ANDROID调试检测技术汇编

1. IDA调试端口检测

原理：
调试器远程调试时，会占用一些固定的端口号。

做法：
读取/proc/net/tcp，查找IDA远程调试所用的23946端口，若发现说明进程正在被IDA调试。（也可以运行netstat -apn结果中搜索23946端口）

```
void CheckPort23946ByTcp()
{
    FILE* pfile=NULL;
    char buf[0x1000]={0};
    // 执行命令
    char* strCatTcp= "cat /proc/net/tcp |grep :5D8A";
    //char* strNetstat="netstat |grep :23946";
    pfile=popen(strCatTcp,"r");
    if(NULL==pfile)
    {
        LOGA("CheckPort23946ByTcp popen打开命令失败!\n");
        return;
    }
    // 获取结果
    while(fgets(buf,sizeof(buf),pfile))
    {
        // 执行到这里，判定为调试状态
        LOGA("执行cat /proc/net/tcp |grep :5D8A的结果:\n");
        LOGB("%s",buf);
    }//while
    pclose(pfile);
}
```

2. 调试器进程名检测

原理：
远程调试要在手机中运行android_server gdbserver gdb等进程。

做法：
遍历进程，查找固定的进程名，找到说明调试器在运行。

```
void SearchObjProcess()
{
    FILE* pfile=NULL;
    char buf[0x1000]={0};
    // 执行命令
    //pfile=popen("ps | awk '{print $9}'","r"); // 部分不支持awk命令
    pfile=popen("ps","r");
    if(NULL==pfile)
    {
        LOGA("SearchObjProcess popen打开命令失败!\n");
        return;
    }
    // 获取结果
    LOGA("popen方案:\n");
    while(fgets(buf,sizeof(buf),pfile))
    {
        // 打印进程
        LOGB("遍历进程:%s\n",buf);
        // 查找子串
        char* strA=NULL,strB=NULL,strC=NULL,strD=NULL;
        strA=strstr(buf,"android_server");
        strB=strstr(buf,"gdbserver");
        strC=strstr(buf,"gdb");
        strD=strstr(buf,"fuwu");
        if(strA || strB ||strC || strD)
        {
            // 执行到这里，判定为调试状态
            LOGB("发现目标进程:%s\n",buf);
        }//if
    }//while
    pclose(pfile);
}
```

3 父进程名检测

原理：
有的时候不使用apk附加调试的方法进行逆向，而是写一个.out可执行文件直接加载so进行调试，这样程序的父进程名和正常启动apk的父进程名是不一样的。

测试发现：
（1）正常启动的apk程序：父进程是zygote
（2）调试启动的apk程序：在AS中用LLDB调试发现父进程还是zygote
（3）附加调试的apk程序：父进程是zygote
（4）vs远程调试 用可执行文件加载so:父进程名为gdbserver
结论：父进程名非zygote的，判定为调试状态。

做法：
读取/proc/pid/cmdline，查看内容是否为zygote

```
void CheckParents()
{
    ////////////////////
    // 设置buf
    char strPpidCmdline[0x100]={0};
    snprintf(strPpidCmdline, sizeof(strPpidCmdline), "/proc/%d/cmdl
ine", getppid());
    // 打开文件
    int file=open(strPpidCmdline,O_RDONLY);
    if(file<0)
    {
        LOGA("CheckParents open错误!\n");
        return;
    }
    // 文件内容读入内存
    memset(strPpidCmdline,0,sizeof(strPpidCmdline));
    ssize_t ret=read(file,strPpidCmdline,sizeof(strPpidCmdline));
    if(-1==ret)
    {
        LOGA("CheckParents read错误!\n");
        return;
    }
    // 没找到返回0
    char sRet=strstr(strPpidCmdline,"zygote");
    if(NULL==sRet)
    {
        // 执行到这里，判定为调试状态
        LOGA("父进程cmdline没有zygote子串!\n");
        return;
    }
    int i=0;
    return;
}
```

