第三届 eBPF开发者大会

www.ebpftravel.com

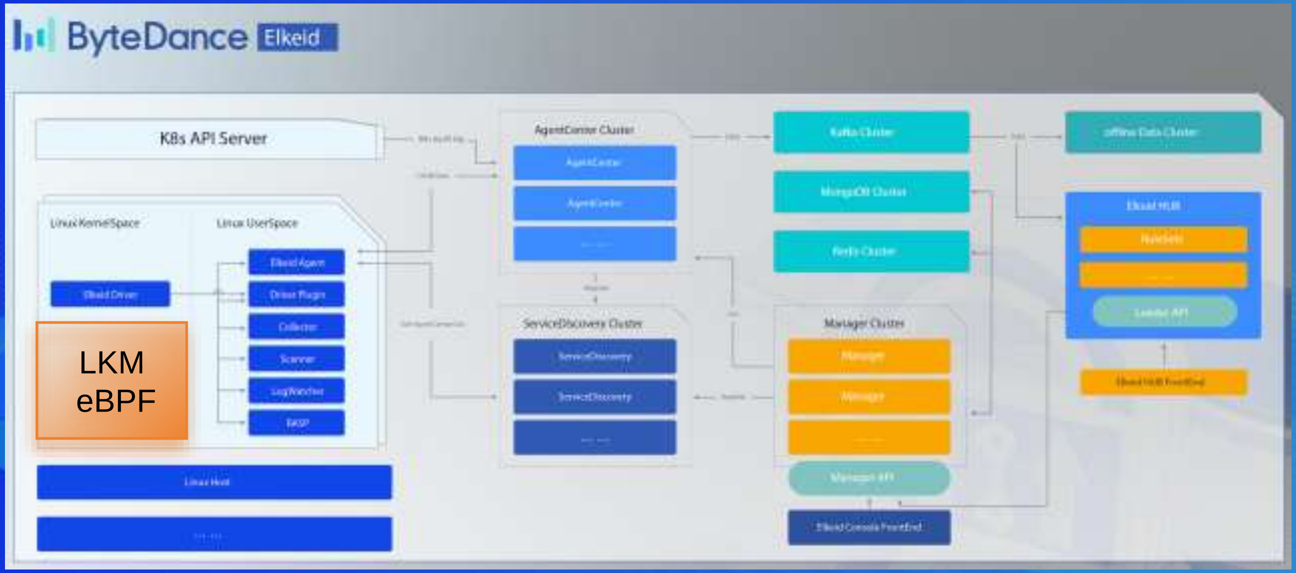
主机安全领域
eBPF的探索与实践

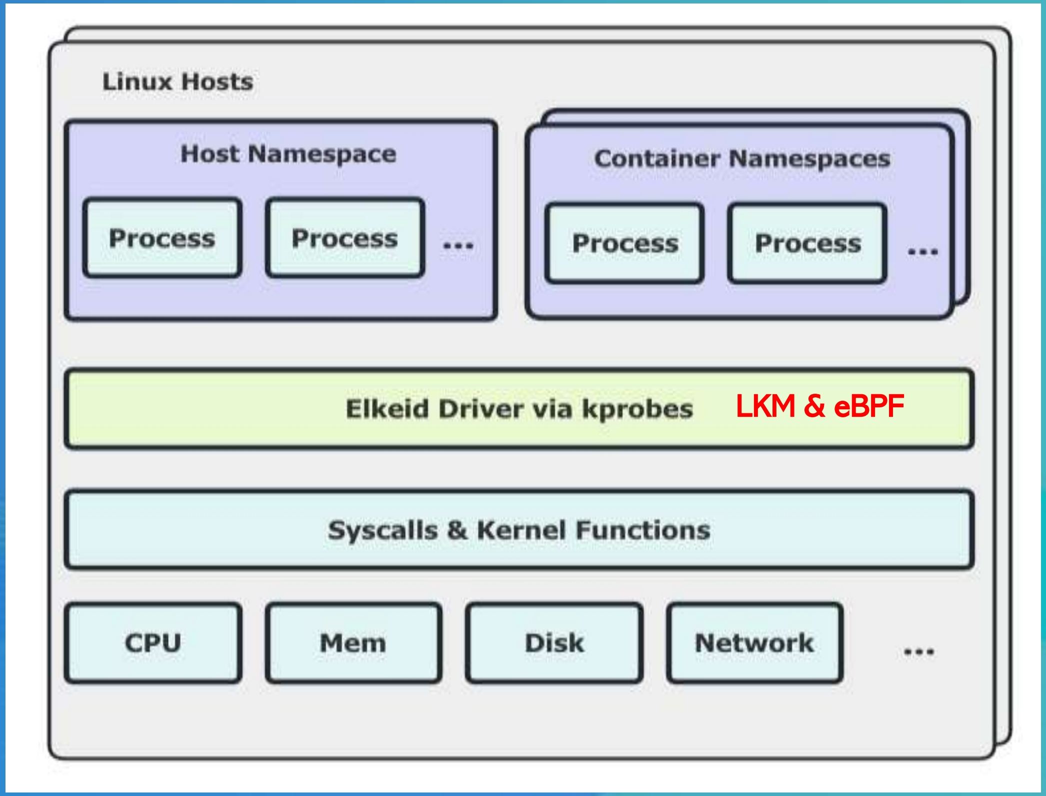
巫强 – 字节跳动终端安全

中国·西安
2025/04/19

ByteHIDS Elkeid CWPP/主机入侵检测

字节跳动终端安全团队开发的主机入侵检测系统，驱动组件的目标是跟踪和审计所有高危操作，如进程创 建、文件创建、网络连接和权限提升等





ByteHIDS：https://github.com/bytedance/elkeid

为什么 **用** eBPF

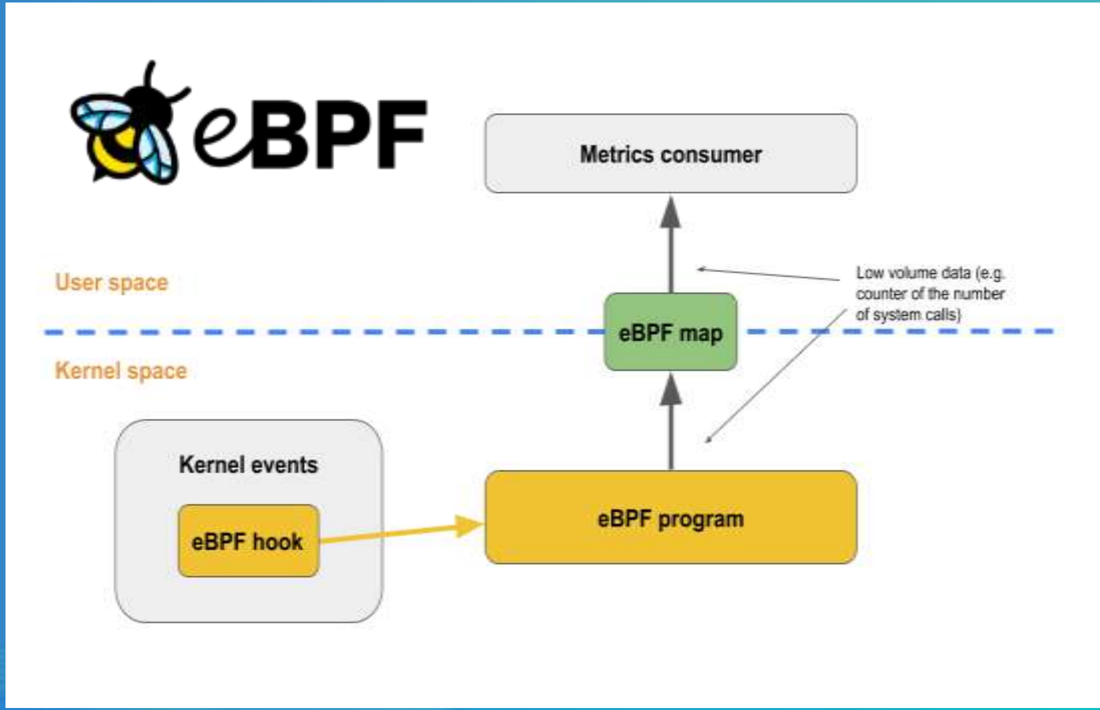
- 高可用场景，低侵入性
- 用户需求以及对LKM的忧虑
- Linux发行版及众多内核版本的适配

为什么 **不得不用** eBPF

- 模块签名校验：Secure Boot、可信计算环境、云主机证书链管理
- 用户自定义内核：缺失头文件，模块支持 (CONFIG_MODULES)
