原文地址:http://drops.wooyun.org/mobile/16969

Author：超六、曲和

0x00 时间相关反调试

通过计算某部分代码的执行时间差来判断是否被调试，在Linux内核下可以通过time、gettimeofday，或者直接通过sys call来获取当前时间。另外，还可以通过自定义SIGALRM信号来判断程序运行是否超时。

0x01 检测关键文件

（1）/proc/pid/status、/proc/pid/task/pid/status

在调试状态下，Linux内核会向某些文件写入一些进程状态的信息，比如向/proc/pid/status或/proc/pid/task/pid/status文件的TracerPid字段写入调试进程的pid，在该文件的statue字段中写入t（tracing stop）：

```
root@generic:/ # cat /proc/1314/status
Name:   ndroid.calendar
State:  t (tracing stop)
Tgid:   1314
Pid:    1314
PPid:   56
TracerPid:      1604
Uid:    10017   10017   10017   10017
Gid:    10017   10017   10017   10017
```


（2）/proc/pid/stat、/proc/pid/task/pid/stat

调试状态下/proc/pid/stat、/proc/pid/task/pid/stat文件中第二个字段是t（T）：

```
root@generic:/data # cat /proc/914/task/914/stat
914 (m.android.music) t 58 58 0 0 -1 4194624 2815 0 3 0 7 13 0 0 20 0 11 0 8169
241414144 4789 4294967295 3069575168 3069580360 3198532416 3198530488 3069187532
 0 4612 0 38136 3221386416 0 0 17 0 0 0 0 0 0 3069586920 3069587456 3081834496
```


（3）/proc/pid/wchan、/proc/pid/task/pid/wchan

若进程被调试，也会往/proc/pid/wchan、/proc/pid/task/pid/wchan文件中写入ptrace_stop。

0x02 检测端口号

使用IDA动态调试APK时，android_server默认监听23946端口，所以通过检测端口号可以起到一定的反调试作用。具体而言，可以通过检测/proc/net/tcp文件，或者直接system执行命令`netstat -apn`等。

0x03 检测android_server、gdb、gdbserver

在对APK进行动态调试时，可能会打开android_server、gdb、gdbserver等调试相关进程，一般情况下，这几个打开的进程名和文件名相同，所以可以通过运行状态下的进程名来检测这些调试相关进程。具体而言，可以通过打开/proc/pid/cmdline、/proc/pid/statue等文件来获取进程名。当然，这种检测方法非常容易绕过——直接修改android_server、gdb、gdbserver的名字即可。

0x04 signal

信号机制在apk调试攻防中有着非常重要的作用，大部分主流加固厂商都会通过信号机制来增加壳的强度。在反调试中最常见的要数SIGTRAP信号了，SIGTRAP原本是调试器设置断点时发出的信号，为了能更好的理解SIGTRAP信号反调试，先让我们看看一下调试器设置断点的原理：

和x86架构类似，arm架构下调试器设置断点先要完成两件事：

1. 保存目标地址上的数据
2. 将目标地址上头几个字节替换成arm/thumb下的breakpoint指令

Arm架构下各类指令集breakpoint机器码如下：

指令集	**Breakpoint机器码（little endian）**
Arm	0x01, 0x00, 0x9f, 0xef
Thumb	0x01, 0xde
Thumb2	0xf0, 0xf7, 0x00, 0xa0

调试器设置完断点之后程序继续运行，直至命中断点，触发breakpoint，这时程序向操作系统发送SIGTRAP信号。调试器收到SIGTRAP信号后，会继续完成以下几件事：

1. 在目标地址上用原来的指令替换之前的breakpoint指令
2. 回退被跟踪进程的当前pc值

当控制权回到原进程时，pc就恰好指向了断点所在位置，这就是调试器设置断点的基本原理。在知道上述原理之后，再让我们继续分析SIGTRAP反调试的细节，如果我们在程序中间插入一条breakpoint指令，而不做其他处理的话，操作系统会用原来的指令替换breakpoint指令，然而这个breakpoint是我们自定义插入的，该地址上并不存在原指令，所以操作系统就跳过这个步骤，进入下一步回退pc值，即breakpoint的前一条指令。这时就出现问题了，下一条指令还是breakpoint指令，这也就造成了无限循环。

