Linux 内核监控在 Android 攻防中的应用

原创 evilpan 有价值炮灰 2022-01-04 20:30

收录于话题
#Android 9 #Linux 9

背景

在日常分析外部软件时，遇到的反调试/反注入防护已经越来越多，之前使用的基于 frida 的轻量级沙盒已经无法满足这类攻防水位的需要，因此需要有一种更加深入且通用的方式来对 APP 进行全面的监测和绕过。本文即为对这类方案的一些探索和实践。

为了实现对安卓 APP 的全面监控，需要知道目标应用访问/打开了哪些文件，执行了哪些操作，并且可以修改控制这些操作的返回结果。一个直观的想法是通过 libc 作为统一收口来对应用行为进行收集，比如接管 open/openat/faccess/fstatat 实现文件访问监控以及进一步的文件重定向。

然而，现今许多聪明的加固 APP 都使用了内联系统调用汇编来绕过 libc 实现暗度陈仓，在 binary 层再加上控制流混淆、花指令、代码加密甚至是 VMP 等成熟的防御措施，使得识别这类隐藏调用变得十分困难。某不知名安全研究员[1]曾经说过:

> Never to wrestle with a pig. You get dirty, and besides, the pig likes it.

因此，我们不应该在应用层上和加固厂商做对抗，而是寻找其他突破点，以四两拨千斤的方式实现目的。当然，如果只是为了练手，那手撕虚拟机也是可以的 :)