- 权限限制：禁止root权限 (CAP_SYS_MODULE)，SECCOMP

为什么 **不用** eBPF

- 内核版本：支持与否/支持程度，如tail call可用性 4.2->5.10
- 特殊场景及性能要求: lockdown模式 (v5.4)，架构/硬件相关

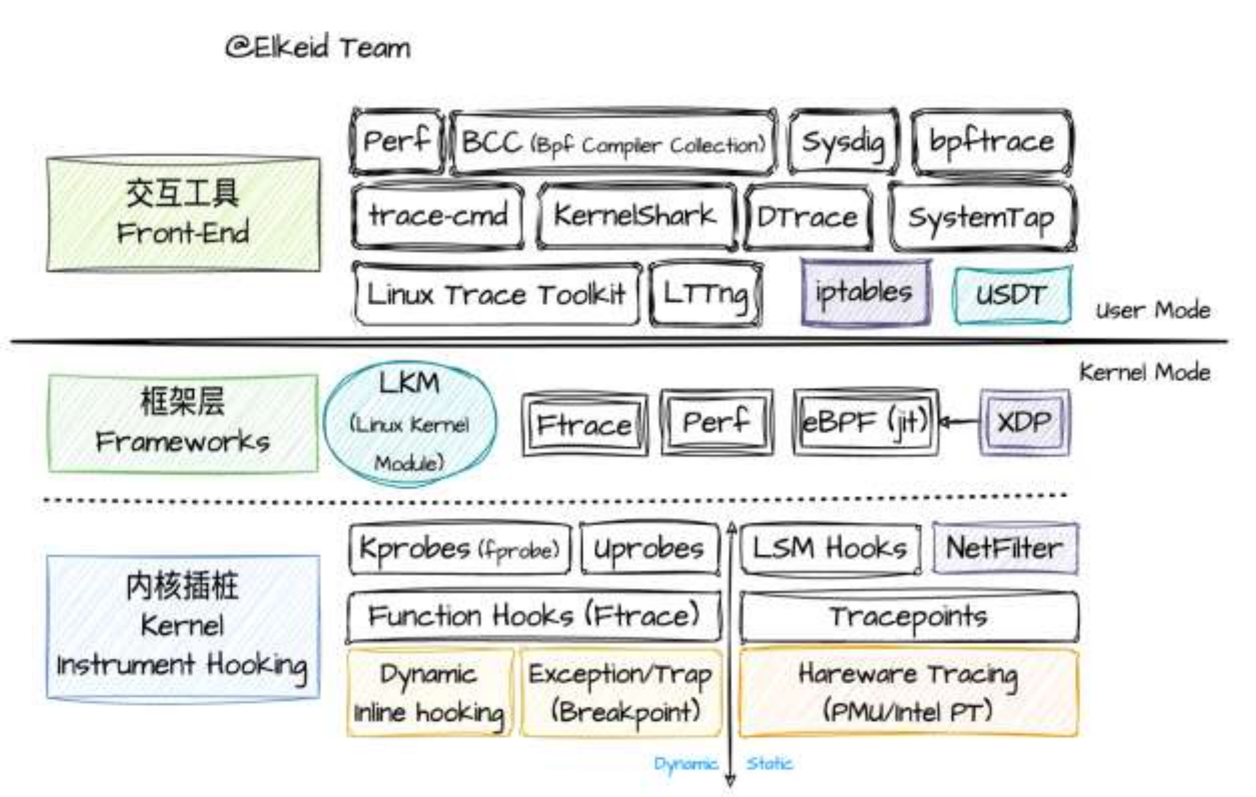


图片来源：https://sysdig.com/blog/sysdig-and-falco-now-powered-by-ebpf/

eBPF ≠ 高枕无忧

eBPF常用监控机制:

- Kprobe (v4.1)
- Kretprobe ✗
- Tracepoint (v4.7)
  - sys_enter_xxx
  - sys_exit_xxx
- Raw-tracepoint (v4.17)
  - sys_enter
  - sys_exit
- Fprobe (x86: v5.5 arm64: 6.0)
  - fentry
  - fexit
- BPF LSM (v5.7: Kernel Runtime Security Instrumentation)