4 自身进程名检测

原理：
和上条一样，也是写个.out加载so来脱壳的场景，
正常进程名一般是apk的com.xxx这样的格式。

代码：
略

5 apk线程检测

原理：
同样.out加载so来脱壳的场景，
正常apk进程一般会有十几个线程在运行(比如会有jdwp线程)，
自己写可执行文件加载so一般只有一个线程，
可以根据这个差异来进行调试环境检测。

```
void CheckTaskCount()
{
    char  buf[0x100]={0};
    char* str="/proc/%d/task";
    snprintf(buf,sizeof(buf),str,getpid());
    // 打开目录:
    DIR* pdir = opendir(buf);
    if (!pdir)
    {
        perror("CheckTaskCount open() fail.\n");
        return;
    }
    // 查看目录下文件个数:
    struct dirent* pde=NULL;
    int Count=0;
    while ((pde = readdir(pdir)))
    {
        // 字符过滤
        if ((pde->d_name[0] <= '9') && (pde->d_name[0] >= '0'))
        {
            ++Count;
            LOGB("%d 线程名称:%s\n",Count,pde->d_name);
        }
    }
    LOGB("线程个数为：%d",Count);
    if(1>=Count)
    {
        // 此处判定为调试状态.
        LOGA("调试状态!\n");
    }
    int i=0;
    return;
}
```

6 apk进程fd文件检测

原理：
根据/proc/pid/fd/路径下文件的个数差异，判断进程状态。
（apk启动的进程和非apk启动的进程fd数量不一样）
（apk的debug启动和正常启动，进程fd数量也不一样）

代码：
略

7 安卓系统自带调试检测函数

// android.os.Debug.isDebuggerConnected();
原理：
分析android自带调试检测函数isDebuggerConnected()在native的实现，
尝试在native使用。

做法：
（1）dalvik模式下:
找到进程中libdvm.so中的dvmDbgIsDebuggerConnected()函数，
调用他就能得知程序是否被调试。
dlopen(/system/lib/libdvm.so)
dlsym(_Z25dvmDbgIsDebuggerConnectedv)

（2）art模式下：
art模式下，结果存放在libart.so中的全局变量gDebuggerActive中，
符号名为_ZN3art3Dbg15gDebuggerActiveE。
但是貌似新版本android不允许使用非ndk原生库，dlopen(libart.so)会失败。
所以无法用dalvik那样的方法了。
有一种麻烦的方法，手动在内存中搜索libart模块，然后手工寻找该全局变量符号。

```
// 只写了dalvik的代码，art的就不写了
typedef unsigned char wbool;
typedef  wbool (*PPP)();
void NativeIsDBGConnected()
{
    void* Handle=NULL;
    Handle=dlopen("/system/lib/libdvm.so", RTLD_LAZY);
    if(NULL==Handle)
    {
        LOGA("dlopen打开libdvm.so失败!\n");
        return;
    }
    PPP Fun = (PPP)dlsym(Handle, "_Z25dvmDbgIsDebuggerConnectedv");
    if(NULL==Fun)
    {
        LOGA("dlsym获取_Z25dvmDbgIsDebuggerConnectedv失败!\n");
        return;
    }
    else
    {
        wbool ret = Fun();
        if(1==ret)
        {
            // 此处判定为调试模式
            LOGA("dalvikm模式，调试状态!\n");
            return;
        }
    }
    return;
}
```

8 ptrace检测

原理：
每个进程同时刻只能被1个调试进程ptrace，再次p自己会失败。

做法：
1 主动ptrace自己,根据返回值判断自己是否被调试了。
2 或者多进程ptrace。

```
// 单线程ptrace
void ptraceCheck()
{
    // ptrace如果被调试返回值为-1，如果正常运行，返回值为0
    int iRet=ptrace(PTRACE_TRACEME, 0, 0, 0);
    if(-1 == iRet)
    {
        LOGA("ptrace失败，进程正在被调试\n");
        return;
    }
    else
    {
        LOGB("ptrace的返回值为:%d\n",iRet);
        return;
    }
}
```

