# 1 反调试

[ANDROID调试检测技术汇编\_抚琴-CSDN博客](https://blog.csdn.net/qq_42186263/article/details/113711359)

[Anti-debugging Skills in APK - 国产大熊猫个人空间 - OSCHINA - 中文开源技术交流社区](https://my.oschina.net/9199771/blog/1584388)



## Ptrace检测

[TracerPid反调试实现与逆向\_明风的博客-CSDN博客](https://blog.csdn.net/tianxuhong/article/details/78415864)

原理： 每个进程同时刻只能被1个进程ptrace调试

做法： 1. 主动ptrace自己，根据返回值判断自己是否被调试

2. 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;      }  } |

Anti：frida replace ptrace函数

|  |
| --- |
| export function anti\_ptrace() {          var ptrace = Module.findExportByName(null, "ptrace");          if (null != ptrace) {              ptrace = ptrace.or(1);              log.i('anti\_ptrace', "ptrace addr: " + ptrace);  *// Interceptor.attach(ptrace, {*  *//     onEnter: function (args) {*  *//         log.i('anti\_ptrace', 'entry');*  *//     }*  *// });*              Interceptor.replace(ptrace.or(1), new NativeCallback(function (p1: any, p2: any, p3: any, p4: any) {                  log.i('anti\_ptrace', 'entry');                  return 1;              }, 'long', ['int', "int", 'pointer', 'pointer']));          }      } |

## 1.2 进程信息检查

[玩转Android10源码开发定制(六)修改内核源码绕过反调试检测\_xiaomaNo01的专栏-CSDN博客](https://blog.csdn.net/xiaomaNo01/article/details/111967193)

第一种：/proc/pid/status 和 /proc/pid/task/pid/status

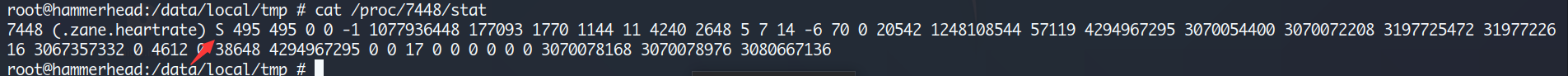
在Android 中调试状态下，linux内核会向/proc/$pid/status或者/proc/$pid/task/$pid/status 中写入进程状态信息。其中TracerPid 字段写入调试该进程的进程的的Pid。其中State字段中写入该进程当前处于的状态,取值如下之一:

R (running)", "S (sleeping)", "D (disk sleep)", "T (stopped)", "t(tracing stop)", "Z (zombie)",  "X (dead)"

当State为t (tracing stop)或者T (stopped)的时候,表示正被调试追踪。所以反调试的方法之一就是通过不断轮询读取/proc/$pid/status或者/proc/$pid/task/$pid/status文件检测检查TracerPid的值或者State的值，如果TracerPid非0说明该进程被调试;如果state值为t (tracing stop)或者T (stopped),说明被调试追踪。

第二种：/proc/pid/stat 和 /proc/pid/task/pid/stat

第二个字段是t（T）



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

wchan文件内容表示显示当进程sleep时，kernel当前运行的函数。若进程被调试，内核会往/proc/$pid/wchan、/proc/$pid/task/$pid/wchan文件中写入ptrace\_stop信息。

|  |
| --- |
| */\*\**  *\* @state\_name: cat /proc/xxx/stat ==> ...(<state\_name>) S...*  *\**  *\* anti fgets function include :*  *\* status->TracerPid, SigBlk, S (sleeping)*  *\* State->(package) S*  *\* wchan->SyS\_epoll\_wait*  *\*/*      export function anti\_fgets() {          const tag = 'anti\_fgets';          const fgetsPtr = Module.findExportByName(null, 'fgets');          log.i(tag, 'fgets addr: ' + fgetsPtr);          if (null == fgetsPtr) {              return;          }          var fgets = new NativeFunction(fgetsPtr, 'pointer', ['pointer', 'int', 'pointer']);          Interceptor.replace(fgetsPtr, new NativeCallback(function (buffer, size, fp) {              if (null == this) {                  return 0;              }              var logTag = null;  *// 进入时先记录现场*              const lr = fridaCommon.getLR(this.context);  *// 读取原 buffer*              var retval = fgets(buffer, size, fp);              var bufstr = (buffer as NativePointer).readCString();              if (null != bufstr) {                  if (bufstr.indexOf("TracerPid:") > -1) {                      buffer.writeUtf8String("TracerPid:\t0");                      logTag = 'TracerPid';                  }  *//State:    S (sleeping)*                  else if (bufstr.indexOf("State:\tt (tracing stop)") > -1) {                      buffer.writeUtf8String("State:\tS (sleeping)");                      logTag = 'State';                  }  *// ptrace\_stop*                  else if (bufstr.indexOf("ptrace\_stop") > -1) {                      buffer.writeUtf8String("sys\_epoll\_wait");                      logTag = 'ptrace\_stop';                  }  *//(sankuai.meituan) t*                  else if (bufstr.indexOf(") t") > -1) {                      buffer.writeUtf8String(bufstr.replace(") t", ") S"));                      logTag = 'stat\_t';                  }  *// SigBlk*                  else if (bufstr.indexOf('SigBlk:') > -1) {                      buffer.writeUtf8String('SigBlk:\t0000000000001204');                      logTag = 'SigBlk';                  }                  if (logTag) {                      log.i(tag + " " + logTag, bufstr + " -> " + buffer.readCString() + ' lr: ' + lr                          + "(" + fridaCommon.getModuleByAddr(lr) + ")");                  }              }              return retval;          }, 'pointer', ['pointer', 'int', 'pointer']));      } |