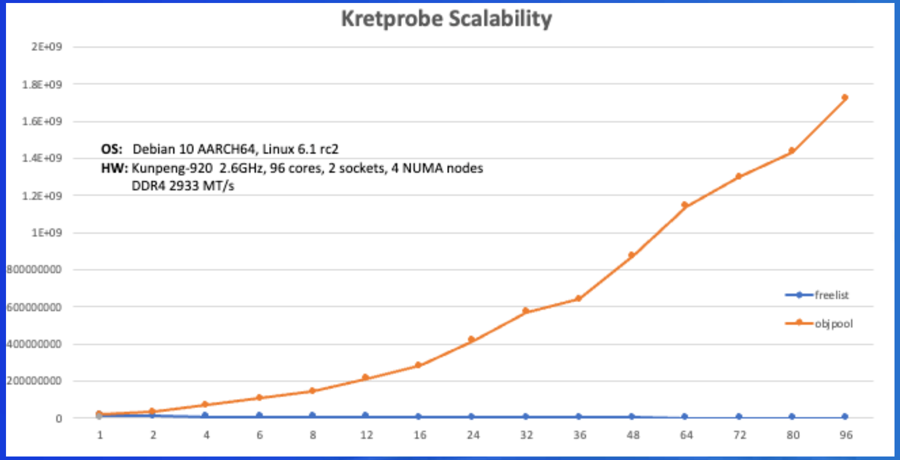


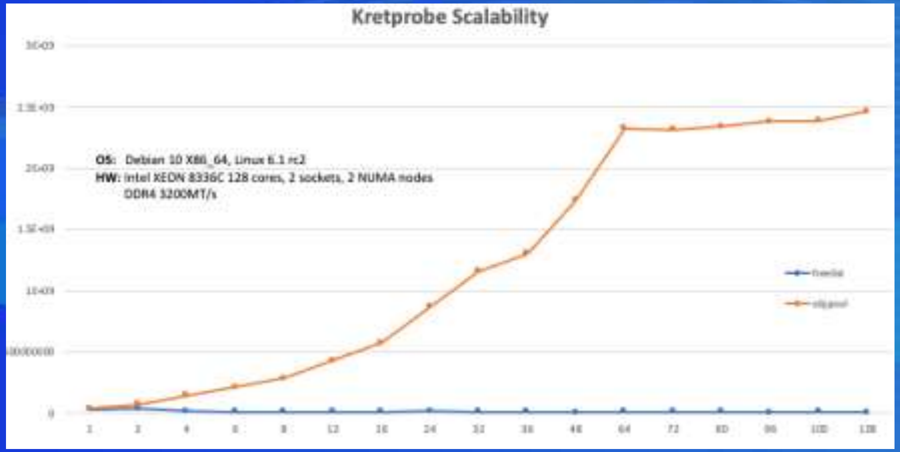
主要考量因素:

- 功能实现（高危操作入口或返回点， 64位系统中32位进程、 …）
- 工程便利（参数获取与信息流关联）
- 性能影响（内核版本与架构）

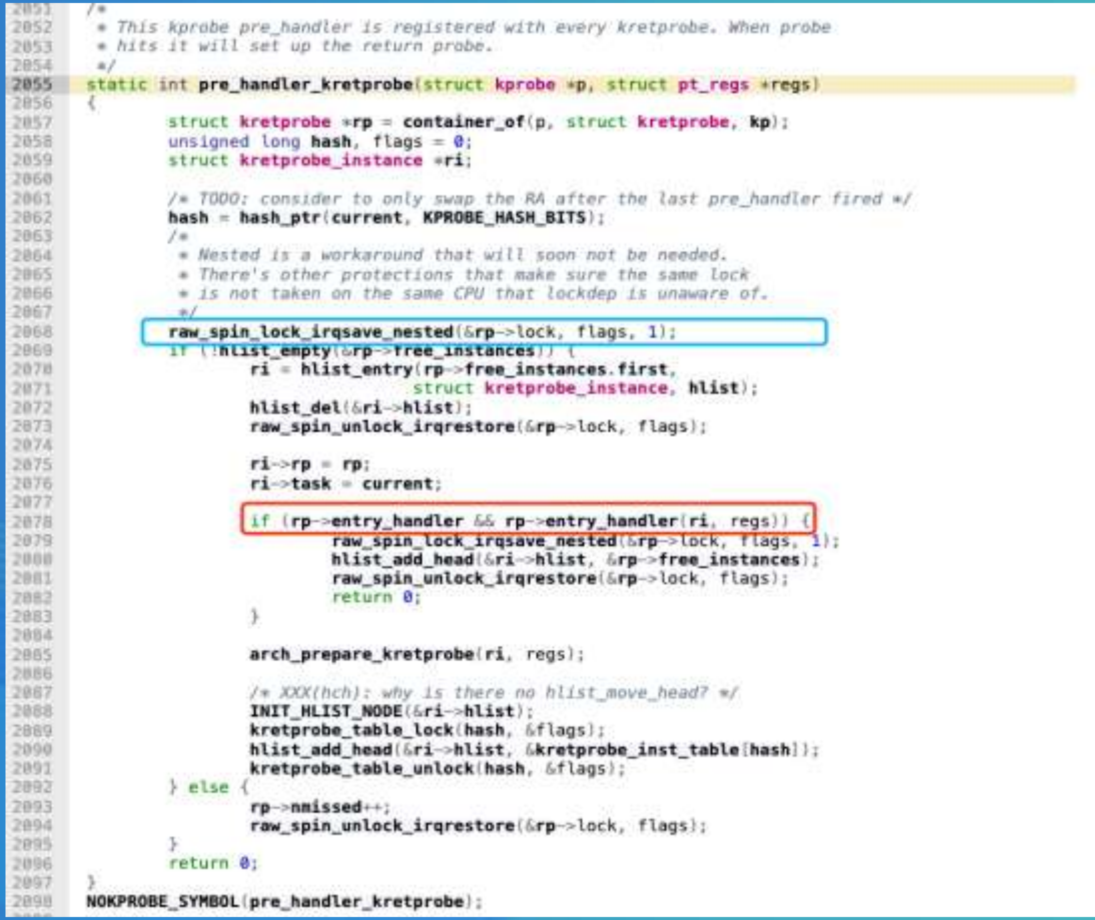
为什么 **不用** kretprobe

性能问题 - 可扩展性 (v6.7通过objpool解决)：





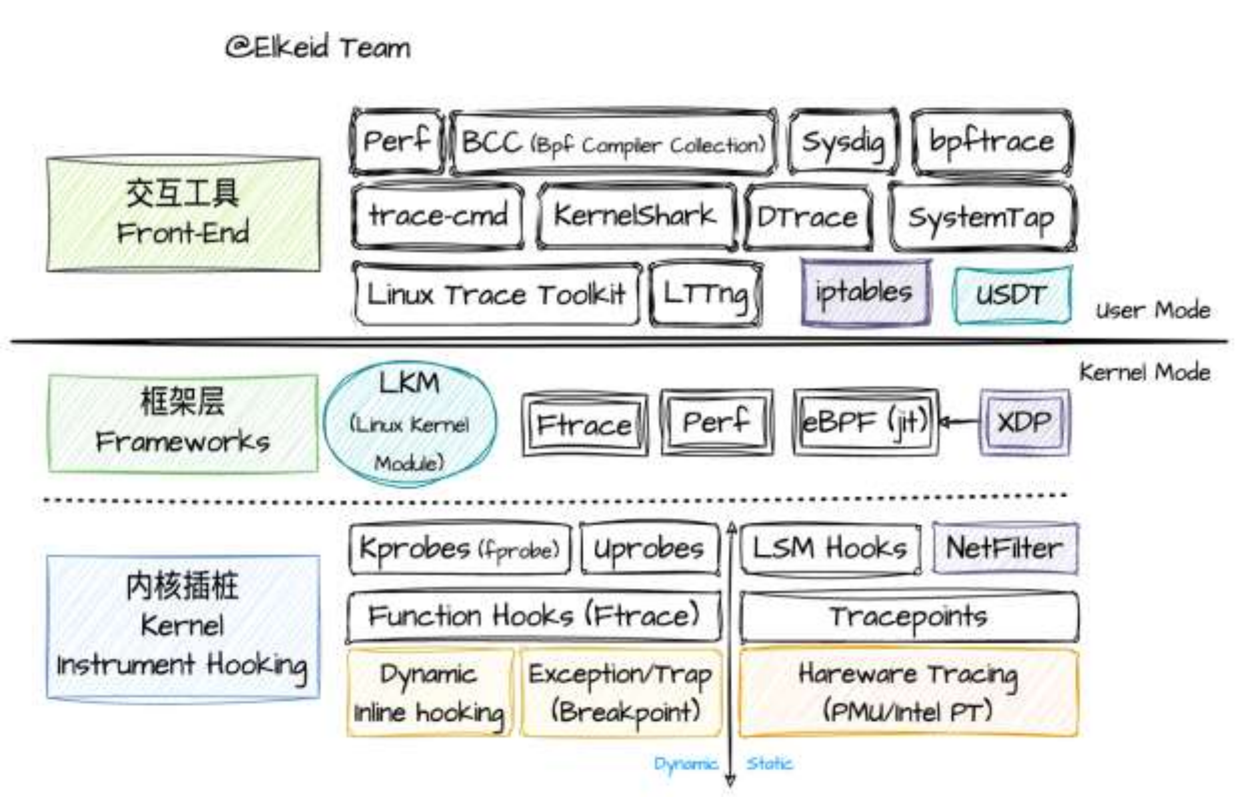
死锁风险 (5.12移除hash lock后才解决)：



```c
/*
 * This kprobe pre_handler is registered with every kretprobe. When probe
 * hits it will set up the return probe.
 */
static int pre_handler_kretprobe(struct kprobe *p, struct pt_regs *regs)
{
	struct kretprobe *rp = container_of(p, struct kretprobe, kp);
	unsigned long hash, flags = 0;
	struct kretprobe_instance *ri;

	/* TODO: consider to only swap the RA after the last pre_handler fired */
	hash = hash_ptr(current, KPROBE_HASH_BITS);
	/*
	 * Nested is a workaround that will soon not be needed.
	 * There's other protections that make sure the same lock
	 * is not taken on the same CPU that lockdep is unaware of.
	 */
	raw_spin_lock_irqsave_nested(&rp->lock, flags, 1);
	if (!hlist_empty(&rp->free_instances)) {
		ri = hlist_entry(rp->free_instances.first,
				struct kretprobe_instance, hlist);
		hlist_del(&ri->hlist);
		raw_spin_unlock_irqrestore(&rp->lock, flags);

		ri->rp = rp;
		ri->task = current;

		if (rp->entry_handler && rp->entry_handler(ri, regs)) {
			raw_spin_lock_irqsave_nested(&rp->lock, flags, 1);
			hlist_add_head(&ri->hlist, &rp->free_instances);
			raw_spin_unlock_irqrestore(&rp->lock, flags);
			return 0;
		}

		arch_prepare_kretprobe(ri, regs);

		/* XXX(hch): why is there no hlist_move_head? */
		INIT_HLIST_NODE(&ri->hlist);
		kretprobe_table_lock(hash, &flags);
		hlist_add_head(&ri->hlist, &kretprobe_inst_table[hash]);
		kretprobe_table_unlock(hash, &flags);
	} else {
		rp->nmissed++;
		raw_spin_unlock_irqrestore(&rp->lock, flags);
	}
	return 0;
}
NOKPROBE_SYMBOL(pre_handler_kretprobe);
```

eBPF常用监控机制:

- Kprobe (v4.1)
- Kretprobe ✗
- Tracepoint (v4.7)
  - ➢ sys_enter_xxx
  - ➢ sys_exit_xxx
- Raw-tracepoint (v4.17)
  - ➢ sys_enter
  - ➢ sys_exit
- Fprobe (x86: v5.5 arm64: 6.0)
  - ➢ fentry
  - ➢ fexit
- BPF LSM (v5.7: Kernel Runtime Security Instrumentation)



主要考量因素:

- 功能实现（高危操作入口或返回点， 64位系统中32位进程、 …）
- 工程便利（参数获取与信息流关联）
- 性能影响（内核版本与架构）

Tracepoint：32位程序的syscall？

/ kernel / trace / trace_syscalls.c

```c
static void perf_syscall_enter(void *ignore, struct pt_regs *regs, long id)
{
        struct syscall_metadata *sys_data;
        struct syscall_trace_enter *rec;
        struct pt_regs *fake_regs;
        struct hlist_head *head;
        unsigned long args[6];
        bool valid_prog_array;
        int syscall_nr;
        int rctx;
        int size;

        syscall_nr = trace_get_syscall_nr(current, regs);
        if (syscall_nr < 0 || syscall_nr >= NR_syscalls)
                return;
        if (!test_bit(syscall_nr, enabled_perf_enter_syscalls))
                return;
```

```c
        rec->nr = syscall_nr;
        syscall_get_arguments(current, regs, args);
        memcpy(&rec->args, args, sizeof(unsigned long) * sys_data->nb_args);

        if ((valid_prog_array &&
             !perf_call_bpf_enter(sys_data->enter_event, fake_regs, sys_data, rec)) ||
            hlist_empty(head)) {
                perf_swevent_put_recursion_context(rctx);
                return;
        }

        perf_trace_buf_submit(rec, size, rctx,
                              sys_data->enter_event->event.type, 1, regs,
                              head, NULL);
}
```