现有方案

在介绍内核监控技术之前，我们先来看看目前已有的一些方案，以及它们的不足之处。

strace

strace[2] 是 Linux 中一个知名的用户态系统调用跟踪工具，可以输入目标进程所执行的系统调用的名称以及参数，常用于快速的应用调试和诊断。strace 的示例输出如下所示:

```
$ strace echo evilpan
execve("/usr/bin/echo", ["echo", "evilpan"], 0x7fe55d5d18 /* 56 vars */) =
brk(NULL)                               = 0x57b1bd2000
faccessat(AT_FDCWD, "/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=19285, ...}) = 0
mmap(NULL, 19285, PROT_READ, MAP_PRIVATE, 3, 0) = 0x79aecf8000
close(3)                                = 0

openat(AT_FDCWD, "/lib/aarch64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) =
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\3\0\267\0\1\0\0\0p\16\2\0\0\0\0
fstat(3, {st_mode=S_IFREG|0777, st_size=1439544, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
mmap(NULL, 1511520, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0)
mprotect(0x79aecb7000, 61440, PROT_NONE) = 0
mmap(0x79aecc6000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
mmap(0x79aeccc000, 12384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_
close(3)                                = 0
mprotect(0x79aecc6000, 16384, PROT_READ) = 0
mprotect(0x5787f5f000, 4096, PROT_READ) = 0
mprotect(0x79aecff000, 4096, PROT_READ) = 0
munmap(0x79aecf8000, 19285)             = 0
brk(NULL)                               = 0x57b1bd2000
brk(0x57b1bf3000)                       = 0x57b1bf3000
fstat(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(0x88, 0x2), ...}) = 0
write(1, "evilpan\n", 8evilpan
)                 = 8
close(1)                                = 0
close(2)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

对于需要监控系统调用的场景，strace 是个非常合适的工具，因为它基于 PTRACE_SYSCALL 去跟踪并基于中断的方式去接管所有系统调用，因此即便目标使用了不依赖 libc 的内联 svc 也可以被识别到。不过这个缺点也很明显，从名称也看出来，本质上该程序是基于 ptrace 对目标进行跟踪，因此如果对方代码中有反调试措施，那么就很有可能被检测到。

另外在 Android 系统中，APP 进程都是由 zygote fork 而出，因此使用 strace 比较不容易确定跟踪时机，而且由于许多应用有多个进程，就需要对输出结果进行额外的过滤和清洗。

更多关于 strace 的实现原理可以参考: How does strace work?[3]

jtrace

在早期 strace 程序还不支持 arm64，因此 *Jonathan Levin* 在编写 **Android Internal** 一书时就写了 jtrace 这个工具，旨在用于对 Android 应用的跟踪。虽然现在 Google 也在 AOSP 中支持了 strace，但 jtrace 仍然有其独特的优点：

- 支持系统属性的访问监控 (setprop/getprop)
- 支持输入事件的监控 (InputReader)
- 支持 Binder 信息的解析
- 支持 AIDL 的解析
- ……

虽然 jtrace 是闭源的，但提供了独特的插件功能，用户可以根据其提供的接口去编写一个插件(动态库)，并使用 `--plugin` 参数或者 `JTRACE_EXT_PATH` 环境变量指定的路径加载插件，从而实现自定义的系统调用参数解析处理。

虽然优点比 strace 多了不少，但其缺点并没有解决，jtrace 本身依然是基于 **PTRACE_SYSCALL** 进行系统调用跟踪的，因此还是很容易被应用的反调试检测到。

详见: http://newandroidbook.com/tools/jtrace.html

Frida

frida[4] 是目前全球最为知名的动态跟踪工具集 (Instrumentation)，支持使用 js 脚本来对目标应用程序进行动态跟踪。相信读者对于 frida 已经不陌生，这里也就不再过多介绍。其功能之丰富毋庸置疑，但也有一些硬伤，比如：

- frida-gum 基于 inline-hook 对目标跟踪代码进行实时重编译 (JIT)，对于应用本身有较大的侵入性；
- frida-inject 需要依赖 ptrace 对目标应用进行第一次注入并加载 agent，有一个较短的注入窗口可能会被反调试应用检测到；
- frida 目前尚不支持系统调用事件级别的追踪，虽然 frida-stalker 可以做到汇编级别，但是开销过大；
- frida 太过知名，以至于有很多针对 frida 的特征检测；
- ……

类似的 Instrumentation 工具还有 QDBI[5]，hookzz[6] 等等。

其他

除了上面提到的这些工具，还有很多其他工具可以进行动态监控，比如 ltrace、gdb 等但这些工具都不能完美实现我的需求。既要马儿跑得快(开销小)，又要马儿不吃草(无侵入)，那我们就只有把眼光放向内核了。

Kernel Tracing 101

如果目标是为了实现系统调用监控，以及部分系统调用参数的修改(例如 IO 重定向)，那么一个直观的想法是修改内核源码，在我们感兴趣的系统调用入口插入自己的代码实现具体功能。但是这样非常低效，一来我们要在不同的系统调用相关函数中增加代码，引入过多修改后会导致更新内核合并上游提交变得困难；二来我们每次修改后都需要重新编译内核以及对应的 AOSP 代码(因为内核在 boot.img 中，详见后文)，再烧写到手机或模拟器中，流程过于复杂。

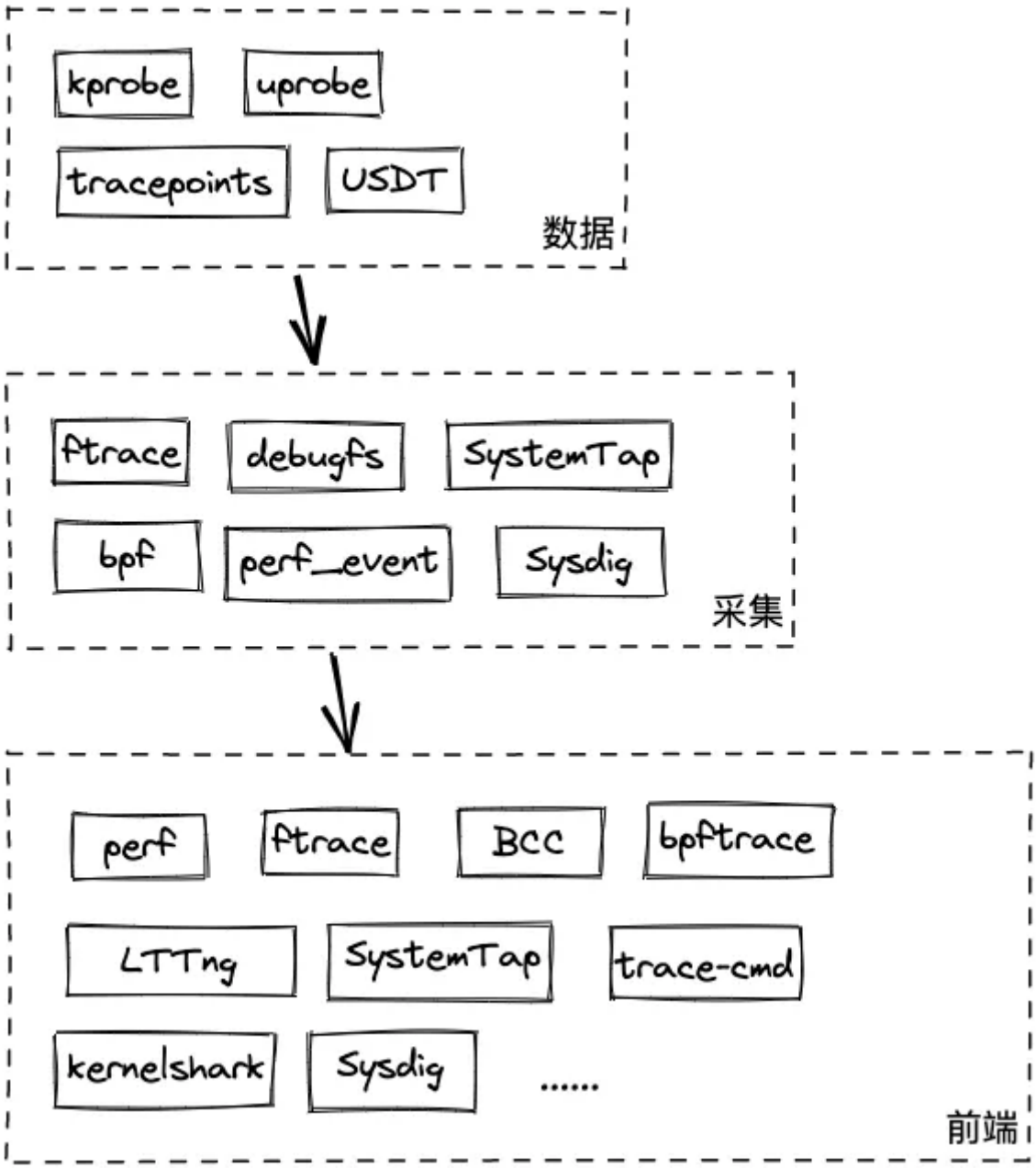
另外一个想法是通过在内核代码中引入一次性的 trampoline，然后在后续增加或者减少系统调用监控入口时通过内核模块的方式去进行修改。这样似乎稍微合理一些，但其实内核中已经有了许多类似的监控方案，这样做纯属重复造轮子，效率低不说还可能随时引入 kernel panic。

大局观

那么，内核中都有哪些监控方案？这其实不是一个容易回答的问题，我们在日常运维时听说过 kprobe、jprobe、uprobe、eBPF、tracefs、systemtab、perf，……到底他们之间的的关系是什么，分别都有什么用呢？

这里推荐一篇文章: Linux tracing systems & how they fit together[7]，根据其中的介绍，这些内核监控方案/工具可以分为三类:

1. 数据: 根据监控数据的来源划分
2. 采集: 根据内核提供给用户态的原始事件回调接口进行划分
3. 前端: 获取和解析监控事件数据的用户工具



trace

后面对这些监控方案分别进行简要的介绍。

kprobe

简单来说，kprobe 可以实现动态内核的注入，基于中断的方法在任意指令中插入追踪代码，并且通过 pre_handler/post_handler/fault_handler 去接收回调。

使用

参考 Linux 源码中的 **samples/kprobes/kprobe_example.c**，一个简单的 kprobe 内核模块实现如下：

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/kprobes.h>

#define MAX_SYMBOL_LEN  64
static char symbol[MAX_SYMBOL_LEN] = "_do_fork";
module_param_string(symbol, symbol, sizeof(symbol), 0644);

/* For each probe you need to allocate a kprobe structure */
static struct kprobe kp = {
    .symbol_name    = symbol,
};
/* kprobe pre_handler: called just before the probed instruction is execu
static int handler_pre(struct kprobe *p, struct pt_regs *regs)
{
    pr_info("<%s> pre_handler: p->addr = 0x%p, pc = 0x%lx\n", p->symbol_na
    /* A dump_stack() here will give a stack backtrace */
    return 0;
}
/* kprobe post_handler: called after the probed instruction is executed */
static void handler_post(struct kprobe *p, struct pt_regs *regs, unsigned
{
    pr_info("<%s> post_handler: p->addr = 0x%p\n", p->symbol_name, p->addr
}
/*
 * fault_handler: this is called if an exception is generated for any
 * instruction within the pre- or post-handler, or when Kprobes
 * single-steps the probed instruction.
 */
static int handler_fault(struct kprobe *p, struct pt_regs *regs, int trapn
{
    pr_info("fault_handler: p->addr = 0x%p, trap #%dn", p->addr, trapnr);
    /* Return 0 because we don't handle the fault. */
    return 0;
}
```

```
static int __init kprobe_init(void)
{
    int ret;
    kp.pre_handler = handler_pre;
    kp.post_handler = handler_post;
    kp.fault_handler = handler_fault;

    ret = register_kprobe(&kp);
    if (ret < 0) {
        pr_err("register_kprobe failed, returned %d\n", ret);
        return ret;
    }
    pr_info("Planted kprobe at %p\n", kp.addr);
    return 0;
}

static void __exit kprobe_exit(void)
{
    unregister_kprobe(&kp);
    pr_info("kprobe at %p unregistered\n", kp.addr);
}

module_init(kprobe_init)
module_exit(kprobe_exit)
MODULE_LICENSE("GPL");
```

安装该内核模块后，每当系统中的进程调用 fork，就会触发我们的 handler，从而在 dmesg 中输出对应的日志信息。值得注意的是，kprobe 模块依赖于具体的系统架构，上述 pre_handler 中我们打印指令地址使用的是 `regs->pc`，这是 ARM64 的情况，如果是 X86 环境，则对应 `regs->ip`，可查看对应 arch 的 `struct pt_regs` 实现。

原理

kprobe 框架基于中断实现。当 kprobe 被注册后，内核会将对应地址的指令进行拷贝并替换为断点指令(比如 X86 中的 `int 3`)，随后当内核执行到对应地址时，中断会被触发从而执行流程会被重定向到我们注册的 `pre_handler` 函数；当对应地址的原始指令执行完后，内核会再次执行 `post_handler` (可选)，从而实现**指令级别**的内核动态监控。也就是说，kprobe 不仅可以跟踪任意带有符号的内核函数，也可以跟踪函数中间的任意指令。

另一个 kprobe 的同族是 **kretprobe**，只不过是针对函数级别的内核监控，根据用户注册时提供的 `entry_handler` 和 `ret_handler` 来分别在函数进入时和返回前进行回调。当然实现上和 kprobe 也有所不同，不是通过断点而是通过 trampoline 进行实现，可以略为减少运行开销。

有人可能听说过 Jprobe，那是早期 Linux 内核的的一个监控实现，现已被 Kprobe 替代。

拓展阅读:

- An introduction to KProbes[8]
- Documentation/trace/kprobetrace.rst[9]
- samples/kprobes/kprobe_example.c[10]
- samples/kprobes/kretprobe_example.c[11]

uprobe

uprobe 顾名思义，相对于内核函数/地址的监控，主要用于用户态函数/地址的监控。听起来是不是有点神奇，内核怎么监控用户态函数的调用呢?