## 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);  } |

## 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);  } |

## 父进程名检测与自身进程名检测

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/cmdline", 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;  } |

## 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;  } |

## 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;  } |

## 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

## 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

## 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指令来检测软件断点

|  |
| --- |
| *// 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;  } |

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

## 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);  } |

## 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

|  |
| --- |
| #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;  } |

## 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 |

## 代码执行时间检测

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

|  |
| --- |
| #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;  } |

## 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;  } |

## rtld\_db\_dlactivity函数

rtld\_db\_dlactivity函数实则默认情况下为空函数，这里的值应该为0，而当有调试器时，这里会被改为断点指令，0xDE10实则为thumb指令的断点。函数的功能是用于处理一些调试器的特殊情况的。比如调试器对模块的某个地址进行下断，但这个地址实际不存在，得在模块加载后才存在等的特殊情况下的函数。起的作用就是类似于协商解决特殊问题。

# 2 JEB脚本开发

[三分钟上手JEB脚本开发 - 安全客，安全资讯平台 (anquanke.com)](https://www.anquanke.com/post/id/228981#h3-6)

[IJavaMethod | JEB API Documentation (pnfsoftware.com)](https://www.pnfsoftware.com/jeb/apidoc/reference/com/pnfsoftware/jeb/core/units/code/java/IJavaMethod.html)

[jeb-samplecode/scripts at master · pnfsoftware/jeb-samplecode · GitHub](https://github.com/pnfsoftware/jeb-samplecode/tree/master/scripts)

[[原创]JEB脚本(一)(指令解析 反编译 抽象语法树)-Android安全-看雪论坛-安全社区|安全招聘|bbs.pediy.com](https://bbs.pediy.com/thread-263011.htm#msg_header_h1_4)

# 3 IDA

## 3.1 结构体

[7.IDA-创建结构体\_liujiayu2的专栏-CSDN博客\_ida 结构体](https://blog.csdn.net/liujiayu2/article/details/77488078)

[[IDA] 在IDA里如何自定义数据结构\_盗世九昌的技术博客\_51CTO博客](https://blog.51cto.com/u_13992485/2476125)

# 4 情报研究网站

[研究报告-奇安信 (qianxin.com)](https://www.qianxin.com/threat/reportaptlist)

[360研究报告(安全研究列表页)，从数字中感知威胁，在对抗中排除隐患。基于360安全大数据，进行分析研究，将安全可视化，让大众了解数字背后的含义](http://zt.360.cn/2015/reportlist.html?list=1)

[信息安全知识库 vipread.com](https://vipread.com/index)

# 5 Linux动态链接原理GOT PLT表详解

[Linux动态连接原理 GOT PLT表详解\_lingfong\_cool的博客-CSDN博客](https://blog.csdn.net/lingfong_cool/article/details/7976112)

