第三届 eBPF开发者大会

www.ebpftravel.com

eBPF在rootkit攻防中的应用

中国·西安

个人介绍

- 从业12年，现就职于青藤云安全公司（公司愿景：让云更安全）
- 从业至今一直专注于网络安全领域，在主机侧和网络侧入侵检测、网络流量分析、抗DDOS领域积累了丰富的经验
- 持续探索eBPF技术在安全领域中的应用，研究过Falco、Tracee等开源项目。
- 热爱技术分享，在知乎、b站分享安全类的编程实战，希望能对网络安全的新人有所帮助，为中国网安事业贡献自己的一份力量。

演讲大纲

- Rootkit介绍
- Rootkit种类和常见实现手法
- ebpf来检测Rootkit的优势和劣势
- Rootkit的攻击流程中各个阶段的实现
- 使用ebpf技术检测Rootkit攻击流程中的各个阶段

Rootkit是什么?

1、偷偷敲入你家的小偷

- 不仅潜入你家，还偷走钥匙（获取系统最高权限）
- 在你家装了隐藏的摄像头（后门程序）
- 擦掉所有活动痕迹（清除日志）

2、更可怕的

- 它能让自己隐形（隐藏进程）
- 你家的监控系统看不到他（欺骗安全软件和系统安全组件）
- 他动过的东西你也发现不了（隐藏文件）
- 他还能修改你家得监控系统（修改系统功能）

Rootkit行为

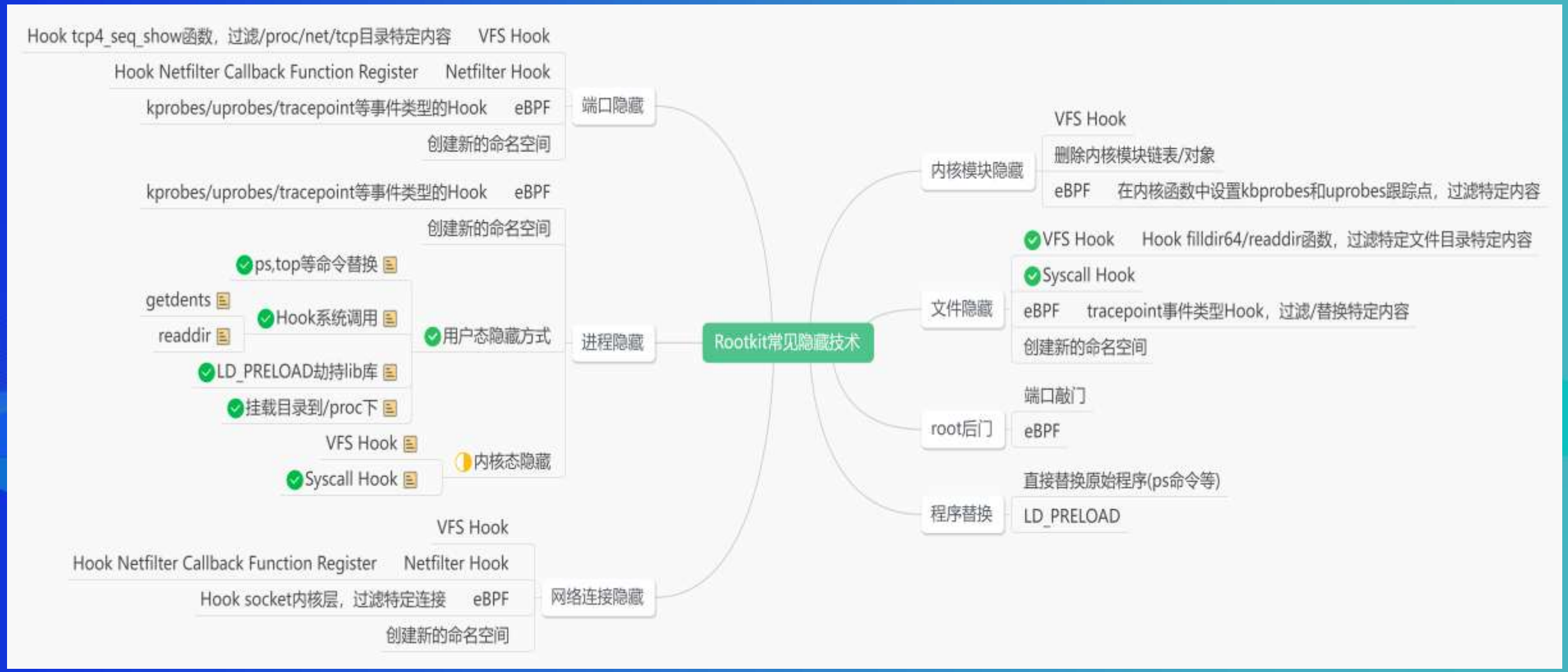
进程

文件

网络连接

隐藏自身

Rootkit的种类和常见实现



为什么采用ebpf检测内核态Rootkit?

跟踪方法评判标准

低开销

隐蔽性

强安全边界

全系统可见性

安全性

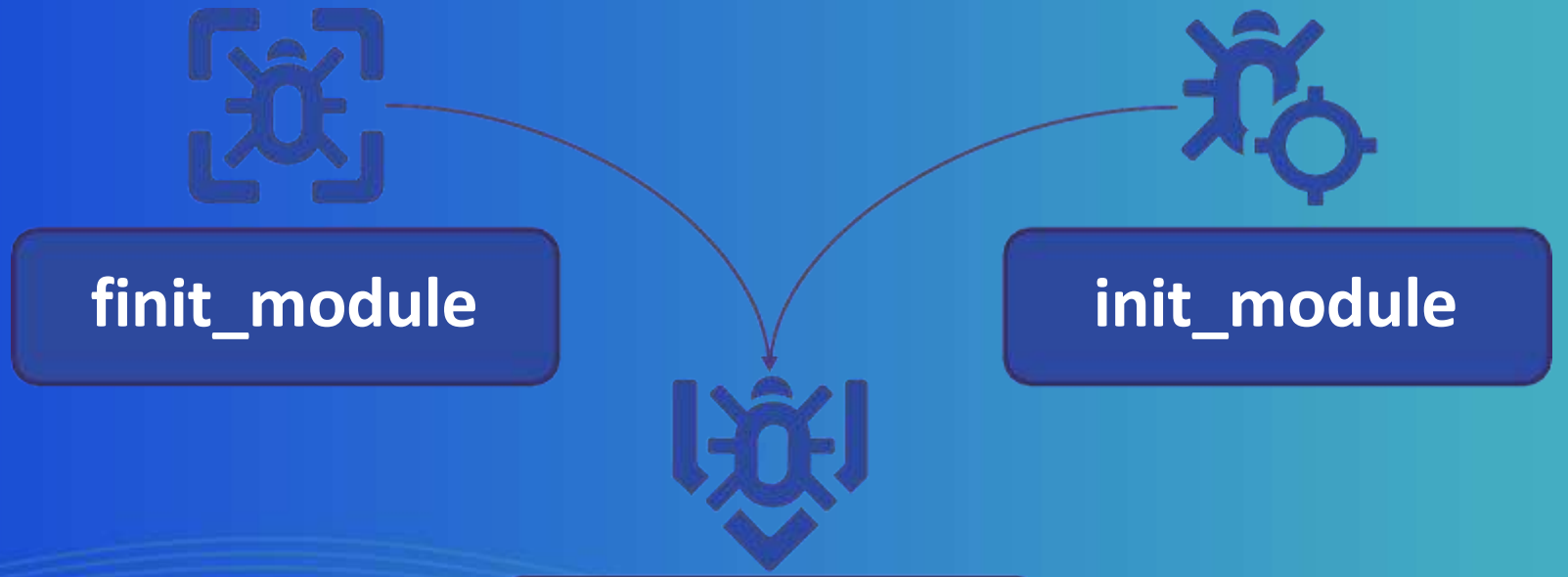
跟踪方法横向对比

eBPF是最好的跟踪技术

技术	低开销	隐蔽性	强安全边界	可见性	安全
ptrace	✗	✗	✗	✗	✗
LD_PRELOAD	✓	✗	✗	✗	✗
内核模块	✓	✓	✓	✓	✗
eBPF	✓	✓	✓	✓	✓