为了能继续正常执行，就需要模拟调试器的操作——替换breakpoint指令，而完成这个步骤的最佳时机就是在自定义signal的handle中。Talk is cheap，show me the code，下面给出此原理的简单实例：

```
#!cpp
char dynamic_ccode[] = {0x1f,0xb4, //push {r0-r4}
                        0x01,0xde, //breakpoint
                        0x1f,0xbc, //pop {r0-r4}
                        0xf7,0x46};//mov pc,lr

char *g_addr = 0;

void my_sigtrap(int sig){

    char change_bkp[] = {0x00,0x46}; //mov r0,r0
    memcpy(g_addr+2,change_bkp,2);
    __clear_cache((void*)g_addr,(void*)(g_addr+8)); // need to clear cache
    LOGI("chang bpk to nop\n");

}

void anti4(){//SIGTRAP

    int ret,size;
    char *addr,*tmpaddr;

    signal(SIGTRAP,my_sigtrap);

    addr = (char*)malloc(PAGESIZE*2);

    memset(addr,0,PAGESIZE*2);
    g_addr = (char *)(((int) addr + PAGESIZE-1) & ~(PAGESIZE-1));

    LOGI("addr: %p ,g_addr : %p\n",addr,g_addr);

    ret = mprotect(g_addr,PAGESIZE,PROT_READ|PROT_WRITE|PROT_EXEC);
    if(ret!=0)
    {
        LOGI("mprotect error\n");
        return ;
    }

    size = 8;
    memcpy(g_addr,dynamic_ccode,size);

    __clear_cache((void*)g_addr,(void*)(g_addr+size)); // need to clear cache

    __asm__("push {r0-r4,lr}\n\t"
            "mov r0,pc\n\t"  //此时pc指向后两条指令
            "add r0,r0,#4\n\t"//+4 是的lr 地址为 pop{r0-r5}
            "mov lr,r0\n\t"
            "mov pc,%0\n\t"
            "pop {r0-r5}\n\t"
            "mov lr,r5\n\t" //恢复lr
    :
    :"r"(g_addr)
    :);

    LOGI("hi, i'm here\n");
    free(addr);

}
```

在代码中主动触发breakpoint指令，然后在自定义SIGTRAP handle中将breakpoint替换成nop指令，于是程序可以正常执行完毕。

其中可使用r_debug-r_brk来触发异常，其原理即是用到了linker中一些调试特性。Linker中有一个和调试相关的结构体r_debug，其定义如下：

```
#!cpp
struct r_debug {
    int32_t r_version;
    link_map_t* r_map;
    void (*r_brk)(void);
    int32_t r_state;
    uintptr_t r_ldbase;
};
```

r_debug是以静态变量的形式存在于linker中，其初始化代码如下：

```
#!cpp
static r_debug _r_debug = {1, NULL, &rtld_db_dlactivity, RT_CONSISTENT, 0};
```

在初始化时，r_debug中的r_brk函数指针被初始化成了rtld_db_dlactivity函数，该函数只是一个空的桩函数：

```
#!cpp
/*
 * This function is an empty stub where GDB locates a breakpoint to get notified
 * about linker activity.  It can't be inlined away, can't be hidden.
 */
extern "C" void __attribute__((noinline)) __attribute__((visibility("default"))) rtld_db_dlactivity() {
}
```

没调试下，该函数即为空函数，而在调试状态下会将该函数的内容改写为相应指令集的breakpoint指令。所以先注册自己的signal函数处理breakpoint异常（SIGTRAP），然后在运行时调用该函数，即可触发自定义SIGTRAP的接管函数。而动态调试时，SIGTRAP会先被调试器接收，这样不仅能迷惑调试器，还能在自定义接管函数中做一些tricky的事。

