基于 HOOK 的 Anti-debug 调用点 trace 和 Anti-anti

一、概述

相信动态调试过 SO 的坛友对 Anti 并不陌生，比如读取/proc/self/status，/proc/self/task/xxx/status、stat 文件查看状态 TracePid 和 PPid，读取 wchan 查看进程等待，添加 notify，模拟器检测等等。经过各种掉坑之后，虽然知道了这些检测方法，但如果 elf 文件被畸形不能静态分析(修复又是另外一回事了)，也只能动态调试在这些关键 API 处下断点，也免不了掉坑调试崩溃，再来点指令混淆的话，调试起来确实比较恶心。虽然在 Android 平台同样也存在 linux 常用的 ltrace(库函数 trace)和 strace(系统调用和信号 trace)工具，但不能满足调试需求，还有检测 ltrace 的调试。废话不多说，直接进入正题(限于水平，难免会有疏漏和错误之处，请各位大大斧正，小弟感激不尽)。

二、调用点 trace

注入 Hook API 函数，当然首选 zygote 进程是最方便的了(如果 zygote 不能注入的话，那就关文档去修改 ROM 吧)。HooK 住函数之后，通常采用下面这种模式添加代码：

```
Xxx new_fun(xxx){
    Before_call();
    Old_fun();
    After_call();
}
```

相信许多都知道 ARM 函数的调用流程，这里再啰嗦下 ARM 汇编如何调用函数：

直接调用：BL/BLX _xxfun

函数指针调用：BL/BLX Rx

某些混淆代码：mov lr, [pc, #xx], 计算 Rx，Ldr PC, Rx 等

外部函数调用：BL/BLX _plt 表项，plt 表中 load got 表存在的函数地址到 PC

被调函数一般模式：

```
Stmxx {Rx-Rx, lr}    /push
…
ldmxx {Rx-rx, pc}    /pop
```

函数的返回地址保存在 lr 寄存器中，之后被被调函数存在放栈上。只要 lr 的值未被修改，那么就保存着调用点的下一条指令地址！只要获取 lr 的值，那么就能跟踪到调用点。

知道了 lr 保存了调用点的值，获取 lr 的值也要注意时机。调用函数时 lr 的值已经被存在放栈上，修改 lr 也无关紧要。无法确定被调函数是否存在显式地给 lr 赋值或者函数调用，那要保证 lr 不被修改，故在函数入口点就保存 lr 的值是最佳也是最简单的选择。获取 lr 的值可以通过汇编来实现：

```
#define GETLR(store_lr) \
    __asm__ __volatile__(   \
        "mov %0, lr\n\t"    \
        :     "=r"(store_lr)  \
    )
```

则 Hook 函数模式：

```
Xxx new_fun(xxx){
    Unsigned lr;
```

```
    GETLR(lr)
    Before_call();
    Old_fun();
    After_call();
}
```

来个实例，简单起见只 Hook fopen 函数：

```
FILE* new_fopen(const char *path,const char * mode){
    unsigned lr;
    GETLR(lr);

    if(strstr(path, "status") != NULL){
        LOGD("[*] Traced-fopen Call function: 0x%x\n", lr);
        if(strstr(path, "task") != NULL){
            LOGD("[*] Traced-anti-task/status");
        }else
            LOGD("[*] Traced-anti-status");
    }else if(strstr(path, "wchan") != NULL){
        LOGD("[*] Traced-fopen Call function: 0x%x\n", lr);
        LOGD("[*] Traced-anti-wchan");
    }
    return old_fopen(path, mode);
}
```

注入后，某 APK 输出：

```
LOAD:000012BC    BLX     getpid
LOAD:000012C0    LDR     R1, =(aProcDStatus - 0x12CA)
LOAD:000012C2    MOVS    R2, R0
LOAD:000012C4    ADD     R0, SP, #0x80+s ; s
LOAD:000012C6    ADD     R1, PC  ; "/proc/%d/status"
LOAD:000012C8    BLX     sprintf
LOAD:000012CC    LDR     R1, =(aR - 0x12D4)
LOAD:000012CE    ADD     R0, SP, #0x80+s ; filename
LOAD:000012D0    ADD     R1, PC  ; "r"
LOAD:000012D2    BLX     fopen
LOAD:000012D6    MOVS    R5, R0
LOAD:000012D8    CMP     R0, #0
```

图 1 调用点

Logcat:

```
1950  1950  TK  [*] Traced-fopen Call function: 0x80c012d7
1950  1950  TK  [*] Traced-anti-status
```

图 2 trace

Trace 到了这个检测点，大概就知道这个函数想干什么了。动态调试时，也好准备”跨坑”而不是”掉坑”了。

除了直接获得这些调用点过坑外，还可以组合一些其他功能。比如监控 APK 启动时 mmap 分配内存，直接 dd，便于大致分析某段代码做了什么，配合 mprotect 能起到一定的效果。当然，不必担心会监控到 linker 加载 SO 时 mmap 的无用信息，因为 linker 自身实现了 mmap 函数。

```
static void* new_mmap(void* start,size_t length,int prot,int flags,int fd,off_t offset){
    unsigned lr;
    void* base = NULL;

    GETLR(lr);
    base = old_mmap(start, length, prot, flags, fd, offset);
    if((flags & MAP_ANONYMOUS) == 0){   //文件映射
        char file_name[256];
        char buf[256];
        memset(buf, 0, 256);
        sprintf(file_name, "/proc/self/fd/%d", fd);
        if(readlink(file_name, buf, 256) < 0){
            LOGD("[E] Traced-mmap --> readlink %s error\n", file_name);
            goto _done;
        }
        LOGD("[*] Traced-mmap --> [file] start = %p, length = 0x%x, filename = %s, offset =
0x%x",
            start, length, buf, offset);
    }else{      //内存映射
        LOGD("[*] Traced-mmap --> [mem] start = %p, length = 0x%x",
            start, length);
    }
    LOGD("[*] Traced-mmap Call function: 0x%x, Ret address: 0x%x\n", lr, (unsigned)base);
_done:
    return base;
}
```

```
TK                                  [*] Traced-mmap --> [mem] start = 0x0, length = 0x5000
TK                                  [*] Traced-mmap Call function: 0x80c016a5, Ret address: 0x45728000
```

