The Convergence of eBPF, Buildroot and QEMU for Automated Linux Malware Analysis

Nikhil Ashok Hegde
@ka1do9

whoami

- Senior Engineer at Netskope
- Security Research; Malware Analysis
- Gaming, Anime and Hiking!

This is my personal research, any views and opinions expressed are my own, and not those of any employer

Agenda

Runtime Behavior Tracing

Creating Lightweight Linux Images

Bringing It All Together

extended Berkeley Packet Filter (eBPF)

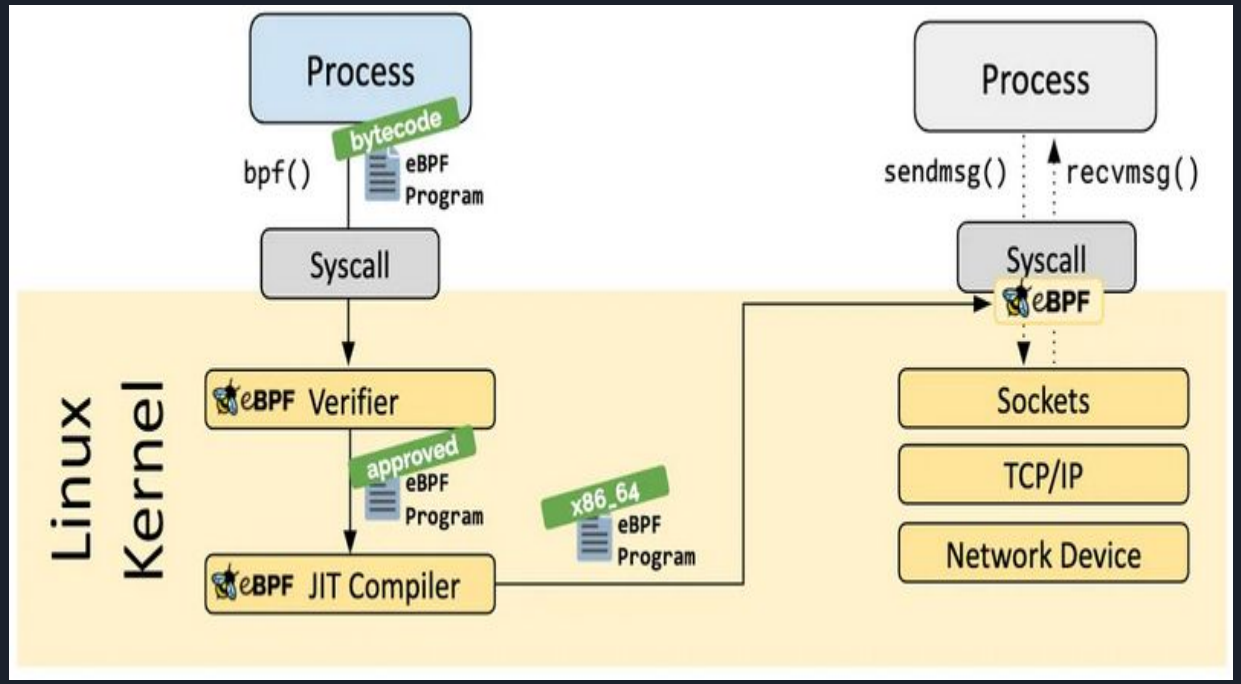
Runtime Behavior Tracing

What is eBPF?

- Safely runs sandboxed programs in the kernel
- BPF first introduced in 1992. Later enhanced in 2014. Today, BPF and eBPF terms are used interchangeably
- Variety of use-cases:
  - network packet filtering
  - performance troubleshooting
  - application behavior tracing → *Today's focus*

eBPF Verifier and JIT Compiler

Built into the Linux kernel and verified for safety



Source: ebpf.io

eBPF is Safe!