Rootkit攻击流程简介

一、内核模块加载


finit_module
init_module
do_init_module
kprobe


内核模块加载
内核模块隐藏
Hooking技术
通信通道
命令执行

一、内核模块加载和检测实验

1 ebpf检测内核模块加载

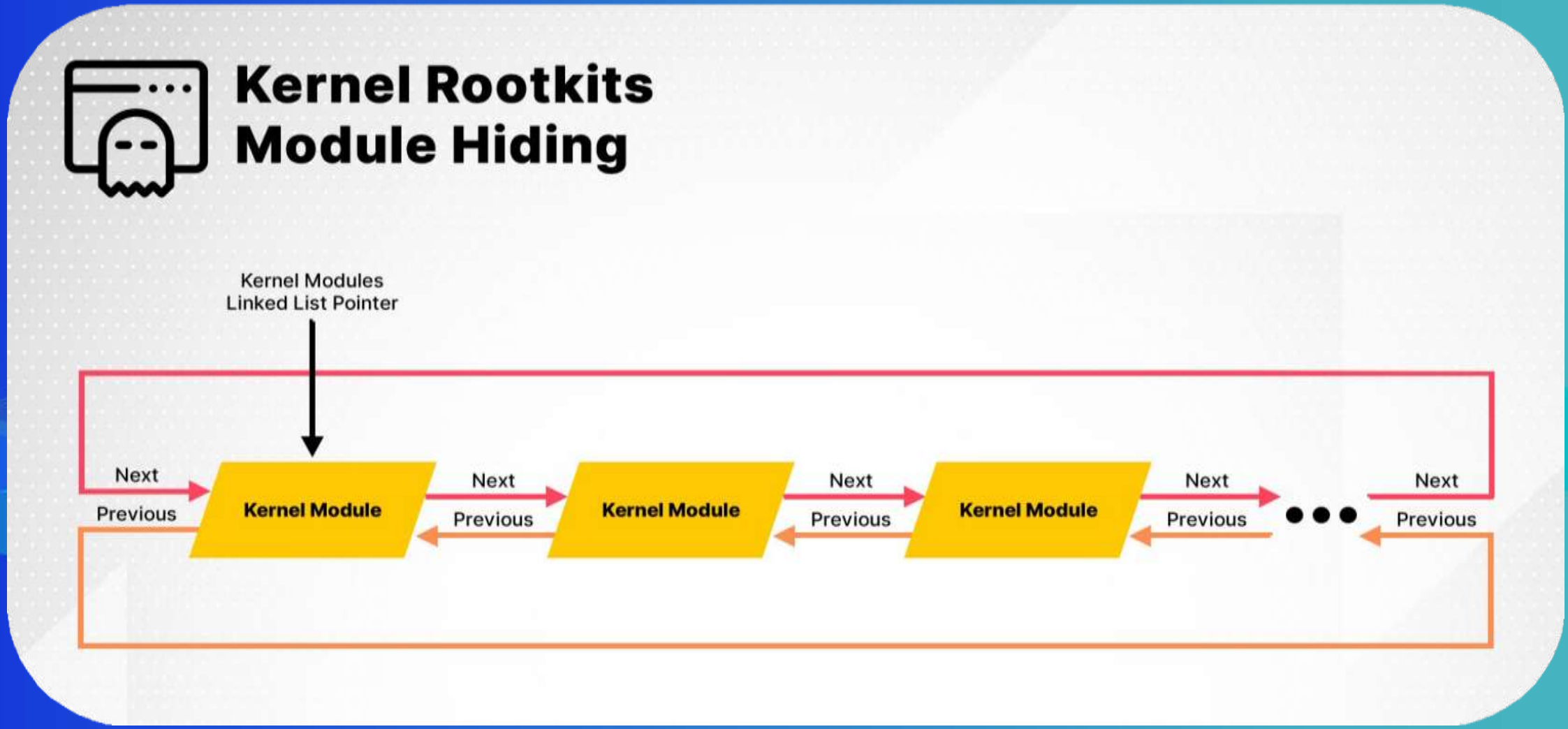
```
root@ubuntu:/home/work/goProject/src/tracee/dist# ls
btfhub  libbpf  signatures  tracee  tracee.bpf.o  tracee-ebpf  tracee-rules
root@ubuntu:/home/work/goProject/src/tracee/dist# ./tracee-ebpf -e=init_module,finit_module,do_init_module,magic_write
 -c module

TIME             UID    COMM             PID     TID     RET             EVENT                ARGS
07:54:34:305227  0      insmod           2373    2373    0               do_init_module       name: diamorphine,
 version: , src_version: 36D91AB513C97F2E501C612
07:54:34:270726  0      insmod           2373    2373    0               finit_module         fd: 3, param_value
s: , flags: 0
07:54:34:305763  0      systemd-udevd    2374    2374    2               magic_write          pathname: /proc/23
74/oom_score_adj, bytes: [48 10], dev: 23, inode: 60677
```