使用

站在用户视角，我们先看个简单的例子，假设有这么个一个用户程序:

```
// test.c
#include <stdio.h>
void foo() {
    printf("hello, uprobe!\n");
}
int main() {
    foo();
    return 0;
}
```

编译好之后，查看某个符号的地址，然后告诉内核我要监控这个地址的调用:

```
$ gcc test.c -o test
$ readelf -s test | grep foo
    87: 0000000000000764    32 FUNC    GLOBAL DEFAULT   13 foo
$ echo 'p /root/test:0x764' > /sys/kernel/debug/tracing/uprobe_events
```

```
$ echo 1 > /sys/kernel/debug/tracing/events/uprobes/p_test_0x764/enable
$ echo 1 > /sys/kernel/debug/tracing/tracing_on
```

然后运行用户程序并检查内核的监控返回:

```
$ ./test && ./test
hello, uprobe!
hello, uprobe!

$ cat /sys/kernel/debug/tracing/trace
# tracer: nop
#
# WARNING: FUNCTION TRACING IS CORRUPTED
#          MAY BE MISSING FUNCTION EVENTS
# entries-in-buffer/entries-written: 3/3   #P:8
#
#                              _-----=> irqs-off
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
            test-7958  [006] .... 34213.780750: p_test_0x764: (0x623621870
            test-7966  [006] .... 34229.054039: p_test_0x764: (0x5f586cb70
```

当然，最后别忘了关闭监控:

```
$ echo 0 > /sys/kernel/debug/tracing/tracing_on
$ echo 0 > /sys/kernel/debug/tracing/events/uprobes/p_test_0x764/enable
$ echo > /sys/kernel/debug/tracing/uprobe_events
```

原理

上面的接口是基于 debugfs (在较新的内核中使用 tracefs)，即读写文件的方式去与内核交互实现 uprobe 监控。其中写入 `uprobe_events` 时会经过一系列内核调用:

- probes_write
- create_trace_uprobe
- kern_path: 打开目标 ELF 文件；

- alloc_trace_uprobe: 分配 uprobe 结构体；
- register_trace_uprobe: 注册 uprobe；
- regiseter_uprobe_event: 将 probe 添加到全局列表中，并创建对应的 uprobe debugfs 目录，即上文示例中的 p_test_0x764；

当已经注册了 uprobe 的 ELF 程序被执行时，可执行文件会被 mmap 映射到进程的地址空间，同时内核会将该进程虚拟地址空间中对应的 uprobe 地址替换成断点指令。当目标程序指向到对应的 uprobe 地址时，会触发断点，从而触发到 uprobe 的中断处理流程 (arch_uprobe_exception_notify)，进而在内核中打印对应的信息。

与 kprobe 类似，我们可以在触发 uprobe 时候根据对应寄存器去提取当前执行的上下文信息，比如函数的调用参数等。同时 uprobe 也有类似的同族: **uretprobe**。使用 uprobe 的好处是我们可以获取许多对于内核态比较抽象的信息，比如 bash 中 readline 函数的返回、SSL_read/write 的明文信息等。

拓展阅读:

- Linux uprobe: User-Level Dynamic Tracing[12]
- Documentation/trace/uprobetracer.rst[13]
- Linux tracing - kprobe, uprobe and tracepoint[14]

tracepoints

tracepont 是内核中提供的一种轻量级代码监控方案，可以实现动态调用用户提供的监控函数，但需要子系统的维护者根据需要自行添加到自己的代码中。

使用

tracepoint 的使用和 uprobe 类似，主要基于 debugfs/tracefs 的文件读写去进行实现。一个区别在于 uprobe 使用的的用户自己定义的观察点(event)，而 tracepoint 使用的是内核代码中预置的观察点。

查看内核(或者驱动)中定义的所有观察点:

```
$ cat /sys/kernel/debug/tracing/available_events
sctp:sctp_probe
sctp:sctp_probe_path
sde:sde_perf_uidle_status
```

```
....
random:random_read
random:urandom_read
...
```

在 events 对应目录下包含了以子系统结构组织的观察点目录:

```
$ ls /sys/kernel/debug/tracing/events/random/
add_device_randomness  credit_entropy_bits  extract_entropy       get_ran
add_disk_randomness    debit_entropy        extract_entropy_user  get_ran
add_input_randomness   enable               filter                mix_poo

$ ls /sys/kernel/debug/tracing/events/random/random_read/
enable  filter  format  id  trigger
```

以 urandom 为例，这是内核的伪随机数生成函数，对其开启追踪:

```
$ echo 1 > /sys/kernel/debug/tracing/events/random/urandom_read/enable
$ echo 1 > /sys/kernel/debug/tracing/tracing_on
$ head -c1 /dev/urandom
$ cat /sys/kernel/debug/tracing/trace_pipe
            head-9949  [006] .... 101453.641087: urandom_read: got_bits 4(
```

其中 trace_pipe 是输出的管道，以阻塞的方式进行读取，因此需要先开始读取再获取 `/dev/urandom`，然后就可以看到类似上面的输出。这里输出的格式是在内核中定义的，我们下面会看到。

当然，最后记得把 trace 关闭。

原理

根据内核文档介绍，子系统的维护者如果想在他们的内核函数中增加跟踪点，需要执行两步操作:

1. 定义跟踪点
2. 使用跟踪点

内核为跟踪点的定义提供了 `TRACE_EVENT` 宏。还是以 urandom_read 这个跟踪点为例，其在内核中的定义在 **include/trace/events/random.h**:

```
#undef TRACE_SYSTEM
#define TRACE_SYSTEM random

TRACE_EVENT(random_read,
    TP_PROTO(int got_bits, int need_bits, int pool_left, int input_left),

    TP_ARGS(got_bits, need_bits, pool_left, input_left),


    TP_STRUCT__entry(
        __field(      int,  got_bits        )
        __field(      int,  need_bits       )
        __field(      int,  pool_left       )
        __field(      int,  input_left      )
    ),

    TP_fast_assign(
        __entry->got_bits   = got_bits;
        __entry->need_bits  = need_bits;
        __entry->pool_left  = pool_left;
        __entry->input_left = input_left;
    ),

    TP_printk("got_bits %d still_needed_bits %d "
          "blocking_pool_entropy_left %d input_entropy_left %d",
          __entry->got_bits, __entry->got_bits, __entry->pool_left,
          __entry->input_left)
);
```

其中:

- **random_read**: trace 事件的名称，不一定要内核函数名称一致，但通常为了易于识别会和某个关键的内核函数相关联。隶属于 **random** 子系统(由 `TRACE_SYSTEM` 宏定义);
- **TP_PROTO**: 定义了跟踪点的原型，可以理解为入参类型;
- **TP_ARGS**: 定义了"函数"的调用参数;
- **TP_STRUCT__entry**: 用于 fast binary tracing，可以理解为一个本地 C 结构体的定义;
- **TP_fast_assign**: 上述本地 C 结构体的初始化;
- **TP_printk**: 类似于 printk 的结构化输出定义，上节中 trace_pipe 的输出结果就是这里定义的;