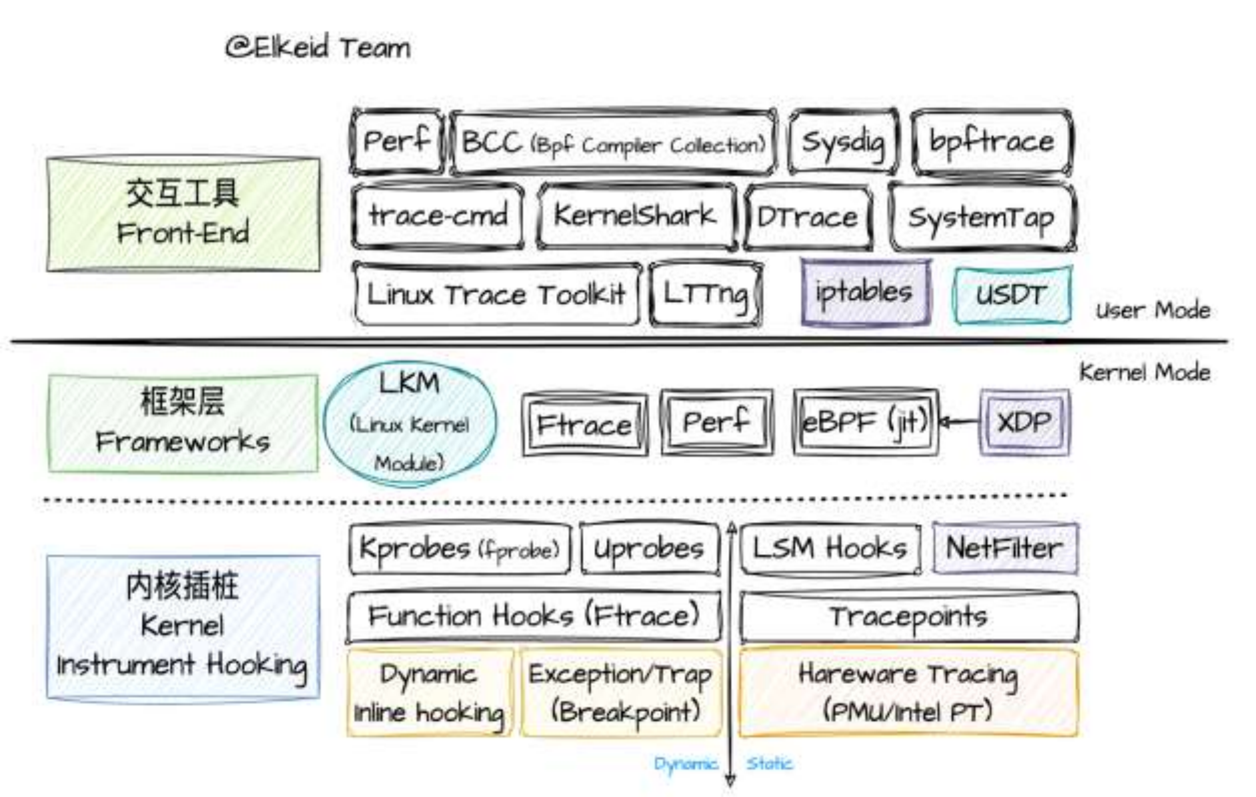
/ kernel / trace / trace_syscalls.c

```c
#ifdef ARCH_TRACE_IGNORE_COMPAT_SYSCALLS
/*
 * Some architectures that allow for 32bit applications
 * to run on a 64bit kernel, do not map the syscalls for
 * the 32bit tasks the same as they do for 64bit tasks.
 *
 *     *cough*x86*cough*
 *
 * In such a case, instead of reporting the wrong syscalls,
 * simply ignore them.
 *
 * For an arch to ignore the compat syscalls it needs to
 * define ARCH_TRACE_IGNORE_COMPAT_SYSCALLS as well as
 * define the function arch_trace_is_compat_syscall() to let
 * the tracing system know that it should ignore it.
 */
static int
trace_get_syscall_nr(struct task_struct *task, struct pt_regs *regs)
{
        if (unlikely(arch_trace_is_compat_syscall(regs)))
                return -1;

        return syscall_get_nr(task, regs);
}
```

```
root@P22:/BUILD/linux-6.6-op# grep -rn IGNORE_COMPAT_SYSCALLS *
arch/x86/include/asm/ftrace.h:144:#define ARCH_TRACE_IGNORE_COM
arch/riscv/include/asm/ftrace.h:37:#define ARCH_TRACE_IGNORE_CO
arch/arm64/include/asm/ftrace.h:175:#define ARCH_TRACE_IGNORE_CO
arch/s390/include/asm/ftrace.h:122:#define ARCH_TRACE_IGNORE_CO
```

eBPF常用监控机制:

- Kprobe (v4.1)
- Kretprobe ✘
- Tracepoint (v4.7)
  - sys_enter_xxx
  - sys_exit_xxx
- Raw-tracepoint (v4.17)
  - sys_enter
  - sys_exit
- Fprobe (x86: v5.5 arm64: 6.0)
  - fentry
  - fexit
- BPF LSM (v5.7: Kernel Runtime Security Instrumentation)



主要考量因素:

- 功能实现（高危操作入口或返回点， 64位系统中32位进程、 …）
- 工程便利（参数获取与信息流关联）
- 性能影响（内核版本与架构）

问题：架构无关的悖论

底层设计：

ISA: eBPF 字节码、 Just-In-Time编译器、虚拟机 (BPF VM)

细节依赖：

Tracepoint是对Raw Tracepoint的封装，前者禁止了对pt_regs的访问，牺牲灵便性的代价来换取跨架构的兼容性

实际需求：

1. 64位系统下32位进程的判断（ARM64/X86_64/X64_32）
2. Rootkit检测需要对IDT表/CRx寄存器的审计（X86: SMEP/SMAP ARM64: PAN/PXN）

X86_64

```
struct pt_regs {
/*
 * C ABI says these regs are ca
 * unless syscall needs a comple
 */
	unsigned long r15;
	unsigned long r14;
	unsigned long r13;
	unsigned long r12;
	unsigned long bp;
	unsigned long bx;
/* These regs are callee-clobbe
	unsigned long r11;
	unsigned long r10;
	unsigned long r9;
	unsigned long r8;
	unsigned long ax;
	unsigned long cx;
	unsigned long dx;
	unsigned long si;
	unsigned long di;
/*
 * On syscall entry, this is sys
 * On hw interrupt, it's IRQ num
 */
	unsigned long orig_ax;
/* Return frame for iretq */
	unsigned long ip;
	unsigned long cs;
	unsigned long flags;
	unsigned long sp;
	unsigned long ss;
/* top of stack page */
};
```

ARM64

```
struct pt_regs {
	union {
		struct user_pt_regs us
		struct {
			u64 regs[31];
			u64 sp;
			u64 pc;
			u64 pstate;
		};
	};
	u64 orig_x0;
#ifdef __AARCH64EB__
	u32 unused2;
	s32 syscallno;
#else
	s32 syscallno;
	u32 unused2;
#endif
	u64 sdei_ttbr1;
	/* Only valid when ARM64_HAS_G
	u64 pmr_save;
	u64 stackframe[2];

	/* Only valid for some EL1 exc
	u64 lockdep_hardirqs;
	u64 exit_rcu;
};
```

ARM64: 32b or 64b ELF

```
asmlinkage void noinstr el0t_32_sync_handler(struct pt_regs *regs)
{
	unsigned long esr = read_sysreg(esr_el1);

	switch (ESR_ELx_EC(esr)) {
	case ESR_ELx_EC_SVC32:
		el0_svc_compat(regs);
		break;
	case ESR_ELx_EC_DABT_LOW:
		el0_da(regs, esr);
		break;
	case ESR_ELx_EC_IABT_LOW:
		el0_ia(regs, esr);
		break;
	case ESR_ELx_EC_FP_ASIMD:
```