1 内核模块加载

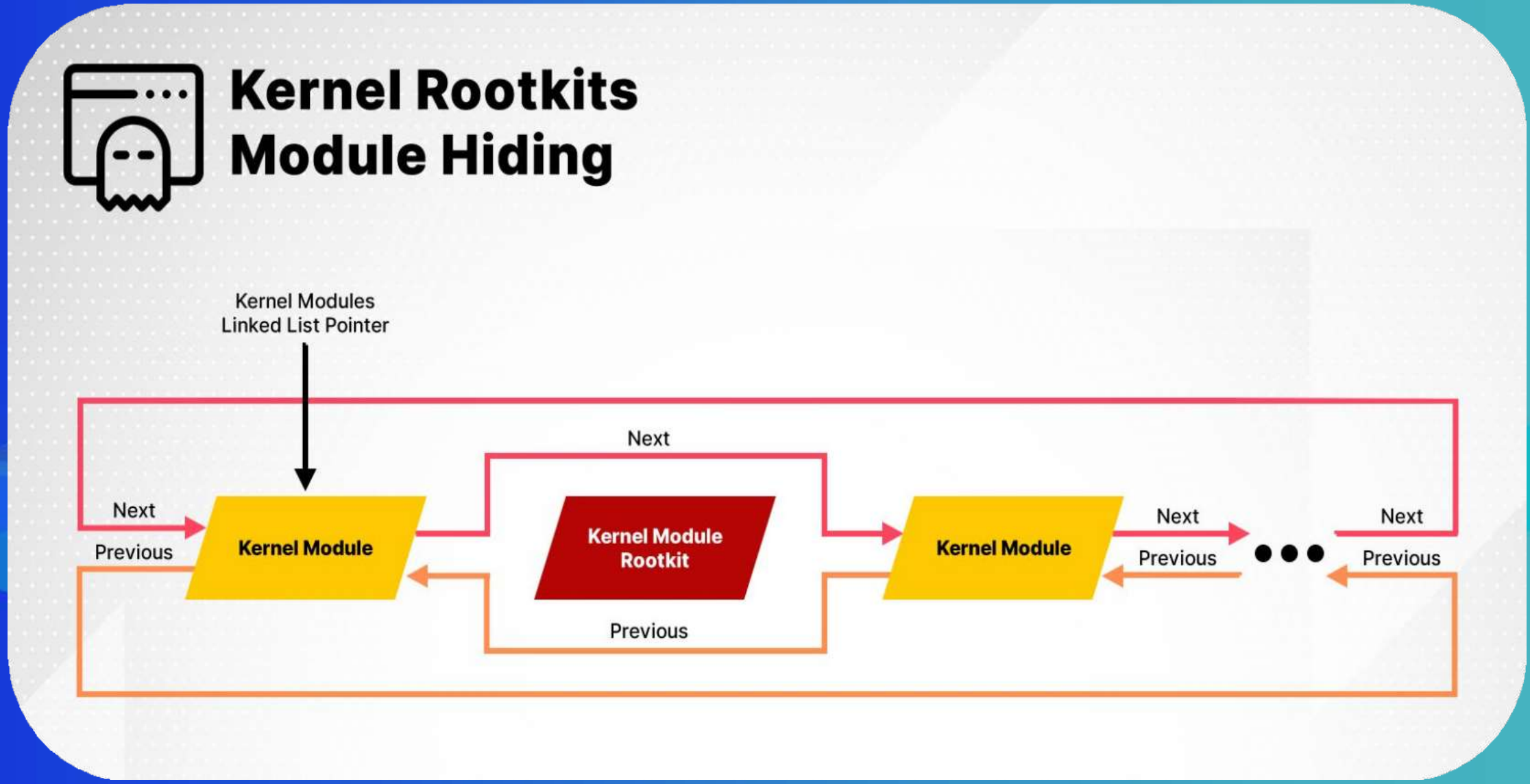
```
diamorphine.c  diamorphine.h  Makefile
root@ubuntu:/home/work/rootkit_study/Diamorphine# make
make -C /lib/modules/5.15.0-136-generic/build M=/home/work/rootkit_study/Diamorphine modules
make[1]: Entering directory '/usr/src/linux-headers-5.15.0-136-generic'
  CC [M]  /home/work/rootkit_study/Diamorphine/diamorphine.o
  MODPOST /home/work/rootkit_study/Diamorphine/Module.symvers
  CC [M]  /home/work/rootkit_study/Diamorphine/diamorphine.mod.o
  LD [M]  /home/work/rootkit_study/Diamorphine/diamorphine.ko
  BTF [M] /home/work/rootkit_study/Diamorphine/diamorphine.ko
Skipping BTF generation for /home/work/rootkit_study/Diamorphine/diamorphine.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-136-generic'
root@ubuntu:/home/work/rootkit_study/Diamorphine# insmod diamorphine.ko
root@ubuntu:/home/work/rootkit_study/Diamorphine#
```

二、内核模块隐藏

二、内核模块组织形式


Kernel Rootkits
Module Hiding
Kernel Modules
Linked List Pointer
Next
Previous
Kernel Module
Next
Previous
Kernel Module
Next
Previous
Kernel Module
Next
Previous
Next
Previous

二、内核模块隐藏原理


Kernel Rootkits
Module Hiding
Kernel Modules
Linked List Pointer
Next
Next
Previous
Kernel Module
Kernel Module
Rootkit
Previous
Kernel Module
Next
Previous
Next
Previous

二、内核模块隐藏实验

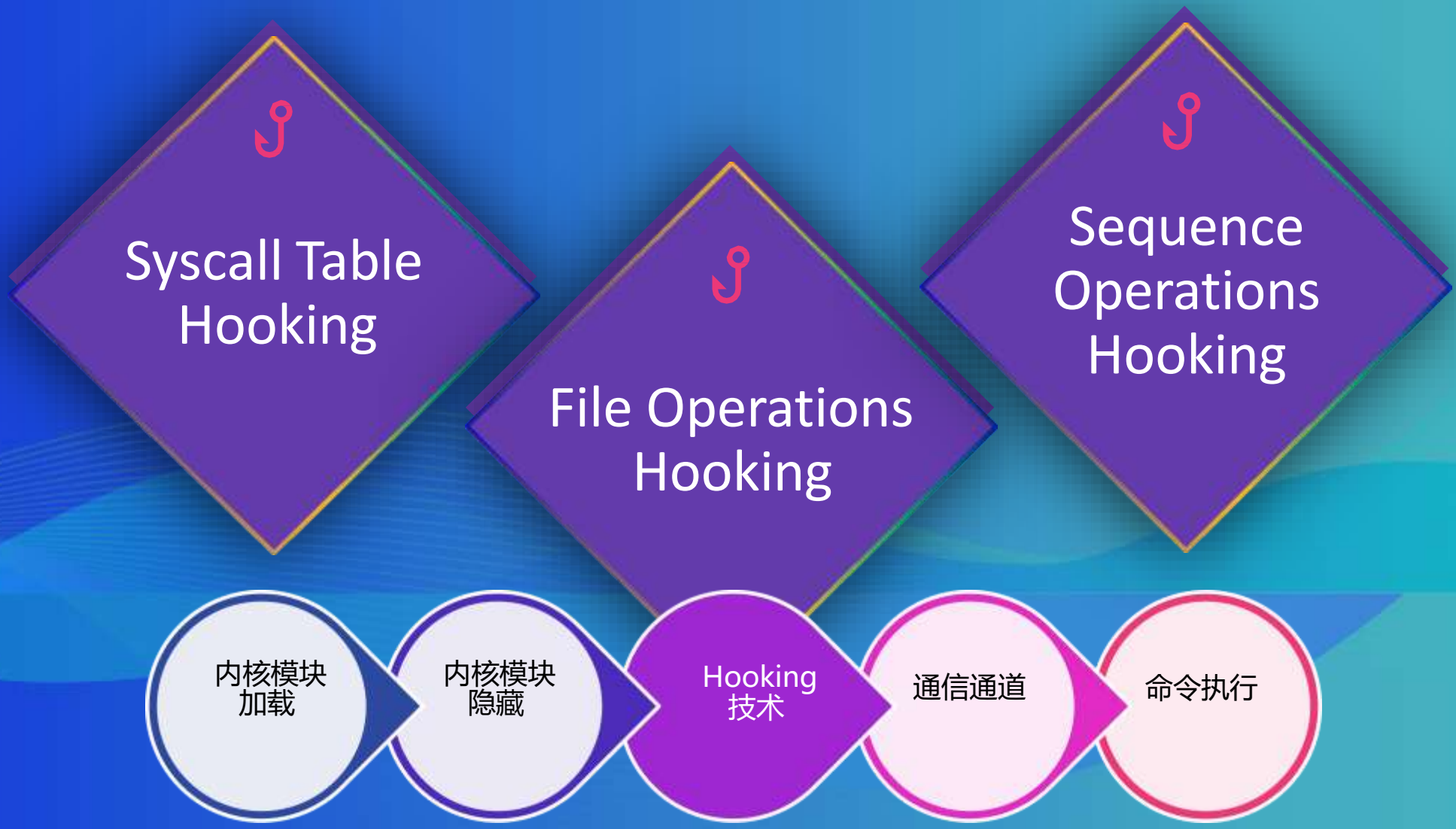
```
root@ubuntu:/home/work/goProject/src/tracee/dist# ./tracee -e hidden_kernel_module
TIME             UID    COMM             PID     TID     RET              EVENT                ARGS
08:01:06:954117  0                       0       0       0                hidden_kernel_module     addre
ss: 0xffffffffc0bb50c0, name: diamorphine, srcversion: 6268FF5B2D94BEC9AA68E39
^C
End of events stream
{"Stats":{"EventCount":1,"EventsFiltered":0,"NetCapCount":0,"BPFLogsCount":0,"ErrorCount":0,"LostEvCount":0,"LostWrCou
nt":0,"LostNtCapCount":0,"LostBPFLogsCount":0}}
root@ubuntu:/home/work/goProject/src/tracee/dist#
```