TRACE_EVENT 宏并不会自动插入对应函数，而是通过展开定义了一个名为 trace_urandom_read 的函数，需要内核开发者自行在代码中进行调用。上述跟踪点实际上是在 **drivers/char/random.c** 文件中进行了调用:

```
static ssize_t
urandom_read_nowarn(struct file *file, char __user *buf, size_t nbytes,
            loff_t *ppos)
{
    int ret;

    nbytes = min_t(size_t, nbytes, INT_MAX >> (ENTROPY_SHIFT + 3));

    ret = extract_crng_user(buf, nbytes);
    trace_urandom_read(8 * nbytes, 0, ENTROPY_BITS(&input_pool)); // <-- 
    return ret;
}

static ssize_t
urandom_read(struct file *file, char __user *buf, size_t nbytes, loff_t *p
{
    unsigned long flags;
    static int maxwarn = 10;

    if (!crng_ready() && maxwarn > 0) {
        maxwarn--;
        if (__ratelimit(&urandom_warning))
            pr_notice("%s: uninitialized urandom read (%zd bytes read)\n",
                  current->comm, nbytes);
        spin_lock_irqsave(&primary_crng.lock, flags);
        crng_init_cnt = 0;
        spin_unlock_irqrestore(&primary_crng.lock, flags);
    }

    return urandom_read_nowarn(file, buf, nbytes, ppos);
}
```

值得注意的是实际上是在 urandom_read_nowarn 函数中而不是 urandom_read 函数中调用的，因此也可见注入点名称和实际被调用的内核函数名称没有直接关系，只需要便于识别和定位即可。

根据上面的介绍我们可以了解到，tracepoint 相对于 probe 来说各有利弊：

- 缺点是需要开发者自己定义并且加入到内核代码中，对代码略有侵入性；
- 优点是对于参数格式有明确定义，并且在不同内核版本中相对稳定，kprobe 跟踪的内核函数可能在下个版本就被改名或者优化掉了；

另外，tracepoint 除了在内核代码中直接定义，还可以在驱动中进行动态添加，用于方便驱动开发者进行动态调试，复用已有的 debugfs 最终架构。这里有一个简单的自定义 tracepoint 示例[15]，可用于加深对 tracepoint 使用的理解。

拓展阅读:

- LWN: Using the TRACE_EVENT() macro (Part 1)[16]
- Documentation/trace/tracepoints.rst[17]
- Taming Tracepoints in the Linux Kernel[18]

USDT

USDT 表示 **Userland Statically Defined Tracing**，即用户静态定义追踪 (币圈同志先退下)。最早源于 Sun 的 Dtrace 工具，因此 USDT probe 也常被称为 Dtrace probe。可以理解为 kernel tracepoint 的用户层版本，由应用开发者在自己的程序中关键函数加入自定义的跟踪点，有点类似于 printf 调试法(误)。

下面是一个简单的示例:

```
#include "sys/sdt.h"
int main() {
  DTRACE_PROBE("hello_usdt", "enter");
  int reval = 0;
  DTRACE_PROBE1("hello_usdt", "exit", reval);
}
```

`DTRACE_PROBEn` 是 UDST (systemtap) 提供的追踪点定义+插入辅助宏，n 表示参数个数。编译上述代码后就可以看到被注入的 USDT probe 信息:

```
$ apt-get install systemtap-sdt-dev
$ gcc hello-usdt.c -o hello-usdt
$ readelf  -n ./hello-usdt
...
Displaying notes found in: .note.stapsdt
  Owner                Data size        Description
  stapsdt              0x0000002e       NT_STAPSDT (SystemTap probe descr
    Provider: "hello_usdt"
    Name: "enter"
```

```
    Location: 0x0000000000001131, Base: 0x0000000000002004, Semaphore: 0x(
    Arguments:
  stapsdt              0x00000038       NT_STAPSDT (SystemTap probe descr
    Provider: "hello_usdt"
    Name: "exit"
    Location: 0x0000000000001139, Base: 0x0000000000002004, Semaphore: 0x(
    Arguments: -4@-4(%rbp)
```

`readelf -n` 表示输出 ELF 中 NOTE 段的信息。

在使用 trace 工具(如 BCC、SystemTap、dtrace) 对该应用进行追踪时，会在启动过程中修改目标进程的对应地址，将其替换为 probe ，在触发调用时候产生对应事件，供数据收集端使用。通常添加 probe 的方式是 **基于 uprobe** 实现的。

使用 USDT 的一个好处是应用开发者可以在自己的程序中定义更加上层的追踪点，方便对于功能级别监控和分析，比如 node.js server 就自带了 USDT probe 点可用于追踪 HTTP 请求，并输出请求的路径等信息。由于 USDT 需要开发者配合使用，不符合我们最初的逆向分析需要，因此就不过多介绍了。(其实是懒得搭环境)

拓展阅读:

- Exploring USDT Probes on Linux[19]
- LWN: Using user-space tracepoints with BPF[20]

小结

上述介绍的四种常见内核监控方案，根据静态/动态类型以及面向内核还是用户应用来划分的话，可以用下表进行概况:

监控方案	静态	动态	内核	用户
Kprobes		✔	✔	
Uprobes		✔		✔
Tracepoints	✔		✔	
USDT	✔			✔

准确来说 USDT 不算是一种独立的内核监控数据源，因为其实现还是依赖于 uprobe，不过为了对称还是放在这里，而且这样目录比较好看。

采集 & 前端

上面我们介绍了几种当今内核中主要的监控数据来源，基本上可以涵盖所有的监控需求。不过从易用性上来看，只是实现了基本的架构，使用上有的是基于内核提供的系统调用/驱动接口，有的是基于 debugfs/tracefs，对用户而言不太友好，因此就有了许多封装再封装的监控前端，本节对这些主要的工具进行简要介绍。

ftrace

ftrace 是内核中用于实现内部追踪的一套框架，这么说有点抽象，但实际上我们前面已经用过了，就是 tracefs 中的使用的方法。

> 在旧版本中内核中(4.1 之前)使用 debugfs，一般挂载到 /sys/kernel/debug/tracing；在新版本中使用独立的 tracefs，挂载到 /sys/kernel/tracing。但出于兼容性原因，原来的路径仍然保留，所以我们将其统一称为 tracefs。

ftrace 通常被叫做 **function tracer**，但除了函数跟踪，还支持许多其他事件信息的追踪:

- hwlat: 硬件延时追踪
- irqsoff: 中断延时追踪
- preemptoff: 追踪指定时间片内的 CPU 抢占事件
- wakeup: 追踪最高优先级的任务唤醒的延时
- branch: 追踪内核中的 likely/unlikely 调用
- mmiotrace: 追踪某个二进制模块所有对硬件的读写事件
- ……

Android 中提供了一个简略的文档指导如何为内核增加 ftrace 支持，详见: Using ftrace[21]。

perf

perf[22] 是 Linux 发行版中提供的一个性能监控程序，基于内核提供的 perf_event_open[23] 系统调用来对进程进行采样并获取信息。Linux 中的 perf 子系统可以