9 函数hash值检测

原理：
so文件中函数的指令是固定，但是如果被下了软件断点，指令就会发生改变(断点地址被改写为bkpt断点指令)，可以计算内存中一段指令的hash值进行校验，检测函数是否被修改或被下断点。

代码：
略

10 断点指令检测

原理：
上面说了，如果函数被下软件断点，则断点地址会被改写为bkpt指令，
可以在函数体中搜索bkpt指令来检测软件断电。

```
// IDA 6.8 断点扫描
// 参数1：函数首地址 参数2：函数size
typedef uint8_t u8;
typedef uint32_t u32;
void checkbkpt(u8* addr,u32 size)
{
    // 结果
    u32 uRet=0;
    // 断点指令
//    u8 armBkpt[4]={0xf0,0x01,0xf0,0xe7};
//    u8 thumbBkpt[2]={0x10,0xde};
    u8 armBkpt[4]={0};
    armBkpt[0]=0xf0;
    armBkpt[1]=0x01;
    armBkpt[2]=0xf0;
    armBkpt[3]=0xe7;
    u8 thumbBkpt[2]={0};
    thumbBkpt[0]=0x10;
    thumbBkpt[1]=0xde;
    // 判断模式
    int mode=(u32)addr%2;
    if(1==mode) {
        LOGA("checkbkpt:(thumb mode)该地址为thumb模式\n");
        u8* start=(u8*)((u32)addr-1);
        u8* end=(u8*)((u32)start+size);
        // 遍历对比
        while(1)
        {
            if(start >= end) {
                uRet=0;
                LOGA("checkbkpt:(no find bkpt)没有发现断点.\n");
                break;
            }
            if( 0==memcmp(start,thumbBkpt,2) ) {
                uRet=1;
                LOGA("checkbkpt:(find it)发现断点.\n");
                break;
            }
            start=start+2;
        }//while
    }//if
    else
    {
        LOGA("checkbkpt:(arm mode)该地址为arm模式\n");
        u8* start=(u8*)addr;
        u8* end=(u8*)((u32)start+size);
        // 遍历对比
        while(1)
```

```
            {
                if (start >= end) {
                    uRet = 0;
                    LOGA("checkbkpt:(no find)没有发现断点.\n");
                    break;
                }
                if (0 == memcmp(start,armBkpt , 4)){
                    uRet = 1;
                    LOGA("checkbkpt:(find it)发现断点.\n");
                    break;
                }
                start = start + 4;
            }//while
        }//else
        return;
    }
```