- Process must be privileged unless unprivileged BPF is enabled
- BPF program
  - must definitely end
  - cannot be arbitrarily large/complex
  - cannot access arbitrary kernel memory directly
- BPF program is hardened

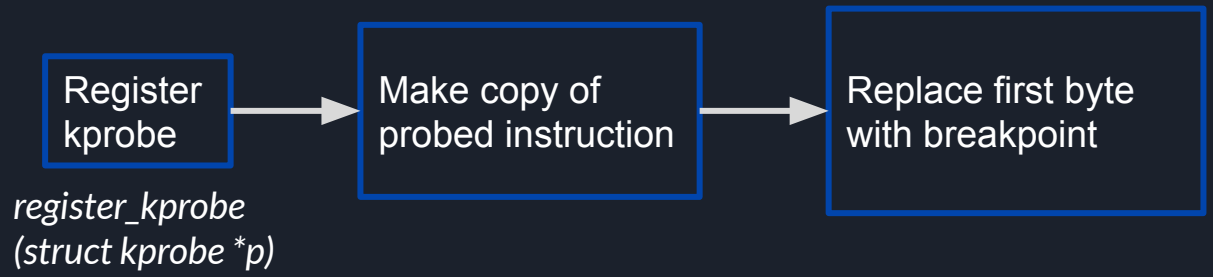
Probes and Tracing

- A probe is a location in the code where instrumentation can occur

Dynamic Instrumentation	Static Instrumentation
kprobe/kretprobe	Tracepoint
uprobe/uretprobe	USDT (User-Statically Defined Tracing)

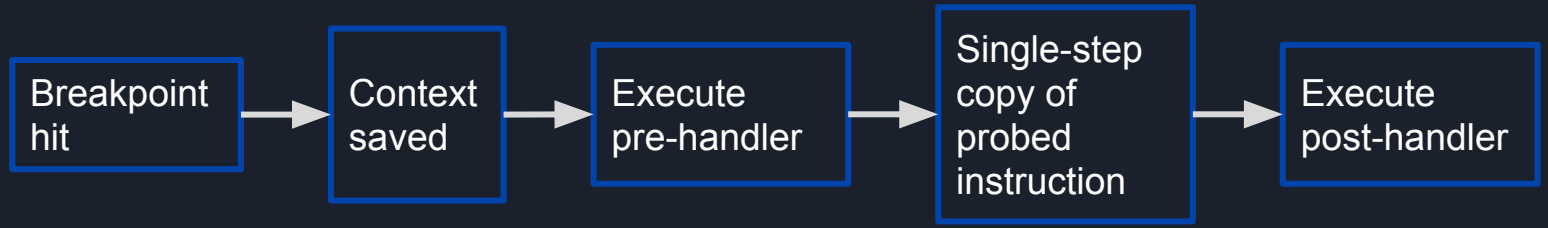
Kprobes

Setup



register_kprobe
*(struct kprobe *p)*

Action



Kprobes Structure

linux / include / linux / **kprobes.h**

Code | Blame | 602 lines (521 loc) · 16.3 KB

```
60  struct kprobe {
61          struct hlist_node hlist;
62
63          /* list of kprobes for multi-handler support */
64          struct list_head list;
65
66          /*count the number of times this probe was temporarily disarmed */
67          unsigned long nmissed;
68
69          /* location of the probe point */
70          kprobe_opcode_t *addr;
71
72          /* Allow user to indicate symbol name of the probe point */
73          const char *symbol_name;
74
75          /* Offset into the symbol */
76          unsigned int offset;
77
78          /* Called before addr is executed. */
79          kprobe_pre_handler_t pre_handler;
80
81          /* Called after addr is executed, unless... */
82          kprobe_post_handler_t post_handler;
83
84          /* Saved opcode (which has been replaced with breakpoint) */
85          kprobe_opcode_t opcode;
86
87          /* copy of the original instruction */
88          struct arch_specific_insn ainsn;
89
90          /*
91           * Indicates various status flags.
92           * Protected by kprobe_mutex after this kprobe is registered.
93           */
94          u32 flags;
95  };
```