实现对 CPU 指令进行追踪和计数，以及收集 kprobe、uprobe 和 tracepoints 的信息，实现对系统性能的分析。

在 Android 中提供了一个简单版的 perf 程序 simpleperf[24]，接口和 perf 类似。

虽然可以监测到系统调用，但缺点是无法获取系统调用的参数，更不可以动态地修改内核。因此对于安全测试而言作用不大，更多是给 APP 开发者和手机厂商用于性能热点分析。值得一提的是，perf 子系统曾经出过不少漏洞，在 Android 内核提权史中也曾经留下过一点足迹 :D

eBPF

eBPF 为 **extended Berkeley Packet Filter** 的缩写，BPF 最早是用于包过滤的精简虚拟机，拥有自己的一套指令集，我们常用的 `tcpdump` 工具内部就会将输入的过滤规则转换为 BPF 指令，比如:

```
$ tcpdump -i lo0 'src 1.2.3.4' -d
(000) ld       [0]
(001) jeq      #0x2000000       jt 2    jf 5
(002) ld       [16]
(003) jeq      #0x1020304       jt 4    jf 5
(004) ret      #262144
(005) ret      #0
```

该汇编指令表示令过滤器只接受 IP 包，并且来源 IP 地址为 1.2.3.4。其中的指令集可以参考 Linux Socket Filtering aka Berkeley Packet Filter (BPF)[25]。eBPF 在 BPF 指令集上做了许多增强(extend):

- 寄存器个数从 2 个增加为 10 个 (R0 - R9)；
- 寄存器大小从 32 位增加为 64 位；
- 条件指令 jt/jf 的目标替换为 jt/fall-through，简单来说就是 else 分支可以默认忽略；
- 增加了 bpf_call 指令以及对应的调用约定，减少内核调用的开销；
- ……

内核存在一个 eBPF 解释器，同时也支持实时编译(JIT)增加其执行速度，但很重要的一个限制是 eBPF 程序不能影响内核正常运行，在 内核加载 eBPF 程序前会对其进行一次语义检查，确保代码的安全性，主要限制为:

- 不能包含循环，这是为了防止 eBPF 程序过度消耗系统资源(5.3 中增加了部分循环支持)；

- 不能反向跳转，其实也就是不能包含循环；
- BPF 程序的栈大小限制为 512 字节；
- ……

具体的限制策略都在内核的 **eBPF verifier** 中，不同版本略有差异。值得一提的是，最近几年 Linux 内核出过很多 eBPF 的漏洞，大多是 verifier 的验证逻辑错误，其中不少还上了 Pwn2Own[26]，但是由于权限的限制在 Android 中普通应用无法执行 `bpf(2)` 系统调用，因此并不受影响。

eBPF 和 perf_event 类似，通过内核虚拟机的方式实现监控代码过滤的动态插拔，这在许多场景下十分奏效。对于普通用户而言，基本上不会直接编写 eBPF 的指令去进行监控，虽然内核提供了一些宏来辅助 eBPF 程序的编写，但实际上更多的是使用上层的封装框架去调用，其中最著名的一个就是 BCC。

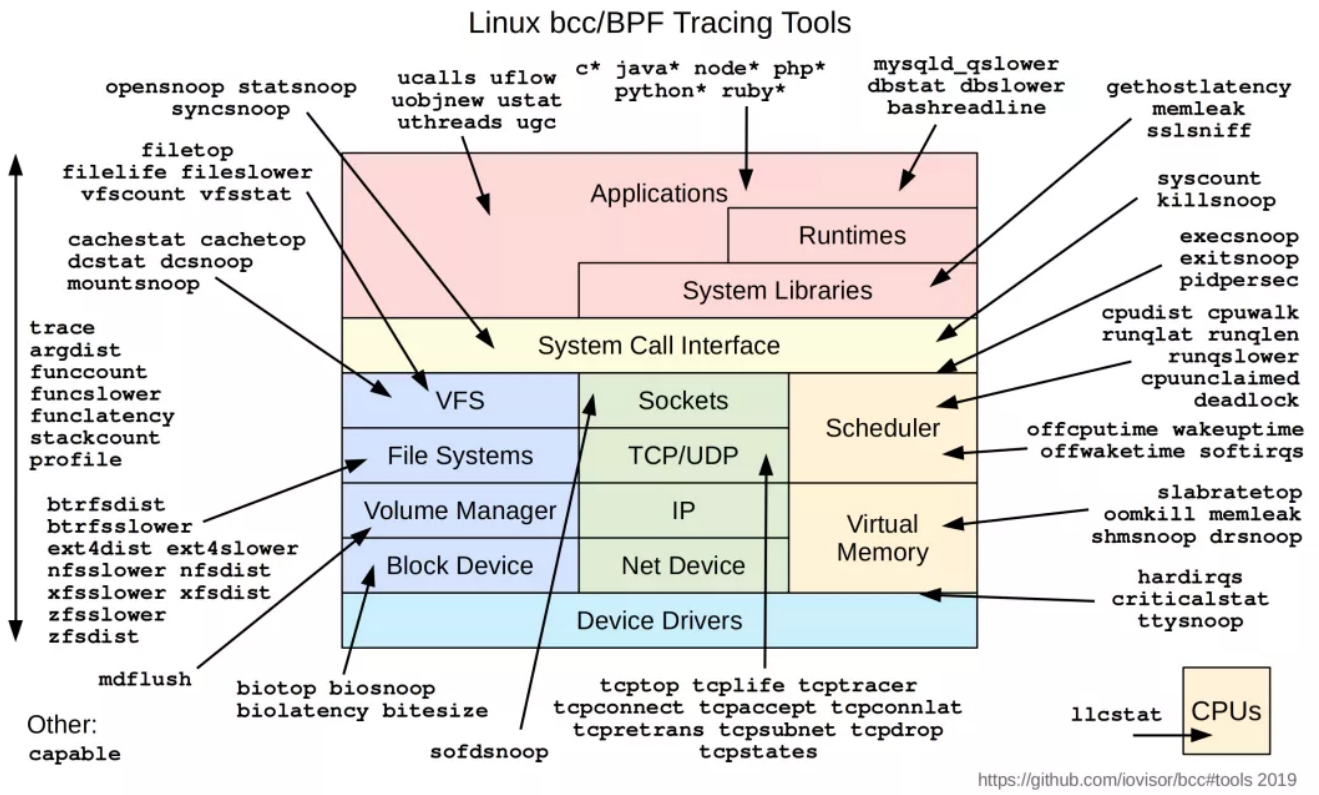
BCC

BCC (BPF Compiler Collection)[27] 包含了一系列工具来协助运维人员编写监控代码，其中使用较多的是其 Python 绑定。一个简单的示例程序如下:

```
from bcc import BPF
prog="""
int kprobe__sys_clone(void *ctx) {
    bpf_trace_printk("Hello, World!\\n");
    return 0;
}
"""
BPF(text=prog).trace_print()
```

执行该 python 代码后，每当系统中的进程调用 clone 系统调用，该程序就会打印 "Hello World" 输出信息。可以看到这对于动态监控代码非常有用，比如我们可以通过 python 传入参数指定打印感兴趣的系统调用及其参数，而无需频繁修改代码。

eBPF 可以获取到内核中几乎所有的监控数据源，包括 kprobes、uprobes、tracepoints 等等，官方 repo 中给出了许多示例程序，比如 opensnoop 监控文件打开行为、execsnoop 监控程序的执行。后文我们会在 Android 系统进行实际演示来感受其威力。



img-bcc

bpftrace

bpftrace[28] 是 eBPF 框架的另一个上层封装，与 BCC 不同的是 bpftrace 定义了一套自己的 DSL 脚本语言，语法(也)类似于 awk，从而可以方便用户直接通过命令行实现丰富的功能，截取几条官方给出的示例:

```
# 监控系统所有的打开文件调用(open/openat)，并打印打开文件的进程以及被打开的文件路径
bpftrace -e 'tracepoint:syscalls:sys_enter_open { printf("%s %s\n", comm,

# 统计系统中每个进程执行的系统调用总数
bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'
```