11 系统源码修改检测

原理：
安卓native下最流行的反调试方案是读取进程的status或stat来检测tracepid，原理是调试状态下的进程tracepid不为0。

对于这种调试检测手段，最彻底的绕过方式是修改系统源码后重新编译，让tracepid永远为0。

对抗这种bypass手段，我们可以创建一个子进程，让子进程主动ptrace自身设为调试状态，此时正常情况下，子进程的tracepid应该不为0。此时我们检测子进程的tracepid是否为0，如果为0说明源码被修改了。

```
bool checkSystem()
{
    // 建立管道
    int pipefd[2];
    if (-1 == pipe(pipefd)){
        LOGA("pipe() error.\n");
        return false;
    }
    // 创建子进程
    pid_t pid = fork();
    LOGB("father pid is: %d\n",getpid());
    LOGB("child  pid is: %d\n",pid);
    // for失败
    if(0 > pid) {
        LOGA("fork() error.\n");
        return false;
    }
    // 子进程程序
    int childTracePid=0;
    if ( 0 == pid )
    {
        int iRet = ptrace(PTRACE_TRACEME, 0, 0, 0);
        if (-1 == iRet)
        {
            LOGA("child ptrace failed.\n");
            exit(0);
        }
        LOGA("%s ptrace succeed.\n");
        // 获取tracepid
        char pathbuf[0x100] = {0};
        char readbuf[100] = {0};
        sprintf(pathbuf, "/proc/%d/status", getpid());
        int fd = openat(NULL, pathbuf, O_RDONLY);
        if (-1 == fd) {
            LOGA("openat failed.\n");
        }
        read(fd, readbuf, 100);
        close(fd);
        uint8_t *start = (uint8_t *) readbuf;
        uint8_t des[100] = {0x54, 0x72, 0x61, 0x63, 0x65, 0x72, 0x5
0, 0x69, 0x64, 0x3A,0x09};
        int i   = 100;
        bool flag= false;
        while (--i)
        {
            if( 0==memcmp(start,des,10) )
            {
                start=start+11;
```

```
                childTracePid=atoi((char*)start);
                flag= true;
                break;
            }else
            {
                start=start+1;
                flag= false;
            }
        }//while
        if(false==flag) {
            LOGA("get tracepid failed.\n");
            return false;
        }
        // 向管道写入数据
        close(pipefd[0]);                               // 关闭管道读端
        write(pipefd[1], (void*)&childTracePid,4);  // 向管道写端写入
数据
        close(pipefd[1]);                               // 写完关闭管道写
端
        LOGA("child succeed, Finish.\n");
        exit(0);
    }
    else
    {
        // 父进程程序
        LOGA("开始等待子进程.\n");
        waitpid(pid,NULL,NULL);                         // 等待子进程
结束
        int buf2 = 0;
        close(pipefd[1]);                               // 关闭写端
        read(pipefd[0], (void*)&buf2, 4);               // 从读端读取
数据到buf
        close(pipefd[0]);                               // 关闭读端
        LOGB("子进程传递的内容为:%d\n", buf2);              // 输出内容
        // 判断子进程ptarce后的tracepid
        if(0 == buf2) {
            LOGA("源码被修改了.\n");
        }else{
            LOGA("源码没有被修改.\n");
        }
        return true;
    }
}
void smain()
{
    bool bRet=checkSystem();
    if(true==bRet)
        LOGA("check succeed.\n");
    else
```

```
            LOGA("check failed.\n");
        LOGB("main Finish pid:%d\n",getpid());
        return;
    }
```

12 单步调试陷阱

原理：
调试器从下断点到执行断点的过程分析：
1 保存：保存目标处指令
2 替换：目标处指令替换为断点指令
3 命中断点：命中断点指令(引发中断 或者说发出信号)
4 收到信号：调试器收到信号后，执行调试器注册的信号处理函数。
5 恢复：调试器处理函数恢复保存的指令
6 回退：回退PC寄存器
7 控制权回归程序.

主动设置断点指令/注册信号处理函数的反调试方案:
1 在函数中写入断点指令
2 在代码中注册断点信号处理函数
3 程序执行到断点指令，发出信号
分两种情况：
(1)非调试状态
进入自己注册的函数，NOP指令替换断点指令，回退PC后正常指令。
(执行断点发出信号—进入处理信号函数—NOP替换断点—退回PC)
(2)调试状态
进入调试器的断点处理流程，他会恢复目标处指令失败，然后回退PC，进入死循环。

```
#!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 clea
r 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 c
lear 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);

}
```

13 利用IDA先截获信号特性的检测

原理：
IDA会首先截获信号，导致进程无法接收到信号，导致不会执行信号处理函数。将关键流程放在信号处理函数中，如果没有执行，就是被调试状态。

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void myhandler(int sig)
{
    //signal(5, myhandler);
    printf("myhandler.\n");
    return;
}
int g_ret = 0;
int main(int argc, char **argv)
{
    // 设置SIGTRAP信号的处理函数为myhandler()
    g_ret = (int)signal(SIGTRAP, myhandler);
    if ( (int)SIG_ERR == g_ret )
        printf("signal ret value is SIG_ERR.\n");

    // 打印signal的返回值(原处理函数地址)
    printf("signal ret value is %x\n",(unsigned char*)g_ret);

    // 主动给自己进程发送SIGTRAP信号
    raise(SIGTRAP);
    raise(SIGTRAP);
    raise(SIGTRAP);
    kill(getpid(), SIGTRAP);

    printf("main.\n");
    return 0;
}
```

14 利用IDA解析缺陷反调试

原理：
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，进一步导致无法进行伪代码还原。

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。（ 可能写断点 ,不知道写ARM还是THUMB ,造成的崩溃 ）