0x05 检测软件断点

上一节说了使用SIGTRAP反调试的原理，由此可以衍生出另一种很常见的反调试方法——检测软件断点。软件断点通过改写目标地址的头几字节为breakpoint指令，只需要遍历so中可执行segment，查找是否出现breakpoint指令即可。实现大致如下：

```
#!cpp
unsigned long GetLibAddr() {
    unsigned long ret = 0;
    char name[] = "libanti_debug.so";
    char buf[4096], *temp;
    int pid;
    FILE *fp;
```

```
    pid = getpid();
    sprintf(buf, "/proc/%d/maps", pid);
    fp = fopen(buf, "r");
    if (fp == NULL) {
        puts("open failed");
        goto _error;
    }
    while (fgets(buf, sizeof(buf), fp)) {
        if (strstr(buf, name)) {
            temp = strtok(buf, "-");
            ret = strtoul(temp, NULL, 16);
            break;
        }
    }
    _error: fclose(fp);
    return ret;
}



void anti5(){

    Elf32_Ehdr *elfhdr;
    Elf32_Phdr *pht;
    unsigned int size, base, offset,phtable;
    int n, i,j;
    char *p;

    //从maps中读取elf文件在内存中的起始地址
    base = GetLibAddr();
    if(base == 0){
        LOGI("find base error\n");
        return;
    }

    elfhdr = (Elf32_Ehdr *) base;

    phtable = elfhdr->e_phoff + base;

    for(i=0;i<elfhdr->e_phnum;i++){

        pht = (Elf32_Phdr*)(phtable+i*sizeof(Elf32_Phdr));

        if(pht->p_flags&1){
            offset = pht->p_vaddr + base + sizeof(Elf32_Ehdr) + sizeof(Elf32_Phdr)*elfhdr->e_phnum;
            LOGI("offset:%X ,len:%X",offset,pht->p_memsz);

            p = (char*)offset;
            size = pht->p_memsz;

            for(j=0,n=0;j<size;++j,++p){

                if(*p == 0x10 && *(p+1) == 0xde){
                    n++;
                    LOGI("### find thumb bpt %X \n",p);
                }else if(*p == 0xf0 && *(p+1) == 0xf7 && *(p+2) == 0x00 && *(p+3) == 0xa0){
                    n++;
                    LOGI("### find thumb2 bpt %X \n",p);
                }else if(*p == 0x01 && *(p+1) == 0x00 && *(p+2) == 0x9f && *(p+3) == 0xef){
                    n++;
                    LOGI("### find arm bpt %X \n",p);
                }

            }
            LOGI("### find breakpoint num: %d\n",n);

        }
    }

}
```

大家在使用IDA调试的时候，也许会注意到IDA的代码窗口和hex view窗口在设置断点的时候，目标地址的内容并没有发生改变，其实这是IDA故意将其隐藏了，设置完断点之后直接用dd dump内存就能看见设置断点的地址头几字节发生了改变。