1 内核模块加载 x +

```
  LD [M]  /home/work/rootkit_study/Diamorphine/diamorphine.ko
  BTF [M] /home/work/rootkit_study/Diamorphine/diamorphine.ko
Skipping BTF generation for /home/work/rootkit_study/Diamorphine/diamorphine.ko due to unavailability of vmlinux
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-136-generic'
root@ubuntu:/home/work/rootkit_study/Diamorphine# insmod diamorphine.ko
root@ubuntu:/home/work/rootkit_study/Diamorphine#
```

三、Hooking技术

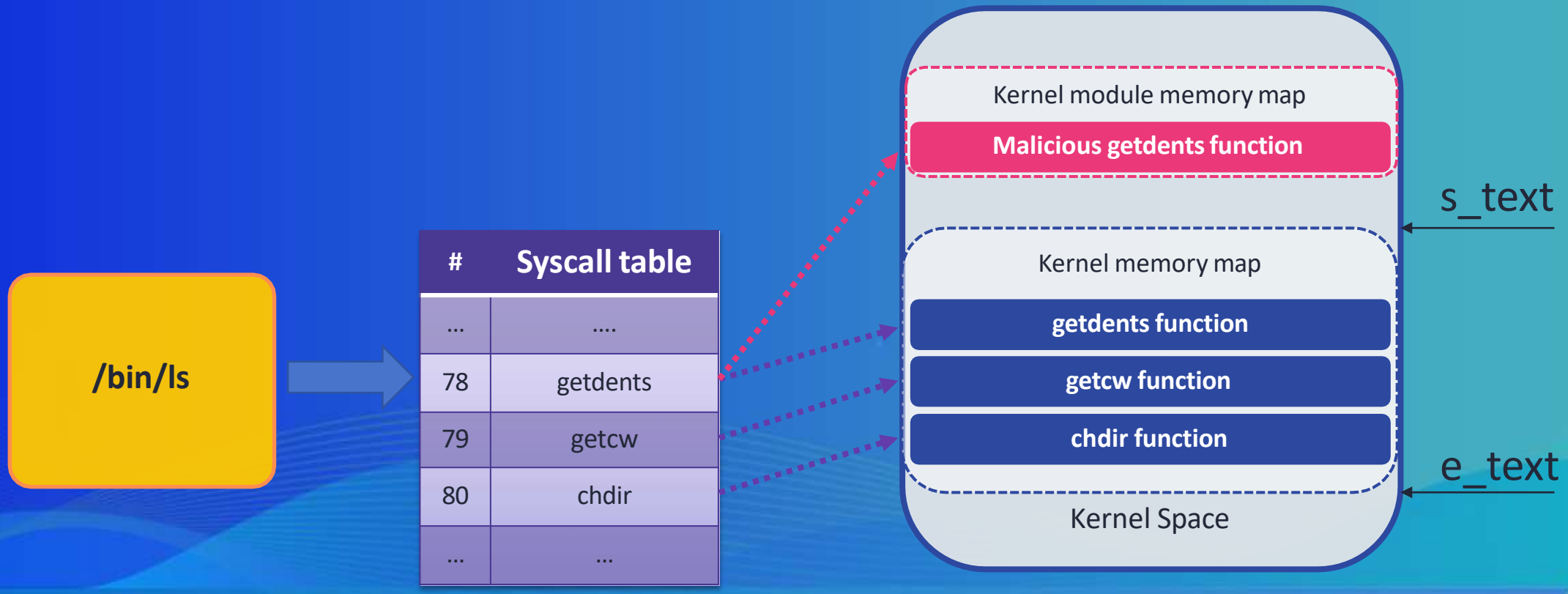
三、Hooking技术


Syscall Table Hooking
File Operations Hooking
Sequence Operations Hooking
内核模块加载
内核模块隐藏
Hooking技术
通信通道
命令执行

三、Hooking技术

1、Syscall Table Hooking

Hooking Syscall table 原理



Hooking Syscall table Diamorphine示例

```
orig_getdents = (orig_getdents_t)__sys_call_table[__NR_getdents];
orig_getdents64 = (orig_getdents64_t)__sys_call_table[__NR_getdents64];
orig_kill = (orig_kill_t)__sys_call_table[__NR_kill];


unprotect_memory();

__sys_call_table[__NR_getdents] = (unsigned long) hacked_getdents;
__sys_call_table[__NR_getdents64] = (unsigned long) hacked_getdents64;
__sys_call_table[__NR_kill] = (unsigned long) hacked_kill;
```