```
#if(JUDGE_THUMB)
#define GETPC_KILL_IDAF5_SKATEBOARD     \
__asm __volatile(                       \
"mov    r0,pc         \n\t"             \
"adds   r0,0x9        \n\t"             \
"push   {r0}          \n\t"             \
"pop    {r0}          \n\t"             \
"bx     r0            \n\t"             \
\
".byte 0x00           \n\t"             \
".byte 0xBF           \n\t"             \
\
".byte 0x00           \n\t"             \
".byte 0xBF           \n\t"             \
\
".byte 0x00           \n\t"             \
".byte 0xBF           \n\t"             \
:::"r0"                                 \
);
#else
#define GETPC_KILL_IDAF5_SKATEBOARD     \
__asm __volatile(                       \
"mov    r0,pc         \n\t"             \
"add    r0,0x10       \n\t"             \
"push   {r0}          \n\t"             \
"pop    {r0}          \n\t"             \
"bx     r0            \n\t"             \
".int 0xE1A00000      \n\t"             \
".int 0xE1A00000      \n\t"             \
".int 0xE1A00000      \n\t"             \
".int 0xE1A00000      \n\t"             \
:::"r0"                                 \
);
#endif


// 常量标签版本
#if(JUDGE_THUMB)
#define IDAF5_CONST_1_2                     \
__asm __volatile(                           \
"b      T1              \n\t"               \
"T2:                    \n\t"               \
"adds   r0,1            \n\t"               \
"bx     r0              \n\t"               \
"T1:                    \n\t"               \
"mov    r0,pc           \n\t"               \
"b      T2              \n\t"               \
:::"r0"                                     \
```

```
);
#else
#define IDAF5_CONST_1_2                    \
__asm __volatile(                          \
"b      T1              \n\t"              \
"T2:                    \n\t"              \
"bx     r0              \n\t"              \
"T1:                    \n\t"              \
"mov    r0,pc           \n\t"              \
"b      T2              \n\t"              \
:::"r0"                                    \
);
#endif
```

15 五种代码执行时间检测

第一类：
原理：
一段代码，在a处获取一下时间，运行一段后，再在b处获取下时间，
然后通过(b时间-a时间)求时间差,正常情况下这个时间差会非常小，
如果这个时间差比较大，说明正在被单步调试。

做法：
五个能获取时间的api：
time()函数
time_t结构体

clock()函数
clock_t结构体

gettimeofday()函数
timeval结构
timezone结构

clock_gettime()函数
timespec结构

getrusage()函数
rusage结构

```c
#include <sys/time.h>
#include <sys/types.h>
#include <sys/resource.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>

static int _getrusage ();   //Invalid
static int _clock ();       //Invalid
static int _time ();
static int _gettimeofday ();
static int _clock_gettime ();

int main ()
{
    _getrusage ();
    _clock ();
    _time ();
    _gettimeofday ();
    _clock_gettime ();

    return 0;
}

int _getrusage ()
{
    struct rusage t1;
    /* breakpoint */
    getrusage (RUSAGE_SELF, &t1);

    long used = t1.ru_utime.tv_sec + t1.ru_stime.tv_sec;
    if (used > 2) {
        puts ("debugged");
    }
    return 0;
}


int _clock ()
{
    clock_t t1, t2;

    t1 = clock ();
    /* breakpoint */
    t2 = clock ();
    double used = (double)(t2 - t1) / CLOCKS_PER_SEC;
    if (used > 2) {
```