问题：性能与评估 (以raw-tracepoint为例)

LKM模块中rawtp框架对syscall的整体影响:

Syscall性能评估	3.10.0	6.9.3
原生系统 (Native)	1905008.6	2516184.7
Rawtp sys_exit (NUL)	1056719.0 55.5%	2354557.7 93.6%
启用 HIDS (elkeid.ko)	1011095.1 53.1%	2330793.3 92.6%

eBPF中rawtp对syscall的整体影响:

Syscall性能评估	5.4.0-48	5.15.152	6.9.3
原生系统 (Native)	5246526.3	4515345.0	4212502.0
Rawtp sys_exit (NUL)	4690607.8 89.4%	4256811.0 94.3%	3850585.3 91.4%
Rawtp sys_exit (NUL) + **BPF_CORE_READ**	3425319.7 65.3%	3935889.2 87.2%	3669082.2 87.1%

序号	名称	Id	监控点	注意事项
1	prctl	157	kprobe:security_task_prctl	改变comm名称：PRCTL_SET_NAME
2	execve	59	raw-tracepoint:sched_process_exec	ARM64系统缺失sys_exit的execve事件（于5.20修复），x86系统无此问题 sched_process_exec：只有成功的情况下才会有回调
3	connect	42	raw-tracepoint:sys_exit	32位程序syscall入口 (socketcall), ipv4及ipv6地址处理
4	bind	49		
5	accept	43		
6	dns	601		recvdat/recvmsg
7	create	602	kprobe:security_inode_create	rename及hardlink可以使用raw tracepoint实现，但新文件创建没有相对应的syscall
8	rename	82	kprobe:security_inode_rename	
9	hardlink	86	kprobe:security_inode_link	
10	call_usermodeh	607	kprobe: call_usermodehelper_exec	
11	init_module finit_module	603	kprobe:do_init_module	可选raw tracepoint：finit_module及init_module的差异
12	mount	165	raw-tracepoint:sys_exit	
13	memfd_create	356		
14	ptrace	101		
15	setsid	112		
16	chmod	90		
17	update cred	604	kprobe: commit_creds	改变当前进程的cred权限信息
18	提权检测	611	嵌入至其它监控点处理中	非set_euid/uid/.../等syscall但导致了uid、euid、gid等权限信息结果上的更改
19	落盘扫描	613	kprobe:filp_close	文件新建且有数据写入

Elkeid 驱动监控点列表

问题：两种ringbuf类型的选择

BPF ringbuf:

5.8及之后的内核才支持，此外bpf_ringbuf_reserve的输入size只能为常量，不支持动态长度

Perf ringbuf:

通用性强，但需要先将数据准备好且需要整体做一次内存拷贝

```c
SEC("raw_tracepoint/sys_enter")
int bpf_prog1(void *ctx)
{
        int max_len, max_buildid_len, total_size;
        struct stack_trace_t *data;
        long usize, ksize;
        void *raw_data;
        __u32 key = 0;

        data = bpf_map_lookup_elem(&stackdata_map, &key);
        if (!data)
                return 0;

        max_len = MAX_STACK_RAWTP * sizeof(__u64);
        max_buildid_len = MAX_STACK_RAWTP * sizeof(struct bpf_stack_build_id);
        data->pid = bpf_get_current_pid_tgid();
        data->kern_stack_size = bpf_get_stack(ctx, data->kern_stack,
                                              max_len, 0);
        data->user_stack_size = bpf_get_stack(ctx, data->user_stack, max_len,
                                             BPF_F_USER_STACK);
        data->user_stack_buildid_size = bpf_get_stack(
                ctx, data->user_stack_buildid, max_buildid_len,
                BPF_F_USER_STACK | BPF_F_USER_BUILD_ID);
        bpf_perf_event_output(ctx, &perfmap, 0, data, sizeof(*data));
```

问题：map动态调整及内存占用

不可动态调整：

只允许在map创建前调整map大小，针对不同规模的系统需要在eBPF加载前进行max entries的调整

内存占用 (过大?):

在开启CGROUP KMEM（CONFIG_MEMCG_KMEM=y）的情况下eBPF程序的内存占用会被统计至加载器所在的CGROUP

性能问题 (过小?):

Hash table类型的map底层用到了pcpu_freelist (基于spinlock分锁)，在数组设置过小的情况下会导致性能问题

```c
int bpf_mem_alloc_init(struct bpf_mem_alloc *ma, int size, bool percpu)
{
        struct bpf_mem_caches *cc, __percpu *pcc;
        struct bpf_mem_cache *c, __percpu *pc;
        struct obj_cgroup *objcg = NULL;
        int cpu, i, unit_size, percpu_size = 0;

        if (percpu && size == 0)
                return -EINVAL;

        /* room for llist_node and per-cpu pointer */
        if (percpu)
                percpu_size = LLIST_NODE_SZ + sizeof(void *);
        ma->percpu = percpu;

        if (size) {
                pc = __alloc_percpu_gfp(sizeof(*pc), 8, GFP_KERNEL);
                if (!pc)
                        return -ENOMEM;

                if (!percpu)
                        size += LLIST_NODE_SZ; /* room for llist_node */
                unit_size = size;

#ifdef CONFIG_MEMCG_KMEM
                if (memcg_bpf_enabled())
                        objcg = get_obj_cgroup_from_current();
#endif
                ma->objcg = objcg;

                for_each_possible_cpu(cpu) {
                        c = per_cpu_ptr(pc, cpu);
                        c->unit_size = unit_size;
                        c->objcg = objcg;
                        c->percpu_size = percpu_size;
                        c->tgt = c;
                        init_refill_work(c);
                        prefill_mem_cache(c, cpu);
                }
                ma->cache = pc;
                return 0;
        }
```