Syscall table Hooking演示视频截图

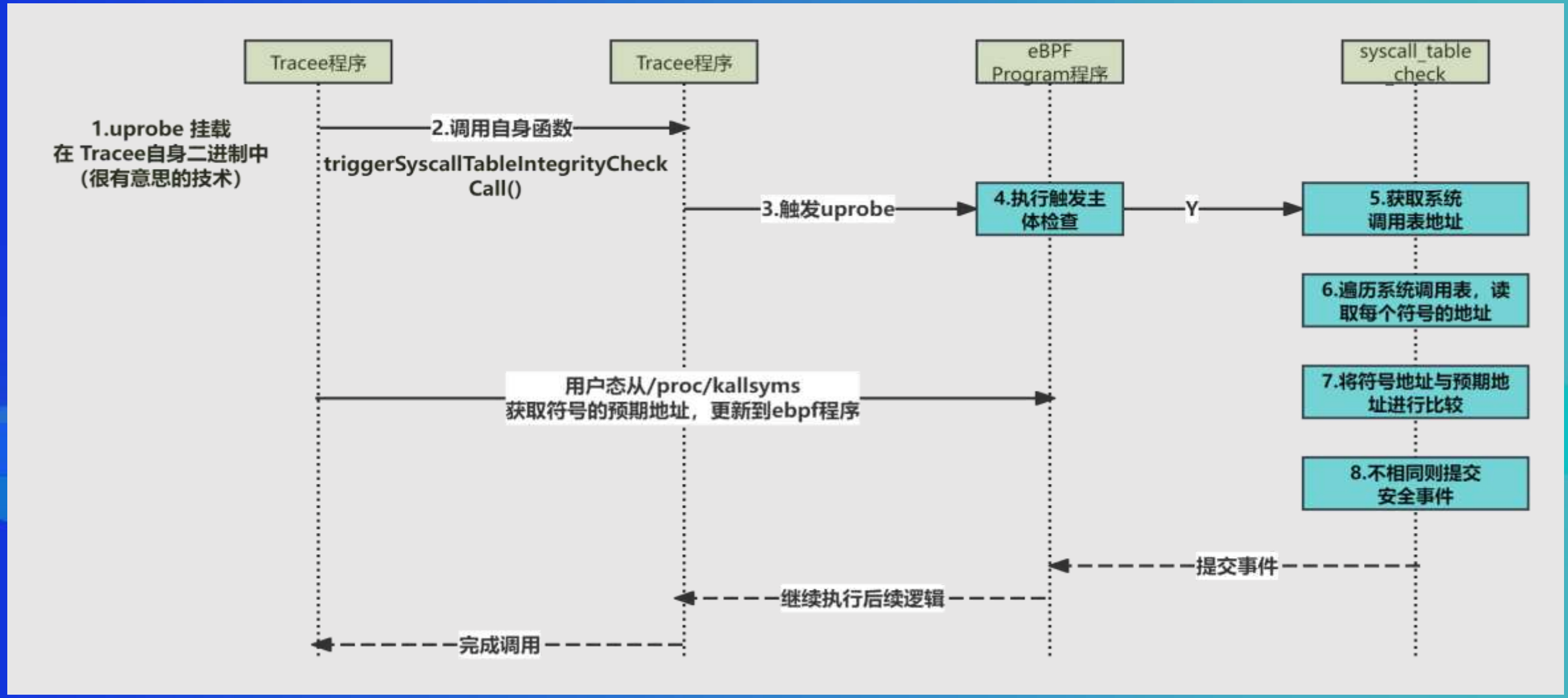
1 ebpf检测syscall table hoo...

```
btfhub  libbpf  signatures  tracee  tracee.bpf.o  tracee-ebpf  tracee-rules
root@ubuntu:/home/work/goProject/src/tracee/dist#
root@ubuntu:/home/work/goProject/src/tracee/dist# ./tracee-ebpf -e hooked_syscall
TIME             UID    COMM             PID     TID     RET             EVENT                ARGS
21:23:28:078791  0                       0       0       0               hooked_syscall       syscall: kill, add
ress: ffffffffc0bb3500, function: , owner:
21:23:28:078791  0                       0       0       0               hooked_syscall       syscall: getdents,
 address: ffffffffc0bb3050, function: , owner:
21:23:28:078791  0                       0       0       0               hooked_syscall       syscall: getdents6
4, address: ffffffffc0bb3230, function: , owner:
```

1 syscall table hooking模拟

```
make[1]: Leaving directory '/usr/src/linux-headers-5.15.0-136-generic'
root@ubuntu:/home/work/rootkit_study/Diamorphine# insmod diamorphine.ko
root@ubuntu:/home/work/rootkit_study/Diamorphine#
root@ubuntu:/home/work/rootkit_study/Diamorphine#
root@ubuntu:/home/work/rootkit_study/Diamorphine#
root@ubuntu:/home/work/rootkit_study/Diamorphine# ls
diamorphine.c  diamorphine.ko   diamorphine.mod.c  diamorphine.o  modules.order
diamorphine.h  diamorphine.mod  diamorphine.mod.o  Makefile       Module.symvers
```

ebpf检测Syscall Table Hooking原理和流程图



ebpf检测Syscall Table Hooking代码剖析

```
// syscall_table_check
SEC("uprobe/syscall_table_check")
int uprobe_syscall_table_check(struct pt_regs *ctx)
{
    //构建SYSCALL_TABLE_CHECK事件的相关数据
    program_data_t p = {};
    if (!init_program_data(&p, ctx, SYSCALL_TABLE_CHECK))
        return 0;

    // 检查触发uprobe进程的是否为tracee自身。
    if (p.config->tracee_pid != p.task_info->context.pid &&
        p.config->tracee_pid != p.task_info->context.host_pid)
        return 0;

    // Uprobes不是由syscalls触发的，所以需要设置标记.
    p.event->context.syscall = NO_SYSCALL;

    // 调用syscall_table_check去检查系统调用表
    syscall_table_check(&p);

    return 0;
}
```

ebpf检测Syscall Table Hooking代码剖析

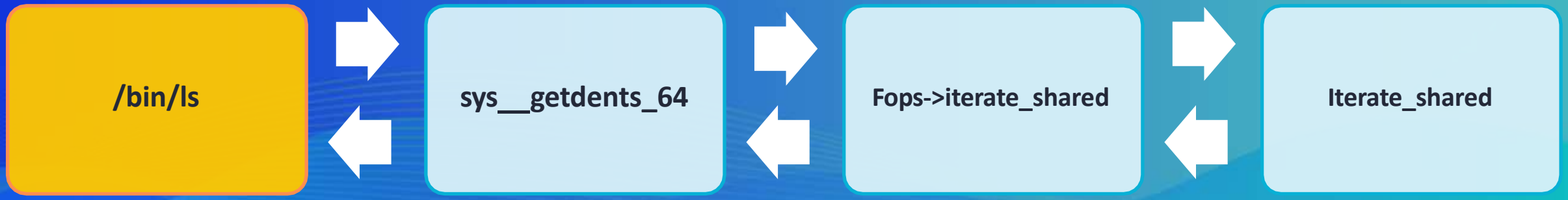
```
statfunc void syscall_table_check(program_data_t *p)
{
    char sys_call_table_symbol[15] = "sys_call_table";
    u64 *sys_call_table = (u64 *) get_symbol_addr(sys_call_table_symbol);//获取sys_call_table符号的地址
#pragma unroll
    for (int i = 0; i < 500; i++) {
        index = i;
        //使用index索引值（范围0~499）来查看正确的系统调用的地址
        syscall_table_entry_t *expected_entry =
            bpf_map_lookup_elem(&expected_sys_call_table, &index);

        //遍历系统调用表，并读取sys_call_table中符号的地址
        u64 effective_address;
        bpf_probe_read_kernel(&effective_address, sizeof(u64), sys_call_table + index);

        //将每个系统调用的实际地址与预期地址（存储在 expected_sys_call_table 映射中）进行比较
        //如果相同则跳过
        if (expected_entry->address == effective_address)
            continue;

        save_to_submit_buf(&(p->event->args_buf), &index, sizeof(int), 0);
        save_to_submit_buf(&(p->event->args_buf), &effective_address, sizeof(u64), 1);
        events_perf_submit(p, 0);
    }
}
```