官方同样也给出了许多 **.bt** 脚本示例，可以通过其代码进行学习和编写。

拓展阅读:

- LWN: A thorough introduction to eBPF[29]
- Extending the Kernel with eBPF[30]

- https://www.opersys.com/downloads/cc-slides/android-debug/slides-main-211122.html#/
- http://www.caveman.work/2019/01/29/eBPF-on-Android/

SystemTap

SystemTap(stab)[31] 是 Linux 中的一个命令行工具，可以对各种内核监控源信息进行结构化输出。同时也实现了自己的一套 DSL 脚本，语法类似于 awk，可实现系统监控命令的快速编程。

使用 systemtap 需要包含内核源代码，因为需要动态编译和加载内核模块。在 Android 中还没有官方的支持，不过有一些开源的 systemtap 移植[32]。

拓展阅读: Comparing SystemTap and bpftrace[33]

其他

除了上面介绍的这些，还有许多开源的内核监控前端，比如 LTTng、trace-cmd[34]、kernelshark[35]等，内核监控输出以结构化的方式进行保存、处理和可视化，对于大量数据而言是非常实用的。限于篇幅不再对这些工具进行一一介绍，而且笔者使用的也不多，后续有机会再进行研究。

Android 移植

上面说了那么多，终究只是 Linux 发行版上的热闹，那么这些 trace 方法在 Android 上行得通吗？理论上 AOSP 的代码是开源的，内核也是开源的，编译一下不就好了。但实践起来我们会遇到几个方面的困难:

1. 许多工具需要编译代码，BCC 工具还需要 Python 运行，这在默认的 Android 环境中不存在；
2. 原厂提供的预编译内核镜像不带有 kprobe 等监控功能支持，需要自行修改配置，烧写和编译内核；
3. Linux 旧版本对于 eBPF 的支持不完善，许多新功能都是在 5.x 后才引进，而 Android 的 Linux 内核都比较旧，需要进行 **cherry-pick** 甚至手动 backport；
4. AOSP 较新版本引入了 GKI(Generic Kernel Image)[36]，需要保持内核驱动接口的兼容性，因此内核代码不能引入过多修改；
5. ……

由于我们主要目的是进行安卓应用逆向分析，因此最好在真机环境运行，因为许多应用并不支持 x86 环境。当然 ARM 模拟器也可以，但在攻防对抗的时可能需要进行额外的模拟器检测绕过。

笔者使用的是 **Google Pixel 5**，使用其他手机的话需要适当进行调整。

1. Debian over Android

Android 系统本身并不是为了开发而设计的，因此只内置了简单的 busybox(toybox) 工具，以及一些包管理相关的程序如 pm/am/dumpsys/input 等。为了在上面构建完整的开发环境，我们需要能在安卓中运行 gcc/clang、python、Makefile 等，一个直观的想法是通过沙盒等方式在上面运行一个常见的 Linux 发行版，比如 Ubuntu 或者 Debian。

androdeb[37] 正是这个想法的一个实现，其核心是基于 **chroot** 在 Android 中运行了一个 Debian aarch64 镜像，并可以通过 apt 等包管理工具安装所需要的编译工具链，从而在上面编译和运行 bcc 等 Linux 项目。

在 Android 上运行 Debian 系统的示例如下:

img-vm

其中的关键之处在于正确挂载原生 Android 中的映射，比如 procfs、devfs、debugfs 等。

2. 自定义内核

解决了在 Android 上运行开发工具的问题之后，我们还需要一个支持动态调试的内核环境。在绝大多数官方固件中自带的内核都没有开启 KPROBES 的支持，这意味着我们自行编译和加载内核。

为了能够支持 KPROBES、UPROBES、TRACEPOINTS 等功能，需要在内核的配置中添加以下选项:

禁用内核的安全特性，开启调试支持:

```
  -d CONFIG_LTO \
  -d CONFIG_LTO_CLANG \
  -d CONFIG_CFI_CLANG \
  -d CFI_PERMISSIVE \
  -d CFI_CLANG \
  -e CONFIG_IRQSOFF_TRACER \
  -e CONFIG_PREEMPT_TRACER \
  -e CONFIG_DEBUG_FS \
  -e CONFIG_CHECKPOINT_RESTORE \
  -d CONFIG_RANDOMIZE_BASE \
```

开启 eBPF 支持:

```
  -e CONFIG_BPF \
  -e CONFIG_BPF_SYSCALL \
  -e CONFIG_BPF_JIT \
  -e CONFIG_HAVE_EBPF_JIT \
  -e CONFIG_IKHEADERS \
```

开启 kprobes 支持:

```
  -e CONFIG_HAVE_KPROBES \
  -e CONFIG_KPROBES \
  -e CONFIG_KPROBE_EVENT \
```

开启 kretprobe 支持:

```
  -e CONFIG_KRETPROBES \
  -e CONFIG_HAVE_KRETPROBES \
  -d CONFIG_SHADOW_CALL_STACK \
  -e CONFIG_ROP_PROTECTION_NONE \
```

开启 ftrace 支持:

```
  -e CONFIG_FTRACE_SYSCALLS \
  -e CONFIG_FUNCTION_TRACER \
```

```
  -e CONFIG_HAVE_DYNAMIC_FTRACE \
  -e CONFIG_DYNAMIC_FTRACE \
```

开启 uprobes 支持:

```
  -e CONFIG_UPROBES \
  -e CONFIG_UPROBE_EVENT \
  -e CONFIG_BPF_EVENTS \
```

BCC 建议设置的选项:

```
  -e CONFIG_DEBUG_PREEMPT \
  -e CONFIG_PREEMPTIRQ_EVENTS \
  -d CONFIG_PROVE_LOCKING \
  -d CONFIG_LOCKDEP
```

为了避免各类环境问题，我建议编译环境最好选择干净的虚拟机英文环境，或者直接使用 Docker 镜像，根据官方的指导去编译，见: Building Kernels[38]。

编译内核常见的依赖:

```
  $ pkg --add-architecture i386
  $ apt install git ccache automake flex lzop bison \
  gperf build-essential zip curl zlib1g-dev zlib1g-dev:i386 \
  g++-multilib python-networkx libxml2-utils bzip2 libbz2-dev \
  libbz2-1.0 libghc-bzlib-dev squashfs-tools pngcrush \
  schedtool dpkg-dev liblz4-tool make optipng maven libssl-dev \
  pwgen libswitch-perl policycoreutils minicom libxml-sax-base-perl \
  libxml-simple-perl bc libc6-dev-i386 lib32ncurses5-dev \
  x11proto-core-dev libx11-dev lib32z-dev libgl1-mesa-dev xsltproc unzip
```

拓展阅读:

- Building a Pixel kernel with KASAN+KCOV[39]
- eBPF/BCC - A better low-level Instrumentation tool on Android[40]

3. 内核移植

当你成功编译好内核并启动后，很可能会发现有一些内核分析工具比如 BCC 在使用上会出现各种问题，这通常是内核版本的原因。由于 eBPF 目前在内核中也在频繁更新，因此许多新的特性并没有增加到当前内核上。

例如，在 Pixel 5 最新的支持的内核是 4.19 版本，在这个版本中，bpf_probe_read_user (issue#3175)[41] 函数还没添加进内核，因此使用 BCC 会回退到 `bpf_probe_read_kernel`，这在内核直接读取用户空间的数据(比如系统调用的参数)时会出现错误，因此我们需要手动去 cherry-pick 对应的 commit，即在 Linux 5.5 中添加的 6ae08ae3dea2[42]。