问题：部署与内核适配

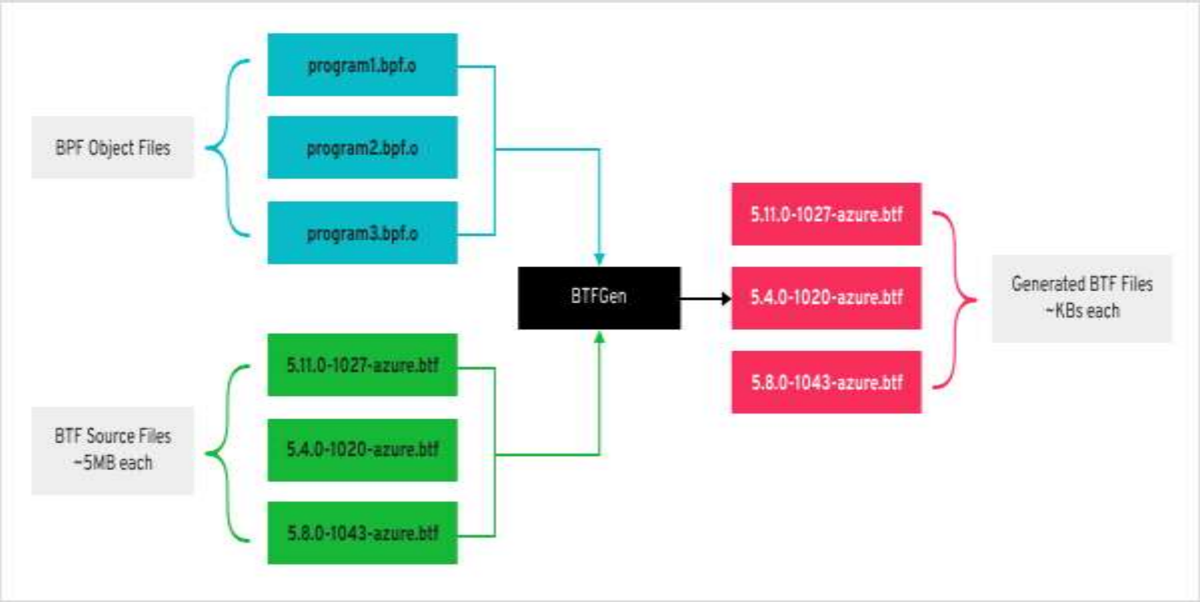
支持BTF的内核版本的适配:

尽可能复用同一个eBPF二进制程序，比如只为 5.4.x、5.10.x内核编译两个不同的eBPF目标文件，不同的发行版及架构间不通用

不支持BTF的内核版本适配:

利用bpftool/btfge具定制生成裁剪版本的BTF文件，同时保持eBPF程序的可复用性

```
bpftool gen min_core_btf /sys/kernel/btf/vmlinux mini.btf elkeid.btf
ls -l /sys/kernel/btf/vmlinux ./mini.btf
        4283 Apr 17 15:44 ./mini.btf
     4791512 Apr 17 15:42 /sys/kernel/btf/vmlinux
```



图片来源：https://inspektor-gadget.io/blog/2022/03/btfgen-one-step-closer-to-truly-portable-ebpf-programs/

?

THANKS

附录： tracepoint: sys_enter_xxx

调用栈:

```
pwndbg> bt
#0  0xffffffffc001c050 in ?? ()
#1  0xffffffff81256c7b in bpf_dispatcher_nop_func (bpf_func=0xffffffffc001c050, insnsi=0xffffc90000e3d048, ctx=0xffffc90000833e10) at ./include/linux/bpf.h:1181
#2  __bpf_prog_run (dfunc=<optimized out>, ctx=0xffffc90000833e10, prog=0xffffc90000e3d000) at ./include/linux/filter.h:609
#3  bpf_prog_run (ctx=0xffffc90000833e10, prog=0xffffc90000e3d000) at ./include/linux/filter.h:616
#4  bpf_prog_run_array (run_prog=<optimized out>, ctx=0xffffc90000833e10, array=<optimized out>) at ./include/linux/bpf.h:1926
#5  trace_call_bpf (call=call@entry=0xffffffff831e8160 <event_enter.openat>, ctx=ctx@entry=0xffffc90000833e10) at kernel/trace/bpf_trace.c:140
#6  0xffffffff8123a9aa in perf_call_bpf_enter (call=0xffffffff831e8160 <event_enter.openat>, regs=regs@entry=0xffffc90000833f58, rec=rec@entry=0xffffe8ffffd7a9b8, sys_data=0xffffffff831e81e0 <__syscal
l_meta__openat>) at kernel/trace/trace_syscalls.c:571
#7  0xffffffff8123ab29 in perf_syscall_enter (ignore=<optimized out>, regs=0xffffc90000833f58, id=<optimized out>) at kernel/trace/trace_syscalls.c:614
#8  0xffffffff81187ecf in trace_sys_enter (id=257, regs=0xffffc90000833f58) at ./include/trace/events/syscalls.h:18
#9  syscall_trace_enter (regs=0xffffc90000833f58, work=<optimized out>, syscall=257) at kernel/entry/common.c:81
#10 0xffffffff81f325bc in __syscall_enter_from_user_work (syscall=<optimized out>, regs=0xffffc90000833f58) at kernel/entry/common.c:94
#11 syscall_enter_from_user_mode (regs=regs@entry=0xffffc90000833f58, syscall=<optimized out>) at kernel/entry/common.c:112
#12 0xffffffff81f2af05 in do_syscall_64 (regs=0xffffc90000833f58, nr=<optimized out>) at arch/x86/entry/common.c:76
#13 0xffffffff820000e6 in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:120
#14 0x00007ff14ab5a6b8 in ?? ()
#15 0x00007ff14ab59020 in ?? ()
#16 0x0000000000000001 in fixed_percpu_data ()
#17 0xffffffffffffffff in ?? ()
#18 0x00007ff14ab5a1f8 in ?? ()
#19 0x000055b379e2d582 in ?? ()
#20 0x0000000000000287 in ?? ()
#21 0x0000000000000000 in ?? ()
```

参数:

struct syscall_tp_t *args; 前8字节 (struct trace_entry) 实际存放的是pt_regs，但ebpf程序不可访问

tracepoint: sys_exit_xxx

调用栈:

```
pwndbg> bt
#0  0xffffffffc001c054 in ?? ()
#1  0xffffffff81256c7b in bpf_dispatcher_nop_func (bpf_func=0xffffffffc001c054, insnsi=0xffffc90000911048, ctx=0xffffc90000cfbe88) at ./include/linux/bpf.h:1181
#2  __bpf_prog_run (dfunc=<optimized out>, ctx=0xffffc90000cfbe88, prog=0xffffc90000911000) at ./include/linux/filter.h:609
#3  bpf_prog_run (ctx=0xffffc90000cfbe88, prog=0xffffc90000911000) at ./include/linux/filter.h:616
#4  bpf_prog_run_array (run_prog=<optimized out>, ctx=0xffffc90000cfbe88, array=<optimized out>) at ./include/linux/bpf.h:1926
#5  trace_call_bpf (call=<optimized out>, ctx=ctx@entry=0xffffc90000cfbe88) at kernel/trace/bpf_trace.c:140
#6  0xffffffff8123a8ae in perf_call_bpf_exit (rec=0xffffe8ffffdba9b0, regs=0xffffc90000cfbf58, call=<optimized out>) at kernel/trace/trace_syscalls.c:672
#7  perf_syscall_exit (ignore=<optimized out>, regs=0xffffc90000cfbf58, ret=<optimized out>) at kernel/trace/trace_syscalls.c:712
#8  0xffffffff81187a03 in __traceiter_sys_exit (__data=0xffffc90000cfbe88, regs=0xffffc90000cfbf58, ret=9) at ./include/trace/events/syscalls.h:44
#9  0xffffffff8118823d in trace_sys_exit (ret=<optimized out>, regs=0xffffc90000cfbf58) at ./include/trace/events/syscalls.h:44
#10 syscall_exit_work (regs=regs@entry=0xffffc90000cfbf58, work=18) at kernel/entry/common.c:247
#11 0xffffffff81f32673 in syscall_exit_to_user_mode_prepare (regs=regs@entry=0xffffc90000cfbf58) at kernel/entry/common.c:278
#12 __syscall_exit_to_user_mode_work (regs=regs@entry=0xffffc90000cfbf58) at kernel/entry/common.c:283
#13 syscall_exit_to_user_mode (regs=regs@entry=0xffffc90000cfbf58) at kernel/entry/common.c:296
#14 0xffffffff81f2af36 in do_syscall_64 (regs=0xffffc90000cfbf58, nr=<optimized out>) at arch/x86/entry/common.c:86
#15 0xffffffff820000e6 in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:120
#16 0x0000000000000000 in ?? ()
```