三、Hooking技术

2、File Operation Hooking

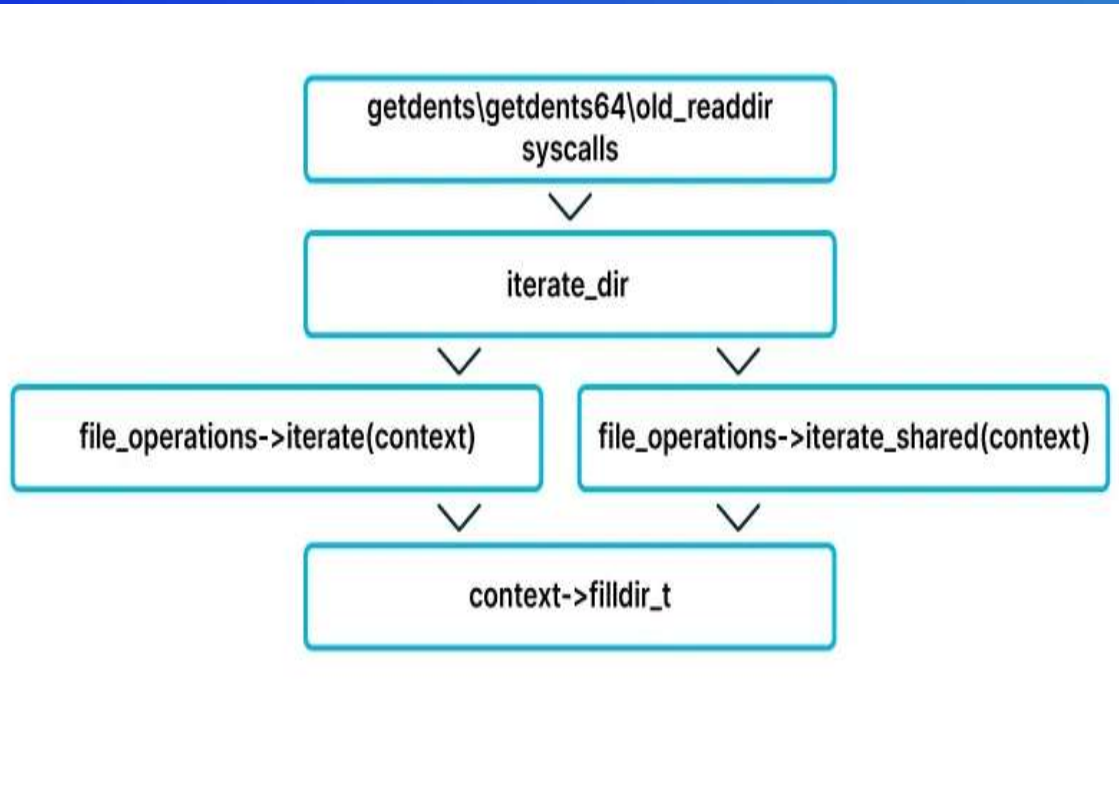
File Operation Hook前

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    ...
};
```



我们以/bin/ls举例，当执行/bin/ls命令时，会调用sys__getdents_64函数，该调用会使用目录项的iterate_shared或iterate文件操作，从其注册的回调函数中获取响应并返回给用户。

File Operation Hook前




```c
// 典型的目录遍历结构
struct dir_context {
    filldir_t actor;    // 填充目录项的函数指针
    loff_t pos;         // 当前位置
};

// filldir_t 函数类型定义
typedef int (*filldir_t)(struct dir_context *, const char *name, int namlen,
                         loff_t offset, u64 ino, unsigned int d_type);

// file_operations 结构体示例
struct file_operations {
    // ...
    int (*iterate)(struct file *, struct dir_context *);
    int (*iterate_shared)(struct file *, struct dir_context *);
    // ...
};
```

1、从getdents和getdents64系统调用开始，都会调用iterate_dir 函数
2、iterate_dir会根据标志位来调用iterate_shared或iterate函数
3、iterate_shared和iterate函数都会调用其函数参数dir_context的actor成员

File Operation Hook后

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    int (*iterate) (struct file *, struct dir_context *);
    int (*iterate_shared) (struct file *, struct dir_context *);
    ...
};
```

为了hook iterate/iterate_shared这个位置，需要覆写file_operations结构体中文件操作的地址（iterate/iterate_shared的函数地址），当文件操作被调用时，会执行被劫持的函数，劫持的函数可以修改返回结果。

File Operation Hook演示视频截图

```
root@ubuntu:/home/work/goProject/src/tracee/dist# ./tracee-ebpf -e hooked_proc_fops
TIME             UID    COMM             PID     TID     RET              EVENT                ARGS
01:00:48:907532  0      ps               5082    5082    0                hooked_proc_fops     hooked_fops_pointe
rs: [{iterate_shared phide}]
01:00:48:927566  0      ps               5082    5082    0                hooked_proc_fops     hooked_fops_pointe
rs: [{iterate_shared phide}]
```

1 syscall table hooking模拟 | 2 ubuntu 20.04 | +

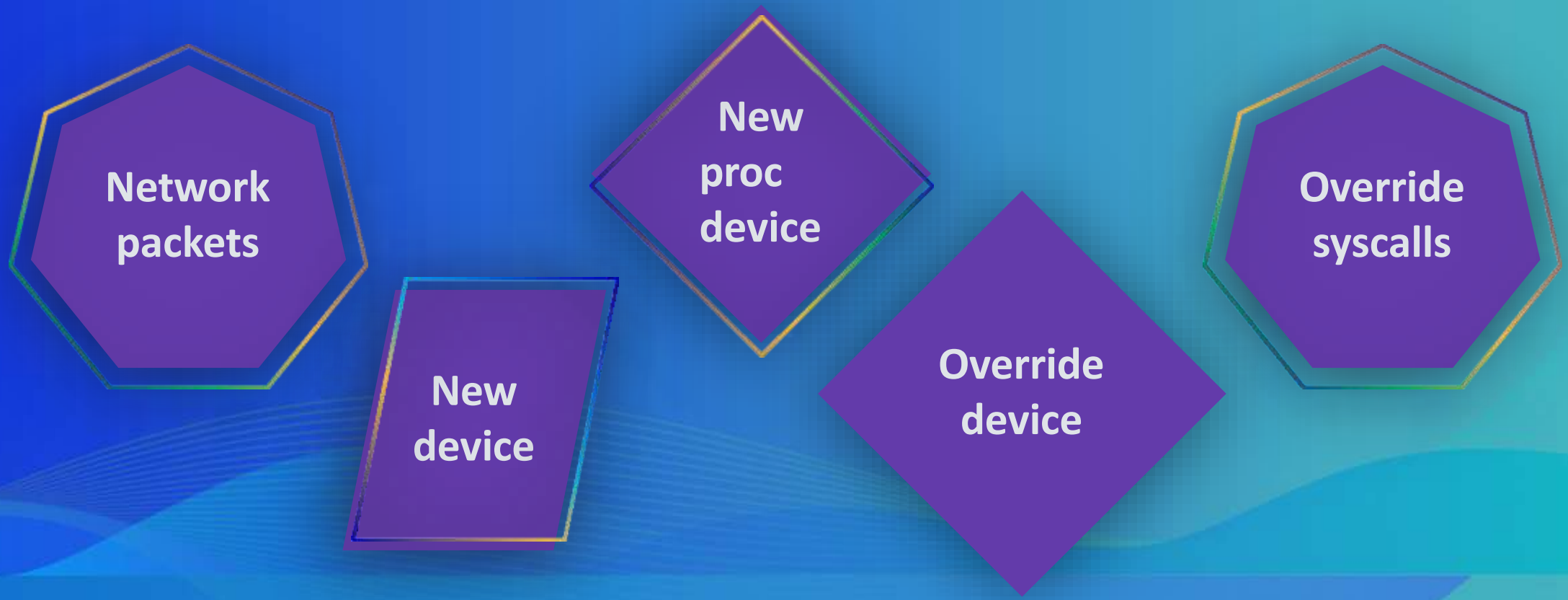
```
root@ubuntu:/home/work/rootkit_study/phide# insmod phide.ko
root@ubuntu:/home/work/rootkit_study/phide# ps aux
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           2  0.0  0.0      0     0 ?        S    Apr03   0:00 [kthreadd]
root           3  0.0  0.0      0     0 ?        I<   Apr03   0:00 [rcu_gp]
root           4  0.0  0.0      0     0 ?        I<   Apr03   0:00 [rcu_par_gp]
root           5  0.0  0.0      0     0 ?        I<   Apr03   0:00 [slub_flushwq]
root           6  0.0  0.0      0     0 ?        I<   Apr03   0:00 [netns]
root           8  0.0  0.0      0     0 ?        I<   Apr03   0:00 [kworker/0:0H-events_highpri]
root          20  0.0  0.0      0     0 ?        S    Apr03   0:00 [idle_inject/1]
root          21  0.0  0.0      0     0 ?        S    Apr03   0:01 [migration/1]
```