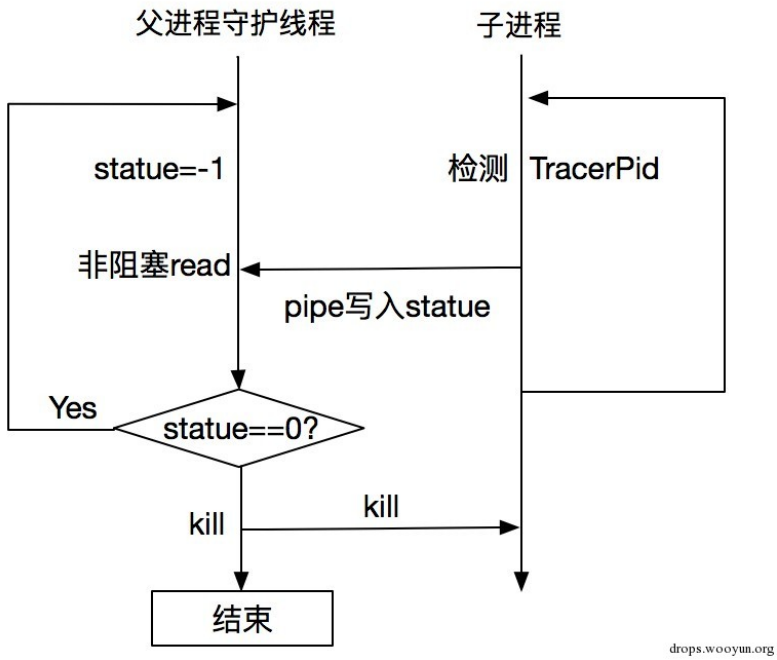
0x06 进程间通信

大部分加固会新建进程或者新建线程，在这些新建的线程和进程中完成反调试操作，然而如果这些进程、线程相对独立的话，很容易通过挂起、杀死的方式直接使得反调试失效。为了保证反调试线程、进程的存活，就需要一种通信方式，定期确认反调试线程、进程依然存活，所以进程间通信是高级反调试不可或缺的方式。在Linux下有很多进程间通信的方式，比如管道、信号、共享内存、套接字（socket）等，下面提供一个通过管道将反调试进程和主进程联系起来的简单例子：

```
#!cpp
int pipefd[2];
int childpid;

void *anti3_thread(void *){

    int statue=-1,alive=1,count=0;

    close(pipefd[1]);

    while(read(pipefd[0],&statue,4)>0)
        break;
    sleep(1);

    //这里改为非阻塞
    fcntl(pipefd[0], F_SETFL, O_NONBLOCK); //enable fd的O_NONBLOCK

    LOGI("pip-->read = %d", statue);

    while(true) {

        LOGI("pip--> statue = %d", statue);
        read(pipefd[0], &statue, 4);
        sleep(1);

        LOGI("pip--> statue2 = %d", statue);
```

```
        if (statue != 0) {
            kill(childpid,SIGKILL);
            kill(getpid(), SIGKILL);
            return NULL;
        }
        statue = -1;
    }
}

void anti3(){
    int pid,p;
    FILE *fd;
    char filename[MAX];
    char line[MAX];

    pid = getpid();
    sprintf(filename,"/proc/%d/status",pid);// 读取proc/pid/status中的TracerPid
    p = fork();
    if(p==0) //child
    {
        close(pipefd[0]); //关闭子进程的读管道
        int pt,alive=0;
        pt = ptrace(PTRACE_TRACEME, 0, 0, 0); //子进程反调试
        while(true)
        {
            fd = fopen(filename,"r");
            while(fgets(line,MAX,fd))
            {
                if(strstr(line,"TracerPid") != NULL)
                {
                    int statue = atoi(&line[10]);
                    LOGI("########## tracer pid:%d", statue);
                    write(pipefd[1],&statue,4);//子进程向父进程写 statue值

                    fclose(fd);

                    if(statue != 0)
                    {
                        return ;
                    }

                    break;
                }
            }
            sleep(1);

        }
    }else{
        childpid = p;
    }
}
pipe(pipefd);
pthread_create(&id_0,NULL,anti3_thread,(void*)NULL);
anti3();
```

传统检测TracerPid的方法是直接在子进程中循环检测，一旦发现则主动杀死进程。本实例将循环检测TracerPid和进程间通信结合，一旦反调试子进程被挂起或被杀死，父进程也会马上终止，原理大致如下图：



父进程的守护线程在从pipe中read到statue值之前，默认statue值为-1，收到子进程往pipe中写的statue值之后，重置statue值，如果未被调试，statue值为0，反之则为被调试状态。该做法的优势在于，一旦反调试进程被终止或被挂起，守护线程也能马上发现。

当然，如果通过hook或者修改kernel同样可以轻易的绕过这种反调试。这种做法只是为了演示而写的简单例子，真实的进程间通信反调试可以写的复杂的多，大家可以尽情发挥想象。

0x07 dalvik 虚拟机内部相关字段

在dalvik虚拟机中自带了检测调试器的代码，其本质是检测DvmGlobals结构体中的相关字段：

```
#!cpp
struct DvmGlobals {
    …
    bool   debuggerConnected;    /* debugger or DDMS is connected */
    bool   debuggerActive;        /* debugger is making requests */
    …
}
```