BCC 所需的所有内核特性及其引进的版本列表可以参考: BCC/kernel-versions.md[43]，部分列表如下所示:

Feature	Kernel version	Commit
`AF_PACKET` (libpcap/tcpdump, `cls_bpf` classifier, netfilter's `xt_bpf`, team driver's load-balancing mode...)	3.15	`bd4cf0ed331a`
Kernel helpers	3.15	`bd4cf0ed331a`
`bpf()` syscall	3.18	`99c55f7d47c0`
Tables (*a.k.a.* Maps; details below)	3.18	`99c55f7d47c0`
BPF attached to sockets	3.19	`89aa075832b0`
BPF attached to `kprobes`	4.1	`2541517c32be`
`cls_bpf` / `act_bpf` for `tc`	4.1	`e2e9b6541dd4`
Tail calls	4.2	`04fd61ab36ec`
Non-root programs on sockets	4.4	`1be7f75d1668`
Persistent maps and programs (virtual FS)	4.4	`b2197755b263`
`tc` 's `direct-action` (`da`) mode	4.4	`045efa82ff56`
`tc` 's `clsact` qdisc	4.5	`1f211a1b929c`
BPF attached to tracepoints	4.7	`98b5c2c65c29`
Direct packet access	4.7	`969bf05eb3ce`
XDP (see below)	4.8	`6a773a15a1e8`
BPF attached to perf events	4.9	`0515e5999a46`
Hardware offload for `tc` 's `cls_bpf`	4.9	`332ae8e2f6ec`

img-commit

因此为了减少可能遇到的兼容性问题，尽量使用最新版本的内核，当然通常厂商都只维护一个较旧的 LTS 版本，只进行必要的安全性更新，如果买机不淑的话就需要自食其力了。

实战测试

通过在上述 Android Debian 环境编译好 BCC 之后，我们就可以使用 Python 编写对应的应用跟踪分析脚本了。一般是通过应用名去过滤系统调用，但是在 Android 中还有个特别的过滤方式就是通过用户 ID，因为应用是根据动态安装获取的 UID 去进行沙盒隔离的。

以某个层层加固的恶意 APK 为例，安装后获取其 UID 为 `u0_a142`，转换成数字是 `10142`，对其进行 exec 系统调用的监控：

img-exec

可以看到目标应用调用了 ps、getprop、pm 等程序，用来检测当前系统的 adb 状态以及所安装的应用，比如其中通过 `pm path com.topjohnwu.magisk` 来判断 Magisk 工具是否存在，因此存在 root 检测行为。上图中 `pm` 实际调用了 `cmd` 程序进行查找，因为 pm 本质上只是一个 shell 脚本：

```
$ cat `which pm`
#!/system/bin/sh
cmd package "$@"
```

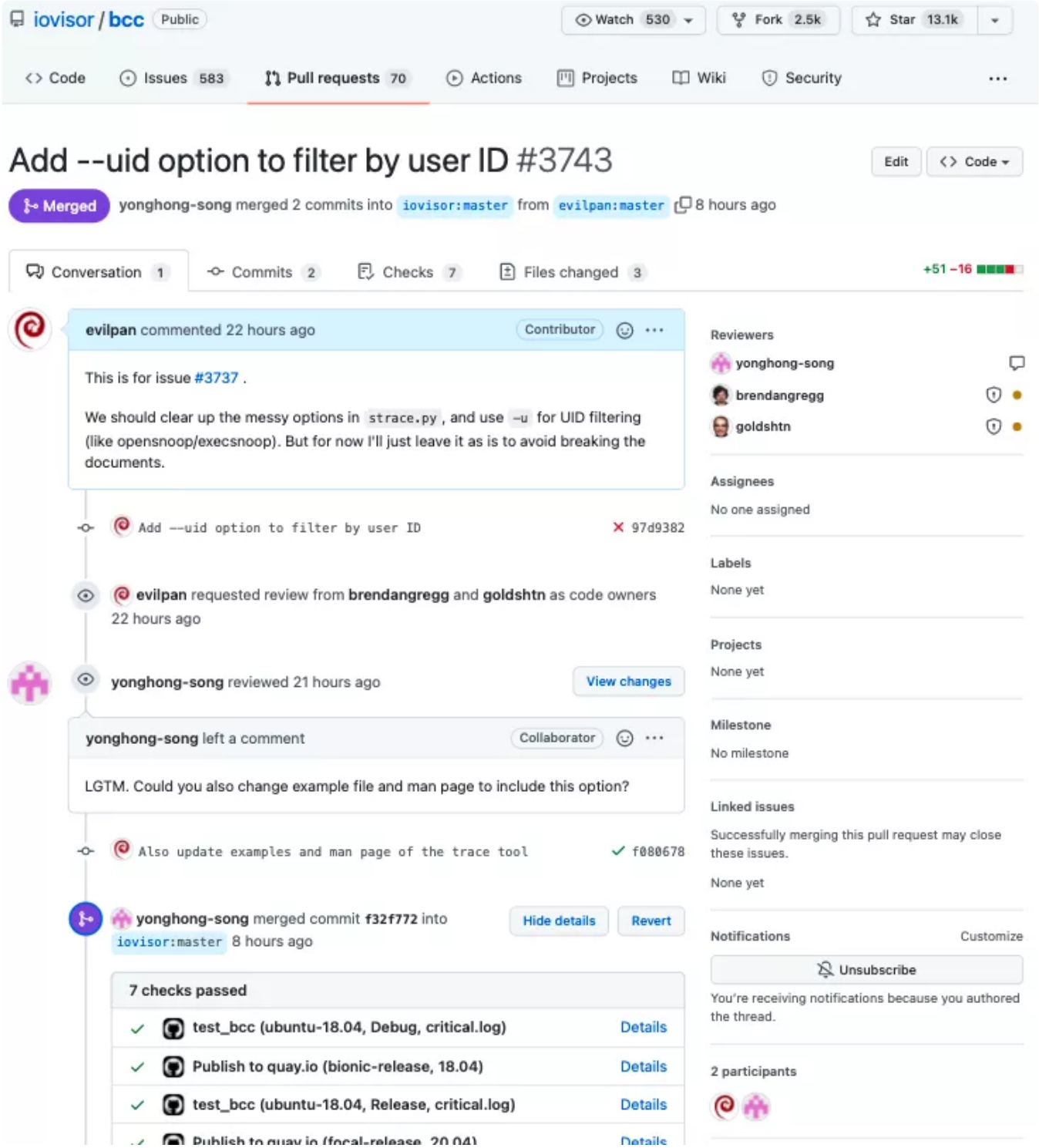
使用 UID 进行过滤的好处是可以跟踪所有 fork 的子进程和孙子进程，这是基于 PID 或者进程名跟踪所无法比拟的。除了 exec，我们还可以跟踪其他内核函数，比如 root 检测经常用到的 openat 或 access，如下所示：

```
root at evilpan in ~
$ ./trace.py 'do_faccessat "%s", arg2' --uid 10142 -f /su
PID     TID     COMM            FUNC             -
16114   16181   DaemonThread-6  do_faccessat     /product/bin/su
16114   16181   DaemonThread-6  do_faccessat     /apex/com.android.runtime/bin/su
16114   16181   DaemonThread-6  do_faccessat     /apex/com.android.art/bin/su
16114   16181   DaemonThread-6  do_faccessat     /system_ext/bin/su
16114   16181   DaemonThread-6  do_faccessat     /system/bin/su
16114   16181   DaemonThread-6  do_faccessat     /system/xbin/su
16114   16181   DaemonThread-6  do_faccessat     /odm/bin/su
16114   16181   DaemonThread-6  do_faccessat     /vendor/bin/su
16114   16181   DaemonThread-6  do_faccessat     /vendor/xbin/su
16114   16182   preload_sg      do_faccessat     /product/bin/su
16114   16182   preload_sg      do_faccessat     /apex/com.android.runtime/bin/su
16114   16182   preload_sg      do_faccessat     /apex/com.android.art/bin/su
16114   16182   preload_sg      do_faccessat     /system_ext/bin/su
16114   16182   preload_sg      do_faccessat     /system/bin/su
16114   16182   preload_sg      do_faccessat     /system/xbin/su
16114   16182   preload_sg      do_faccessat     /odm/bin/su
16114   16182   preload_sg      do_faccessat     /vendor/bin/su
16114   16182   preload_sg      do_faccessat     /vendor/xbin/su
16211   16666   DaemonThread-6  do_faccessat     /product/bin/su
16211   16666   DaemonThread-6  do_faccessat     /apex/com.android.runtime/bin/su
16211   16666   DaemonThread-6  do_faccessat     /apex/com.android.art/bin/su
16211   16666   DaemonThread-6  do_faccessat     /system_ext/bin/su
```