```
        puts ("debugged");
    }
    return 0;
}


int _time ()
{
    time_t t1, t2;
    time (&t1);
    /* breakpoint */
    time (&t2);
    if (t2 - t1 > 2) {
        puts ("debugged");
    }
    return 0;
}

int _gettimeofday ()
{
    struct timeval t1, t2;
    struct timezone t;

    gettimeofday (&t1, &t);
    /* breakpoint */
    gettimeofday (&t2, &t);

    if (t2.tv_sec - t1.tv_sec >2 ) {
        puts ("debugged");
    }
    return 0;
}

int _clock_gettime ()
{
    struct timespec t1, t2;

    clock_gettime (CLOCK_REALTIME, &t1);
    /* breakpoint */
    clock_gettime (CLOCK_REALTIME, &t2);

    if (t2.tv_sec - t1.tv_sec > 2) {
        puts ("debugged");
    }
    return 0;
}
```

16 三种进程信息结构检测

原理：
一些进程文件中存储了进程信息，可以读取这些信息得知是否为调试状态。

做法：
第一种：
/proc/pid/status
/proc/pid/task/pid/status
TracerPid非0
statue字段中写入t（tracing stop）

第二种：
/proc/pid/stat
/proc/pid/task/pid/stat
第二个字段是t（T）

第三种：
/proc/pid/wchan
/proc/pid/task/pid/wchan
ptrace_stop

代码：
略。

17 Inotify事件监控dump

原理：
通常壳会在程序运行前完成对text的解密，所以脱壳可以通过dd与gdb_gcore来dump /proc/pid/mem或/proc/pid/pagemap，获取到解密后的代码内容。

可以通过Inotify系列api来监控mem或pagemap的打开或访问事件，
一旦发生时间就结束进程来阻止dump。

```
void thread_watchDumpPagemap()
{
    LOGA("--------------------watchDump:Pagemap-------------------
\n");
    char dirName[NAME_MAX]={0};
    snprintf(dirName,NAME_MAX,"/proc/%d/pagemap",getpid());
    int fd = inotify_init();
    if (fd < 0)
    {
        LOGA("inotify_init err.\n");
        return;
    }
    int wd = inotify_add_watch(fd,dirName,IN_ALL_EVENTS);
    if (wd < 0)
    {
        LOGA("inotify_add_watch err.\n");
        close(fd);
        return;
    }
    const int   buflen=sizeof(struct inotify_event) * 0x100;
    char        buf[buflen]={0};
    fd_set      readfds;
    while(1)
    {
        FD_ZERO(&readfds);
        FD_SET(fd, &readfds);
        int iRet = select(fd+1,&readfds,0,0,0); // 此处阻塞
        LOGB("iRet的返回值:%d\n",iRet);
        if(-1==iRet)
            break;
        if (iRet)
        {
            memset(buf,0,buflen);
            int len = read(fd,buf,buflen);
            int i=0;
            while(i < len)
            {
                struct inotify_event *event = (struct inotify_even
t*)&buf[i];
                LOGB("1 event mask的数值为:%d\n",event->mask);
                if( (event->mask==IN_OPEN)  )
                {
                    // 此处判定为有true,执行崩溃.
                    LOGB("2 有人打开pagemap,第%d次.\n\n",i);
                    //__asm __volatile(".int 0x8c89fa98");
                }
```

```
                i += sizeof (struct inotify_event) + event->len;
            }
            LOGA("-----3 退出小循环-----\n");
        }
    }
    inotify_rm_watch(fd,wd);
    close(fd);
    LOGA("-----4 退出大循环,关闭监视-----\n");
    return;
}
void smain()
{
    // 监控/proc/pid/mem
    pthread_t ptMem,t,ptPageMap;
    int iRet=0;

    // 监控/proc/pid/pagemap
    iRet=pthread_create(&ptPageMap,NULL,(PPP)thread_watchDumpPagema
p,NULL);
    if (0!=iRet)
    {
        LOGA("Create,thread_watchDumpPagemap,error!\n");
        return;
    }
    iRet=pthread_detach(ptPageMap);
    if (0!=iRet)
    {
        LOGA("pthread_detach,thread_watchDumpPagemap,error!\n");
        return;
    }
    LOGA("--------------------smain-------------------\n");
    LOGB("pid:%d\n",getpid());
    return;
}
```

by fightclub

Reference:

1 Anti-debugging Skills in APK ——wooyun

2 Android逃逸技术汇编 ——360