检测调试器的函数：

```
#!cpp
/*
 * static boolean isDebuggerConnected()
 *
 * Returns "true" if a debugger is attached.
 */
static void Dalvik_dalvik_system_VMDebug_isDebuggerConnected(const u4* args, JValue* pResult)
{
    UNUSED_PARAMETER(args);
    RETURN_BOOLEAN(dvmDbgIsDebuggerConnected());
}
```

本质是检测该dalvik虚拟机中DvmGlobals结构体中的调试器状态字段：

```
#!cpp
bool dvmDbgIsDebuggerConnected()
{
    return gDvm.debuggerActive;
}
```

知道原理之后可以更进一步，不通过这些Dalvik虚拟机的自定义函数，而是直接获取这些字段值，这样可以更好的隐藏反调试信息。

0x08 IDA arm、thumb指令识别缺陷

众所周知，IDA采用递归下降算法来反汇编指令，而该算法最大的缺点在于它无法处理间接代码路径，无法识别动态算出来的跳转。而arm架构下由于存在arm和thumb指令集，就涉及到指令集切换，IDA在某些情况下无法智能识别arm和thumb指令，比如下图所示代码：

```
.text:00005D4E                  LDR     R3, [SP,#arg_124]
.text:00005D50                  BX      R3
.text:00005D50 ; End of function sub_5D28
.text:00005D50
.text:00005D50 ; ---------------------------------------------
.text:00005D52                  ALIGN 4
.text:00005D54                  CODE32
.text:00005D54                  AND     LR, R0, R8
.text:00005D58                  AND     LR, R8, R0,LSL R0
.text:00005D5C                  AND     LR, R12, R0,LSL R0
.text:00005D60                  LDRMIBT LR, [R4],-R6
```


bx r3指令会切换指令集，而参数r3是动态计算出来的，IDA无法失败r3的值，而默认将bx r3后面的指令当成跳转地址，将后面地址的指令识别成了arm指令，而实际上其仍为thumb指令。

在IDA动态调试时，仍然存在该问题，若在指令识别错误的地点写入断点，有可能使得调试器崩溃。

0x09 Ptrace

Ptrace是gdb等调试器实现的核心，通过ptrace可以监控、控制被调试进程的状态、信号、执行等。而每个进程在同一时刻最多只能被一个调试进程ptrace，根据这个原理，可以主动ptrace自己的关键子进程，这样可以在一定程度上防止子进程被调试。

为了防止fork出来的反调试子进程被直接挂起或杀死，可以通过Ptrace的PTRACE_PEEKTEXT、PTRACE_PEEKDATA、PTRACE_POKETEXT等参数来完成父子进程之间的通信，比如子进程中使用的解密密钥先存于父进程空间，父进程往ptrace的子进程中写入密钥后，再解密出关键数据。

总之，通过ptrace增加父子进程之间的联系，是十分有效并且广泛存在于各类加固的反调试方法。

0x0A Inotify 监控文件

在Linux下，inotify可以实现监控文件系统事件（打开、读写、删除等），加固方案可以通过inotify监控apk自身的某些文件，某些内存dump技术通过/proc/pid/maps、/proc/pid/mem来实现内存dump，所以监控对这些文件的读写也能起到一定的反调试效果。

0x0B 总结

本文总结了主流加固厂商大部分反调试技巧，APK下的反调试技巧和win、linux下的大同小异，核心原理都是类似的。说到底，反调试只能尽可能的增加逆向难度，APK的安全防护绝不能仅仅依靠反调试，APK安全需要从整体架构上入手，在关键代码上加入强混淆，甚至通过vmp来增大关键代码的逆向难度。

0x0C Reference

- /bionic/linker/linker.h
- /bionic/linker/linker.cpp
- http://androidxref.com/4.4.4_r1/xref/bionic/linker/rt.cpp#33
- http://androidxref.com/4.4.4_r1/xref/dalvik/vm/native/dalvik_system_VMDebug.cpp
- http://blog.jobbole.com/23632/
- http://www.spongeliu.com/165.html