图 3

至于还能做什么，那就靠各位读者自行研究了吧。附上一些常见 anti trace。

三、Anti-anti

讨论一些常见的基于 Hook 的 Anti-anti 方法，欢迎讨论。

1、 status 和 stat

status 和 stat 的 Anti-anti 方式类似，通过 Hook fopen 实现重定向到/data/local/tmp 目录下：

```
        sprintf(re_path, "/data/local/tmp/status");
        if(!HasGenFile(re_path)){
         char buffer[BUFFERSIZE];
         FILE *fpr, *fpw;
         fpr = old_fopen(path, "r");
         fpw = old_fopen(re_path, "w");
         if(fpr == NULL || fpw == NULL){
            LOGD("[E] re-path [%s]failed", path);
```

```
            return old_fopen(path, mode);
        }
        while(fgets(buffer, BUFFERSIZE, fpr) != NULL){
            if(strstr(buffer, "State") != NULL){
                fputs("State:\tS (sleeping)\n", fpw);
            }
            if(strstr(buffer, "TracerPid") != NULL){
                fputs("TracerPid:\t0\n", fpw);
            }else{
                fputs(buffer, fpw);
            }
        }
        fclose(fpr);
        fclose(fpw);
    }
```

2、 wchan

和 status 类似，只是重定向时，将等待事件设置为 sys_epoll_wait：

```
if(strstr(path, "wchan") != NULL){
        LOGD("[*] Anti-anti-wchan!");
        strcpy(re_path, "data/local/tmp/wchan");
        if(!HasGenFile(re_path)){
            FILE *fpw;
            fpw = old_fopen(re_path, "w");
            if(fpw == NULL){
                LOGE("[E] re-path wchan failed!");
                goto __normal;
            }
            fputs("sys_epoll_wait", fpw);
            fclose(fpw);
        }
     return old_fopen(re_path, mode);
```

3、 inotify_add_watch

检测 mem、pagemem、task 等读取事件。可以直接让其返回-1，但不推荐这么做。如果检测代码并未对返回值做判断，直接使用，这样 Anti-anti 会导致程序崩溃，我的做法是改变 mask 的值：

```
//监控打开和读事件
    if(strstr(pathname, "mem") != NULL){
     LOGD("[*] inotify_add_watch --> patch mem");
     return old_inotify_add_watch(fd, pathname, 0x00000200);       //mem 永远不会被删除，改为 0 也可以
    //监控打开和读事件，防获取反调试线程信息
    }else if(strstr(pathname, "task") != NULL){
```

```
        LOGD("[*] inotify_add_watch --> patch task");
    return old_inotify_add_watch(fd, pathname, 0x00000200);
```

4、 ptrace

这个函数算是用得比较多的吧，简单的检测代码是调用 PTRACE_TRACEME，直接返回 0 就可以，不过现在这种方式很少了吧。某加固 fork 了进程去作 ptrace，并写入一些数据。通过对 ptrace hook，可以监控到写入数据。模拟这个通信过程就可以 Anti-anti。具体就不展开了吧，相信各位读者可以做到了。

当然，还有一些 Anti-anti 方法，限于篇幅，就不展开了吧(让小弟学学各位大牛的 Anti-anti 方法吧)。

四、总结

做好 Anti-trace 和 Anti-anti 有些时候能大大节省时间，将精力专注于算法或者其他逻辑上。但也存在一些问题：

1、 HOOK 检测

导出表 HOOK 检测：读取/system/lib/libc.so 特定函数偏移，再获取本进程 libc.so 的基地址来检测是否有 HOOK。(Patch 方法：Hook fopen、dlopen 和 open 函数)

Inline HooK 检测：Inline Hook 时要替换函数起始的字节码，可以检测一些比较奇怪的二进制码(BX pc 等字节码)。

2、 LOGCAT

许多反调试都会起一个进程或者线程循环监控，此时 anti-trace 的输出会比较多，比如：

Application	Tag	Text
com.crackme	TK	[*] Traced-mmap Call function: 0x80c27280, :
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58
com.crackme	TK	[*] Traced-anti-task/status
com.crackme	TK	[*] Traced-fopen Call function: 0x80c33a58

图 4

这看起来比较烦，而且很多都是相同的。可以利用调用点 trace 地址的唯一性进行过滤，让同一信息只输出一次即可，也可以写个 apk，将信息通过 AF_UNIX 本地套接字发送给 apk，利用数据库的优势来存储等等。

3、 SVC

有些关键性的 API，为了隐藏，直接通过汇编实现系统调用。ARM 系统调用时的调用号通过 r7 寄存器来传递的。如果编写时，没有手写花指令，直接通过扫描 SVC 字节码和附近的关于 r7 的寄存器赋值操作，可以获得一些搜索结果。不过为了隐藏，手写点花指令也是可以的。

比如：常用来刷 cache 的代码

```
static void clearcache(char* begin, char *end)
{
    const int syscall = 0xf0002;
    __asm __volatile (
        "mov        r0, %0\n"
        "mov        r1, %1\n"
        "mov        r7, %2\n"
        "mov        r2, #0x0\n"
        "svc        0x00000000\n"
        :
        :      "r" (begin), "r" (end), "r" (syscall)
        :      "r0", "r1", "r7"
        );
}
```