参数:
struct syscall_tp_t *args; 前8字节实际存放的是pt_regs，但ebpf程序不可访问

kprobe hook

调用栈:

```
pwndbg> bt
#0  0xffffffffc0020a20 in ?? ()
#1  0xffffffff81256c7b in bpf_dispatcher_nop_func (bpf_func=0xffffffffc0020a20, insnsi=0xffffc90000bf3048, ctx=0xffffc90007be7c48) at ./include/linux/bpf.h:1181
#2  __bpf_prog_run (dfunc=<optimized out>, ctx=0xffffc90007be7c48, prog=0xffffc90000bf3000) at ./include/linux/filter.h:609
#3  bpf_prog_run (ctx=0xffffc90007be7c48, prog=0xffffc90000bf3000) at ./include/linux/filter.h:616
#4  bpf_prog_run_array (run_prog=<optimized out>, ctx=0xffffc90007be7c48, array=<optimized out>) at ./include/linux/bpf.h:1926
#5  trace_call_bpf (call=call@entry=0xffff8881071ca250, ctx=ctx@entry=0xffffc90007be7c48) at kernel/trace/bpf_trace.c:140
#6  0xffffffff8125a6ff in kprobe_perf_func (tk=tk@entry=0xffff8881071cbf00, regs=regs@entry=0xffffc90007be7c48) at kernel/trace/trace_kprobe.c:1478
#7  0xffffffff8125a991 in kprobe_dispatcher (kp=0xffff8881071cbf18, regs=0xffffc90007be7c48) at kernel/trace/trace_kprobe.c:1616
#8  0xffffffff810900aa in kprobe_ftrace_handler (ip=18446744071579872624, parent_ip=<optimized out>, ops=<optimized out>, fregs=<optimized out>) at arch/x86/kernel/kprobes/ftrace.c:42
#9  0xffffffffc07570f7 in ?? ()
#10 0xffff88817d010000 in ?? ()
#11 0xffff88810308c200 in ?? ()
#12 0xffff888103328029 in ?? ()
#13 0xffff88817d010000 in ?? ()
#14 0xffffc90007be7d70 in ?? ()
#15 0xffff888108a17800 in ?? ()
#16 0xfefefefefefefeff in ?? ()
#17 0xffff88817d010b53 in ?? ()
#18 0x0000000000000003 in fixed_percpu_data ()
#19 0x0000000000000005 in fixed_percpu_data ()
#20 0xffff8881020410a0 in ?? ()
#21 0x0000000000000000 in ?? ()
```

参数:

struct pt_regs *regs;

raw-tracepoint: sys_enter/sys_exit

调用栈:

```
pwndbg> bt
#0  0xffffffffc00138ac in ?? ()
#1  0xffffffff81254bd1 in bpf_dispatcher_nop_func (bpf_func=0xffffffffc00138ac, insnsi=0xffffc900019b1048, ctx=0xffffc90000d33e90) at ./include/linux/bpf.h:1181
#2  __bpf_prog_run (dfunc=<optimized out>, ctx=0xffffc90000d33e90, prog=0xffffc900019b1000) at ./include/linux/filter.h:609
#3  bpf_prog_run (ctx=0xffffc90000d33e90, prog=0xffffc900019b1000) at ./include/linux/filter.h:616
#4  __bpf_trace_run (args=0xffffc90000d33e90, prog=0xffffc900019b1000) at kernel/trace/bpf_trace.c:2306
#5  bpf_trace_run2 (prog=0xffffc900019b1000, arg0=<optimized out>, arg1=<optimized out>) at kernel/trace/bpf_trace.c:2345
#6  0xffffffff81187d29 in __bpf_trace_sys_exit () at ./include/trace/events/syscalls.h:44
#7  0xffffffff81187a03 in __traceiter_sys_exit (__data=0xffffc90000d33e90, regs=0xffffc90000d33f58, ret=-110) at ./include/trace/events/syscalls.h:44
#8  0xffffffff8118823d in trace_sys_exit (ret=<optimized out>, regs=0xffffc90000d33f58) at ./include/trace/events/syscalls.h:44
#9  syscall_exit_work (regs=regs@entry=0xffffc90000d33f58, work=18) at kernel/entry/common.c:247
#10 0xffffffff81f32673 in syscall_exit_to_user_mode_prepare (regs=regs@entry=0xffffc90000d33f58) at kernel/entry/common.c:278
#11 __syscall_exit_to_user_mode_work (regs=regs@entry=0xffffc90000d33f58) at kernel/entry/common.c:283
#12 syscall_exit_to_user_mode (regs=regs@entry=0xffffc90000d33f58) at kernel/entry/common.c:296
#13 0xffffffff81f2af36 in do_syscall_64 (regs=0xffffc90000d33f58, nr=<optimized out>) at arch/x86/entry/common.c:86
#14 0xffffffff820000e6 in entry_SYSCALL_64 () at arch/x86/entry/entry_64.S:120
#15 0x00007ff4307ed000 in ?? ()
#16 0x00007fffdccd0da0 in ?? ()
#17 0x0000000000000016 in fixed_percpu_data ()
#18 0xffffffffffffff78 in ?? ()
#19 0x00007ff430fecd50 in ?? ()
#20 0x000000c00005c400 in ?? ()
#21 0x0000000000000202 in ?? ()
#22 0x00007ff430fecd40 in ?? ()
#23 0x0000000000000000 in ?? ()
```

参数:

struct bpf_raw_tracepoint_args *ctx;
其中ctx->args[0]为pt_regs指针，sys_exit的情况下ctx->args[1]为syscall返回值