img-su

基于内核级别的监控，让应用中所有的加固/隐藏/内联汇编等防御措施形同虚设，而且可以在应用启动的初期进行观察，让应用的一切行为在我们眼中无所遁形。

PS: 如果在使用 BCC 的过程中发现没有过滤 UID 的选项，那可能需要切换到最新的 release 版本或者 master 分支，因为这个选项是笔者最近才加上去的。



img-pr

:D

拓展阅读:

- eBPF super powers on ARM64 and Android (slides)[44]
- eBPF - Android Reverse Engineering Superpowers[45]

总结

本文总结并分析了几种内核主要的监控方案，它们通常用于性能监控和内核调试，但我们也可以将其用做安全分析，并在 Android 中进行了实际的移植和攻防测试，并且获得了超出预期的实战效果。除了内核级别监控，我们还可以基于 uprobes 实现应用内任意地址的监控，如在 SSL_read/write 地址处获取所有 SSL 加密的数据。得益于内核提供的丰富监控原语，我们可以实现内核级别移动端沙盒，全面监控移动应用行为，也可以通过内核读写原语去实现系统调用参数修改，从而实现应用运行环境的模拟和伪造。

引用链接

[1] 某不知名安全研究员: *https://evilpan.com/about/#2018-06-30*

[2] strace: *https://man7.org/linux/man-pages/man1/strace.1.html*

[3] How does strace work?: *https://blog.packagecloud.io/eng/2016/02/29/how-does-strace-work/*

[4] frida: *https://github.com/frida/frida*

[5] QDBI: *https://qbdi.quarkslab.com/*

[6] hookzz: *https://github.com/jmpews/Dobby*

[7] Linux tracing systems & how they fit together: *https://jvns.ca/blog/2017/07/05/linux-tracing-systems/*

[8] An introduction to KProbes: *https://lwn.net/Articles/132196/*

[9] Documentation/trace/kprobetrace.rst: *https://www.kernel.org/doc/html/latest/trace/kprobes.html*

[10] samples/kprobes/kprobe_example.c: *https://elixir.bootlin.com/linux/latest/source/samples/kprobes/kprobe_example.c*

[11] samples/kprobes/kretprobe_example.c: *https://elixir.bootlin.com/linux/latest/source/samples/kprobes/kretprobe_example.c*

[12] Linux uprobe: User-Level Dynamic Tracing: *https://www.brendangregg.com/blog/2015-06-28/linux-ftrace-uprobe.html*

[13] Documentation/trace/uprobetracer.rst: *https://www.kernel.org/doc/html/latest/trace/uprobetracer.html*

[14] Linux tracing - kprobe, uprobe and tracepoint: *https://terenceli.github.io/%E6%8A%80%E6%9C%AF/2020/08/05/tracing-basic*

[15] 自定义 tracepoint 示例: *https://lwn.net/Articles/383362/*

[16] LWN: Using the TRACE_EVENT() macro (Part 1): *https://lwn.net/Articles/379903/*

[17] Documentation/trace/tracepoints.rst: *https://www.kernel.org/doc/html/latest/trace/tracepoints.html*

[18] Taming Tracepoints in the Linux Kernel: *https://blogs.oracle.com/linux/post/taming-tracepoints-in-the-linux-kernel*

[19] Exploring USDT Probes on Linux: *https://leezhenghui.github.io/linux/2019/03/05/exploring-usdt-on-linux.html*

[20] LWN: Using user-space tracepoints with BPF: *https://lwn.net/Articles/753601/*

[21] Using ftrace: *https://source.android.google.cn/devices/tech/debug/ftrace*

[22] perf: *https://perf.wiki.kernel.org/index.php/Tutorial*

[23] perf_event_open: *https://man7.org/linux/man-pages/man2/perf_event_open.2.html*

[24] simpleperf: *https://android.googlesource.com/platform/system/extras/+/master/simpleperf/doc*

[25] Linux Socket Filtering aka Berkeley Packet Filter (BPF): *https://www.kernel.org/doc/Documentation/networking/filter.txt*

[26] Pwn2Own: *https://flatt.tech/assets/reports/210401_pwn2own/whitepaper.pdf*

[27] BCC (BPF Compiler Collection): *https://github.com/iovisor/bcc*

[28] bpftrace: *https://github.com/iovisor/bpftrace*

[29] LWN: A thorough introduction to eBPF: *https://lwn.net/Articles/740157/*

[30] Extending the Kernel with eBPF: *https://source.android.com/devices/architecture/kernel/bpf*

[31] SystemTap(stab): *https://sourceware.org/systemtap/wiki*
[32] 开源的 systemtap 移植: *https://github.com/flipreverse/systemtap-android*
[33] Comparing SystemTap and bpftrace: *https://lwn.net/Articles/852112/*
[34] trace-cmd: *https://git.kernel.org/pub/scm/linux/kernel/git/rostedt/trace-cmd.git*
[35] kernelshark: *https://kernelshark.org/Documentation.html*
[36] GKI(Generic Kernel Image): *https://source.android.com/devices/architecture/kernel/generic-kernel-image*
[37] androdeb: *https://github.com/joelagnel/adeb*
[38] Building Kernels: *https://source.android.com/setup/build/building-kernels*
[39] Building a Pixel kernel with KASAN+KCOV: *https://source.android.com/devices/tech/debug/kasan-kcov*
[40] eBPF/BCC - A better low-level Instrumentation tool on Android: *https://blog.senyuuri.info/2021/06/30/ebpf-bcc-android-instrumentation/#_1-choose-kernel-features*
[41] bpf_probe_read_user (issue#3175): *https://github.com/iovisor/bcc/issues/3175*
[42] Linux 5.5 中添加的 6ae08ae3dea2: *https://git.kernel.org/pub/scm/linux/kernel/git/bpf/bpf-next.git/commit/?id=6ae08ae3dea2cfa03dd3665a3c8475c2d429ef47*
[43] BCC/kernel-versions.md: *https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md*
[44] eBPF super powers on ARM64 and Android (slides): *http://www.joelfernandes.org/resources/bcc-ospm.pdf*
[45] eBPF - Android Reverse Engineering Superpowers: *https://www.aisp.sg/cyberfest/document/CRESTConSpeaker/eBPF.pdf*

收录于话题 #Android 9

下一篇 · ART 在 Android 安全攻防中的应用

阅读原文