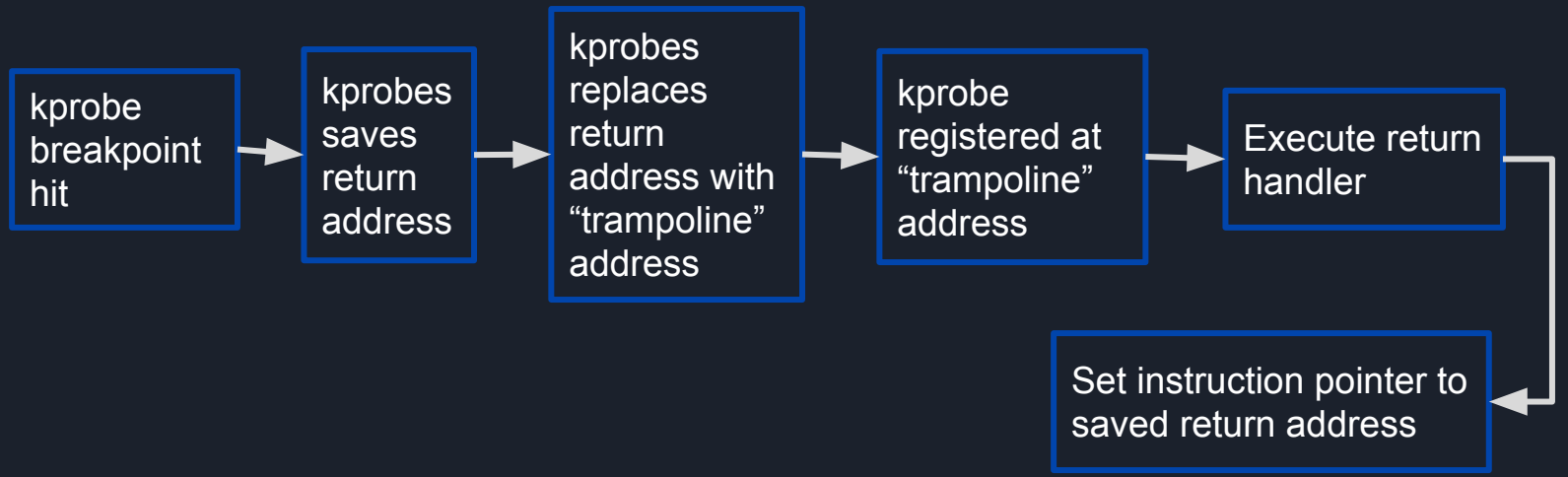
Declared in include/linux/kprobes.h

Kretprobes

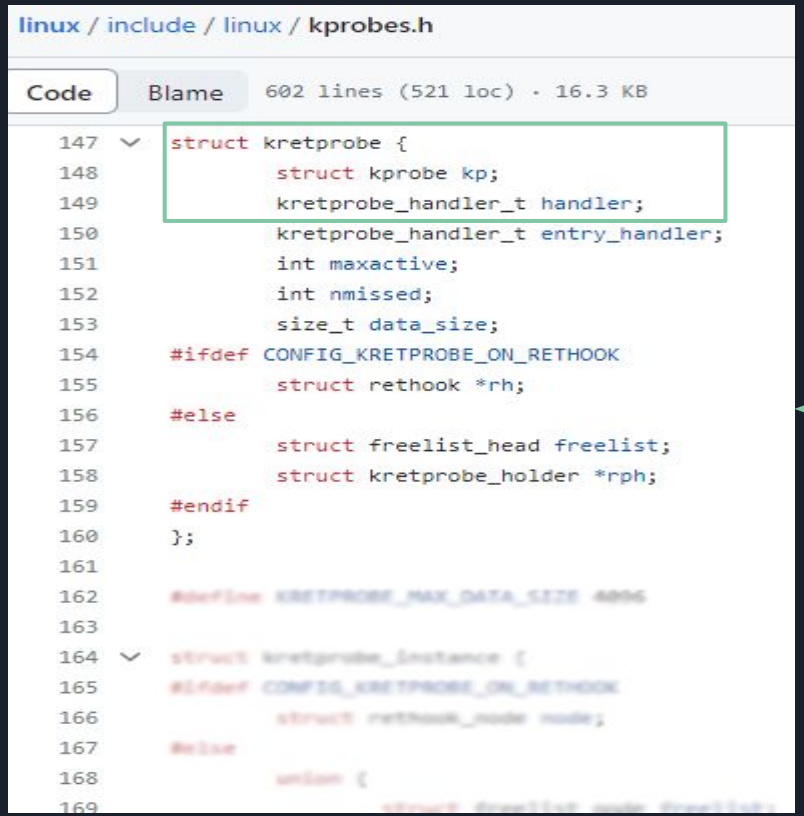
Setup

register_kretprobe
*(struct kretprobe *rp)*

Action



Kretprobes Structure



Declared in include/linux/kprobes.h

Kprobes/Kretprobes Support

- i386/x86-64
- ppc/ppc64
- arm
- mips
- ia64
- s390
- parisc
- sparc64 (only kprobes)