tracee检测原理和流程图

tracee检测原理代码剖析

```
SEC("kprobe/security_file_permission")
int BPF_KPROBE(trace_security_file_permission)
{
    // 判断是否是 procfs 文件系统
    if (s_magic != PROC_SUPER_MAGIC) {
        return 0;
    }

    // 获取文件操作函数表
    struct file_operations *fops = (struct file_operations *) BPF_CORE_READ(f_inode, i_fop);

    // 获取 iterate 和 iterate_shared 函数指针
    unsigned long iterate_addr = (unsigned long) BPF_CORE_READ(fops, iterate);
    unsigned long iterate_shared_addr = (unsigned long) BPF_CORE_READ(fops, iterate_shared);

    // 获取内核文本段边界
    void *stext_addr = get_stext_addr();
    void *etext_addr = get_etext_addr();

    // 如果函数指针在内核文本段内，标记为 0（表示正常）
    if (iterate_shared_addr >= (u64) stext_addr && iterate_shared_addr < (u64) etext_addr)
        iterate_shared_addr = 0;
    if (iterate_addr >= (u64) stext_addr && iterate_addr < (u64) etext_addr)
        iterate_addr = 0;

    // 将可能被钩子劫持的函数指针地址保存到数组中
    unsigned long fops_addresses[2] = {iterate_shared_addr, iterate_addr};

    // 将数组保存到事件参数缓冲区并提交事件
    save_u64_arr_to_buf(&p.event->args_buf, (const u64 *) fops_addresses, 2, 0);
    events_perf_submit(&p, 0);
    return 0;
}
```

四、通信通道


Network packets
New proc device
Override syscalls
New device
Override device

通信通道建立之覆盖syscall table项

```
asmlinkage int hacked_kill(pid_t pid, int sig)
{
    struct task_struct *task;

    switch (sig) {
        case SIGINVIS:          // 处理进程隐藏信号
            if ((task = find_task(pid)) == NULL)
                return -ESRCH;

            // 通过切换PF_INVISIBLE标志位，控制进程的隐藏和显示
            task->flags ^= PF_INVISIBLE;
            break;

        case SIGMODINVIS:    // 处理模块隐藏信号
            // 根据模块当前状态决定是显示还是隐藏
            if (module_hidden)
                module_show();      // 如果模块已隐藏，则显示模块
            else
                module_hide();      // 如果模块可见，则隐藏模块
            break;

        default:                // 处理普通信号
            // 对于普通信号，调用原始的kill系统调用处理
            return orig_kill(pid, sig);
    }

    return 0;
}
```

Diamorphine rootkit采用了一种巧妙的通信方案。
通过劫持Linux系统的kill系统调用来建立用户态和内核态之间的隐蔽通信通道。

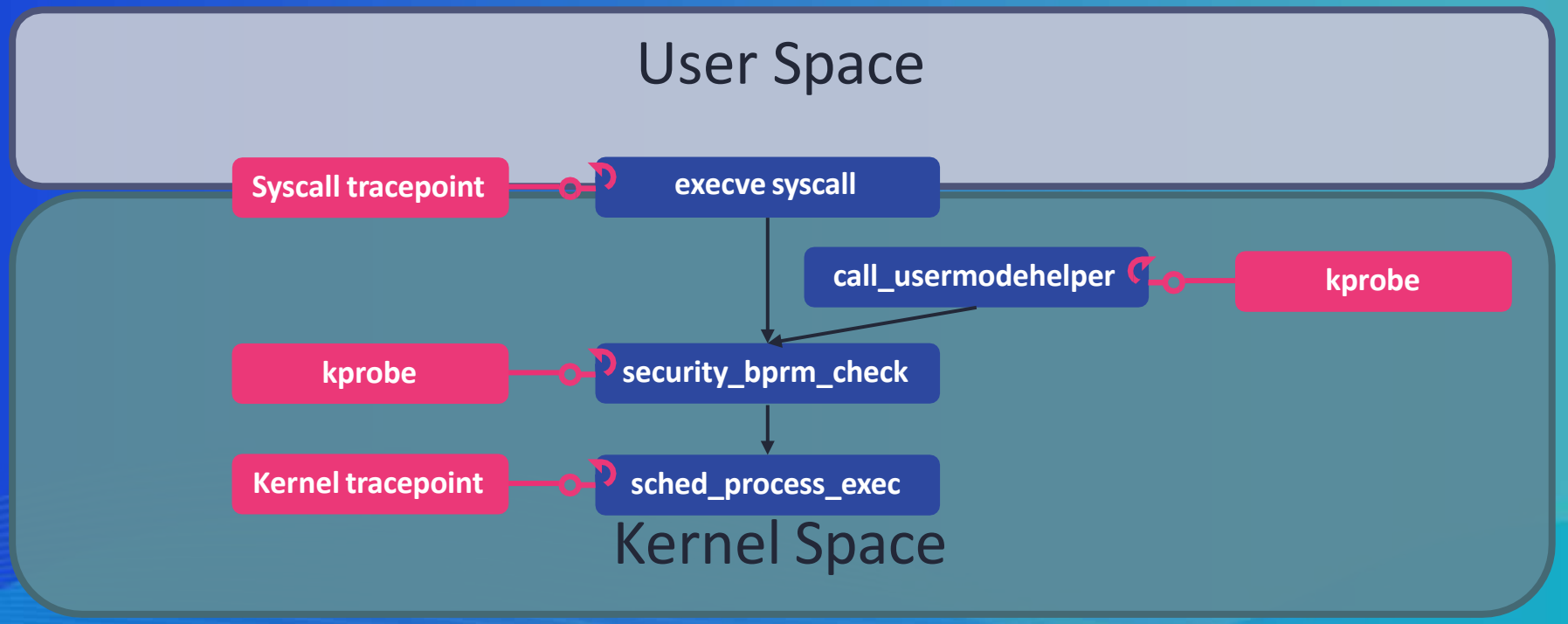
这种方式具有极强的隐蔽性，因为kill是Linux系统中使用频率较高的系统调用，不会引起特别的注意。

