bin/sh ./blocksig.sh 16700
16728 └─ bpftrace -e kprobe:__x64_sys_kill { if (arg1 == 16700 || arg1 == 16727)
```

Shell problem

- Every command is a new program
- We need to know the pid of our shell script
- Bpftrace needs to know it's pid before being run

BCC to the rescue

Time to rewrite with a more powerful tool

Advantages

- It's in python
- Good abstractions
- No more pid problems

Hooking to the kill syscall, BCC edition

```
#!/usr/bin/python
from bcc import BPF

BPF(text='int kprobe__sys_kill(void *ctx) { \
            bpf_trace_printk("Hello, World!\\n"); \
            return 0; \
        } \
').trace_print()
```

The plan updated

1. Parse arguments (pids and signals to catch)
2. Fill 2 maps with pids and signals
3. Attach to kill kprobe
4. Override returns value
5. Enjoy more!

Maps in BCC

```
// Syntax: BPF_HASH(name, key_type, value_type)
BPF_HASH(pids, int, u8);
BPF_HASH(sigs, u8, 65);
```

Our Python code

```
def initialize_bpf(args):
    b = BPF(src_file="blocksig.c")
    kill_fnname = b.get_syscall_fnname('kill')
    b.attach_kprobe(event=kill_fnname, fn_name='syscal
    pids_map = b.get_table('pids')
    sigs_map = b.get_table('sigs')

    args.pids.append(str(os.getpid()))
    for pid in args.pids:
        pids_map[c_int(int(pid))] = c_int(1)

    for sig in args.sig_array:
        sigs_map[sig] = c_int(1)
```

Our eBPF code

```
static u8 needs_block(u8 pid, u8 sig) {
    return pid != 0 && sig != 0;
}
int syscall__kill(struct pt_regs *ctx, int pid,
                  int sig)
{
    u8 *protected_pid = pids.lookup(&pid);
    u8 *protected_sig = sigs.lookup(&sig);
    if (!protected_pid || !protected_sig)
        return 0;
    if (needs_block(*protected_pid, *protected_sig)) {
        bpf_trace_printk("Blocked signal %d"
                         " for %d\\n", sig, pid);
        bpf_override_return(ctx, 0);
    }
    return 0;
}
```

It's demo time again again!

```
$ ping skallwar.fr > /dev/null &
[1] 315629
$ sudo ./blocksig.py 315629 &
$ kill -9 315629
$ # Nothing happened
$ kill -9 $(pidof python) # Pid of blocksig
$ # Nothing happened
```

Closed shell problem

If we are blocking signal for our process and the python interpreter, how can we stop `blocksig`?

With a ticket system

```
def wait_for_close():
    tf = tempfile.NamedTemporaryFile(delete = False)
    print(f"To stop blocksig, run ``rm {tf.name}``")

    try:
        while os.path.isfile(tf.name):
            time.sleep(0.5)
            continue
    except KeyboardInterrupt:
        tf.close()
        os.remove(tf.name)
```

Friendship ended with sudo


```
21995 sudo ./blocksig.py -p 21877
21996 └─ /usr/bin/python ./blocksig.py -p 21877
```

Sudo is my friend again

Using `exec`, sudo does not fork so I have only one pid to protect

It's new new new demo time!

Demo time!

Should you use this

No!

Some use case I found

- Unstoppable download
- Unstoppable long compilation
- Unstoppable `put your very long task here`
- Unstoppable malware

Misc

- First time I used override I had no if condition
- Your process are kill resistant but not `poweroff` resistant

Sources

- https://ebpf.io/what-is-ebpf
- https://skallwar.fr/posts/unkillable_process/

Questions?

Questions?