Tracepoints

- Predetermined hook points in kernel code
- More stable interface than kprobes/kretprobes

Tracepoints

```
398     * Tracepoint for exec:
399     */
400    TRACE_EVENT(sched_process_exec,
401
402            TP_PROTO(struct task_struct *p, pid_t old_pid,
403                     struct linux_binprm *bprm),
404
405            TP_ARGS(p, old_pid, bprm),
406
407            TP_STRUCT__entry(
408                    __string(       filename,       bprm->filename  )
409                    __field(        pid_t,          pid             )
410                    __field(        pid_t,          old_pid         )
411            ),
412
413            TP_fast_assign(
414                    __assign_str(filename, bprm->filename);
415                    __entry->pid            = p->pid;
416                    __entry->old_pid        = old_pid;
417            ),
418
419            TP_printk("filename=%s pid=%d old_pid=%d", __get_str(filename),
420                      __entry->pid, __entry->old_pid)
421    );
```

Name of traced event

subsystem

Declared in include/trace/events/sched.h

Tracepoints

```c
static int exec_binprm(struct linux_binprm *bprm)
{
        pid_t old_pid, old_vpid;
        int ret, depth;

        /* Need to fetch pid before load_binary changes it */
        old_pid = current->pid;
        rcu_read_lock();
        old_vpid = task_pid_nr_ns(current, task_active_pid_ns(current->parent));
        rcu_read_unlock();

        /* This allows 4 levels of binfmt rewrites before failing hard. */
        for (depth = 0;; depth++) {
                struct file *exec;
                if (depth > 5)
                        return -ELOOP;

                ret = search_binary_handler(bprm);
                if (ret < 0)
                        return ret;
                if (!bprm->interpreter)
                        break;

                exec = bprm->file;
                bprm->file = bprm->interpreter;
                bprm->interpreter = NULL;

                allow_write_access(exec);
                if (unlikely(bprm->have_execfd)) {
                        if (bprm->executable) {
                                fput(exec);
                                return -ENOEXEC;
                        }
                        bprm->executable = exec;
                } else
                        fput(exec);
        }

        audit_bprm(bprm);
        trace_sched_process_exec(current, old_pid, bprm);
```

Called by bprm_execve() which is called by execve()

Tracepoint exists in fs/exec.c

Actual tracepoint

eBPF Programming Front-ends

- Linux kernel requires eBPF bytecode for execution

- Popular front-ends to abstract away programming complexity:
  - SysmonForLinux (github.com/Sysinternals/SysmonForLinux)
  - bpftrace (github.com/iovisor/bpftrace)
  - bcc (github.com/iovisor/bcc)
  - ply (github.com/wkz/ply) → *Today's MVP*

Buildroot

Creating Lightweight Linux Images

What is Buildroot?

- Generate lightweight Linux images for various architectures
- Allows for granular customization of image
  - Add kernel parameters
  - Install required utilities

Link: buildroot.org

ELFEN Sandbox

Bringing it All Together

ELFEN Sandbox

- Dockerized Linux malware analysis sandbox
- Performs both static and dynamic analysis of Linux malware
- Leverages eBPF for tracing, Buildroot for building sandbox images

Link: github.com/nikhilh-20/ELFEN

ELFEN Architecture Support

- x86-64
- MIPS (32-bit, little/big-endian)
- PowerPC (32-bit, big-endian)
- ARMv5 (32-bit, little-endian)

ELFEN Tracer Choice: ply

- Lightweight eBPF-based dynamic tracer
  - Only one runtime library dependency: libc
- Available to install in Buildroot

Demo

Analysis with ELFEN

Future Work

- Community-driven detection content
- MITRE ATT&CK information in report
- Networking capability
- Support for more architectures

Questions?

 *ELFEN Sandbox*: github.com/nikhilh-20/ELFEN

 @ka1do9

 linkedin.com/in/nikhilh2/