用法：
当发送信号64时，隐藏/显示rootkit模块
当发送信号63时，隐藏/显示指定进程

tracee检测原理和流程图

1. proc文件系统File Operation Hooking检测
2. Tracee 通过挂钩 Linux LSM 中与网络套接字相关的关键函数,如 connect、listen、accept 和 sendmsg,从而检测异常网络连接行为。
3. Syscall table完整性检查（syscall）
4. 对于新建设备的监，SEC("kprobe/device_add")

五、命令执行(重点)



1、syscall tracepoint针对execve syscall函数

2、LSM hook kprobe针对security_bprm_check函数

3、Kernel tracepoint针对sched_process_exec函数

4、kprobe call_usermodehelper内核函数

五、命令执行代码Demo示例

```c
static __init int test_driver_init(void)
{
    int result = 0;
    char cmd_path[] = "/usr/bin/touch";
    char* cmd_argv[] = {cmd_path, FILE_PATH, NULL};
    char* cmd_envp[] = {"HOME=/", "PATH=/sbin:/bin:/usr/bin", NULL};

    /* 创建文件 */
    result = call_usermodehelper(cmd_path, cmd_argv, cmd_envp, UMH_WAIT_PROC);
    printk(KERN_DEBUG "test driver init exec! there result of call_usermodehelper is %d\n", result);
    printk(KERN_DEBUG "test driver init exec! the process is \"%s\", pid is %d.\n",
           current->comm, current->pid);

    /* 写入内容 */
    if (result == 0) {
        result = write_file(FILE_PATH, CONTENT);
        if (result < 0) {
            printk(KERN_ERR "Failed to write content to file\n");
        }
    }

    return result;
}

static __exit void test_driver_exit(void)
{
    int result = 0;
    char cmd_path[] = "/bin/rm";
    char* cmd_argv[] = {cmd_path, FILE_PATH, NULL};
    char* cmd_envp[] = {"HOME=/", "PATH=/sbin:/bin:/usr/bin", NULL};

    result = call_usermodehelper(cmd_path, cmd_argv, cmd_envp, UMH_WAIT_PROC);
    printk(KERN_DEBUG "test driver exit exec! the result of call_usermodehelper is %d\n", result);
    printk(KERN_DEBUG "test driver exit exec! the process is \"%s\", pid is %d \n",
           current->comm, current->pid);
}
```

五、命令执行代码Demo执行效果

```
root@ubuntu:/home/work/rootkit_study/CommandExecute# ls
call_usermodehelper_test.c    call_usermodehelper_test.mod.c  Makefile
call_usermodehelper_test.ko   call_usermodehelper_test.mod.o  modules.order
call_usermodehelper_test.mod  call_usermodehelper_test.o      Module.symvers
root@ubuntu:/home/work/rootkit_study/CommandExecute# insmod call_usermodehelper_test.ko
insmod: ERROR: could not insert module call_usermodehelper_test.ko: File exists
root@ubuntu:/home/work/rootkit_study/CommandExecute# rmmod call_usermodehelper_test
root@ubuntu:/home/work/rootkit_study/CommandExecute# insmod call_usermodehelper_test.ko
root@ubuntu:/home/work/rootkit_study/CommandExecute# cat /tmp/touchX.txt
hello world,my name is haolipeng.
root@ubuntu:/home/work/rootkit_study/CommandExecute#
```

五、开源项目中命令执行的实现

Reptile浅析

Reptile开源中call_usermodehelper代码示例

```c
static inline int exec(char **argv)
{
        char *envp[] = {"PATH=/sbin:/bin:/usr/sbin:/usr/bin", NULL};
        return call_usermodehelper(argv[0], argv, envp, UMH_WAIT_EXEC);
}

static inline int run_cmd(char *cmd)
{
        char *argv[] = {"/bin/bash", "-c", cmd, NULL};
        return exec(argv);
}
```

五、命令执行实验视频截图

eitani@ubuntu: dmesg

```
root@ubuntu /h/n/dist# ./tracee -f e=do_init_module,call_usermodehelper,execve,sched_process_exec
TIME             UID    COMM             PID     TID     RET             EVENT                ARGS
05:18:09:392879  0      bash             4218    4218    0               execve               pathname: /sbin/insmod,
argv: [insmod reptile.ko]
05:18:09:413398  0      insmod           4218    4218    0               sched_process_exec   cmdpath: /sbin/insmod,
pathname: /bin/kmod, argv: [insmod reptile.ko], invoked_from_kernel: 0
05:18:09:654178  0      insmod           4218    4218    0               call_usermodehelper  pathname: /bin/bash, argv:
[/bin/bash -c /reptile/reptile_start], envp: [PATH=/sbin:/bin:/usr/sbin:/usr/bin], wait: 1
05:18:09:660681  0      bash             4220    4220    0               sched_process_exec   cmdpath: /bin/bash,
pathname: /bin/bashargv: [/bin/bash -c /reptile/reptile_start], invoked_from_kernel: 1
05:18:09:674843  0      insmod           4218    4218    0               do_init_module       name: reptile_module,
version: , src_version: 10902AB483D20A668D9AD54, prev: 0xffffffff92ebce20, next: 0xffffffffc0d025c8, prev_next:
0xffffffffc0b441c8, next_prev: 0x0
```

tracee检测原理代码剖析

```
SEC("kprobe/call_usermodehelper")
int BPF_KPROBE(trace_call_usermodehelper)
{
    program_data_t p = {};
    if (!init_program_data(&p, ctx, CALL_USERMODE_HELPER))
        return 0;

    if (!evaluate_scope_filters(&p))
        return 0;

    void *path = (void *) PT_REGS_PARM1(ctx);
    unsigned long argv = PT_REGS_PARM2(ctx);
    unsigned long envp = PT_REGS_PARM3(ctx);
    int wait = PT_REGS_PARM4(ctx);

    save_str_to_buf(&p.event->args_buf, path, 0);
    save_str_arr_to_buf(&p.event->args_buf, (const char *const *) argv, 1);
    save_str_arr_to_buf(&p.event->args_buf, (const char *const *) envp, 2);
    save_to_submit_buf(&p.event->args_buf, (void *) &wait, sizeof(int), 3);

    return events_perf_submit(&p, 0);
}
```

内核态处理：
用于监控Linux内核中的call_usermodehelper函数的ebpf kprobe，捕获以下参数：

- 待执行的用户空间程序的路径path
- 传递给程序的参数数组argv
- 环境变量数组envp
- 是否等待程序执行完成的标志wait
- 调用events_perf_submit函数将捕获的信息发送到用户空间

用户态处理逻辑：

- 可疑的用户态程序执行
- 异常的命令行参数
- 异常的环境变量设置

tracee检测原理和流程图

六、总结


从内存加载内核模块
内核双向链表分离
系统调用表和fops Hooking技术
丰富多样的通信通道
从内核态创建进程
内核模块加载
内核模块隐藏
Hook技术
通信通道
命令执行

六、总结

谢谢大家的观看
欢迎交流