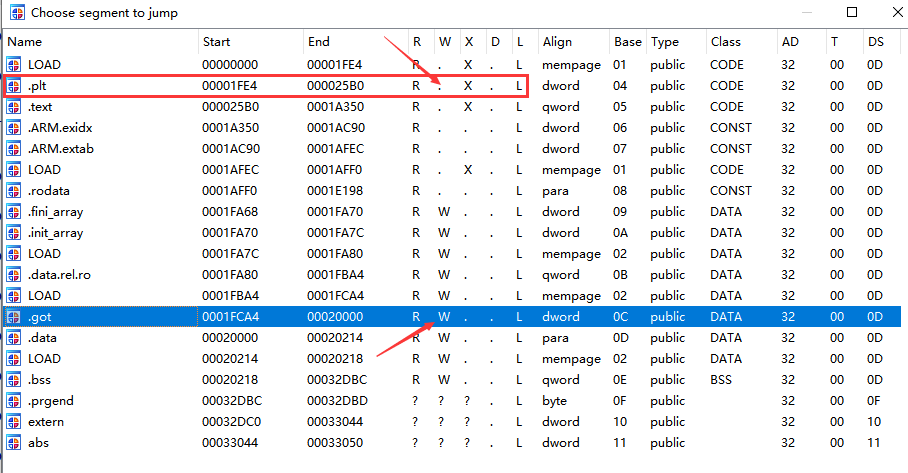
[GOT和PLT原理简析\_softee的专栏-CSDN博客](https://blog.csdn.net/softee/article/details/41256595)

[Linux动态链接为什么要用PLT和GOT表？ - 知乎 (zhihu.com)](https://www.zhihu.com/question/21249496)

[聊聊Linux动态链接中的PLT和GOT（１）——何谓PLT与GOT\_海枫的专栏-CSDN博客](https://blog.csdn.net/linyt/article/details/51635768)

动态链接时，因为不知道模块加载位置，将地址相关代码抽出，放在数据段中就是got表。为了实现地址的延迟绑定，再加了一个中间层，是一小段精巧的指令，用于在运行中填充got表。这些指令组成plt表。

存放函数地址的数据表，称为重局偏移表（GOT, Global Offset Table），而那个额外代码段表，称为程序链接表（PLT，Procedure Link Table）。



如果可执行文件调用的动态库函数很多时，那在进程初始化时都对这些函数做地址解析和重定位工作，大大增加进程的启动时间。所以Linux提出延迟重定位机制，只有动态库函数在被调用时，才会地址解析和重定位工作。

进程启动时，先不对GOT表项做重定位，等到要调用该函数时才做重定位工作。要实现这个机制必须要有一个状态描述该GOT表项是否已完重定位。

\_dl\_runtime\_resolve

# 6 壳

## 6.1 360

[[原创][原创]根据”so劫持”过360加固详细分析-Android安全-看雪论坛-安全社区|安全招聘|bbs.pediy.com](https://bbs.pediy.com/thread-223796.htm)

### 6.1.1 反调试

1.rtld\_db\_dlactivity函数

2.TracePid

3.端口检测

4.时间差

## 6.2 乐固

[[原创]通过一款早期代码抽取壳入门学习 so 层分析-Android安全-看雪论坛-安全社区|安全招聘|bbs.pediy.com](https://bbs.pediy.com/thread-260251.htm)

[[原创]分析一下乐固-Android安全-看雪论坛-安全社区|安全招聘|bbs.pediy.com](https://bbs.pediy.com/thread-266972.htm)

[GitHub - xiaobaiyey/dexload](https://github.com/xiaobaiyey/dexload)

## 6.2.1 dex加载流程

我们知道 App 能够运行起来一定因为是我们写的代码编译成的 Apk、Dex 文件或者是 jar 文件被安卓系统加载起来了，知道这个有助于我们理解 ClassLoader 的存在，ClassLoder 负责加载这些文件，就包括我们要重点要说的 Dex 文件。

需要知道的是 DexClassLoader 和 PathClassLoader，前者能够加载 jar/apk/dex，而后者只能加载系统中已经安装的 Apk 中的 Dex。两者都有共同的父类 BaseDexClassLoader，在 BaseDexClassLoader 的构造方法中，有一个核心功能的类 DexPathList，负责解析加载文件的类。

在 DexPathList 的构造方法中，有一个方法 makeDexElements，makeDexElements 方法判断文件后缀名是否是“dex”，如果是就调用 LoadDexFile 方法加载文件，并且返回一个 DexFile 对象。

而我们的 Dex 文件就是在生成这个 DexFile 对象调用它的构造方法时候，被更具体和底层地进行了加载。在 DexFile 的构造方法中调用了一个 native 方法 openDexFile。这个 native 方法返回了一个极为重要的值，让我们记住它并在不断地学习中加深对它的理解，这个值就是 cookie。

cookie 在 Java 层就是虚拟机的 cookie 值，在 so 层它是 pDexOrJar 指针，虚拟机在进行查找 Dex 文件中的类方法时候，都是需要对 cookie 进行操作的。

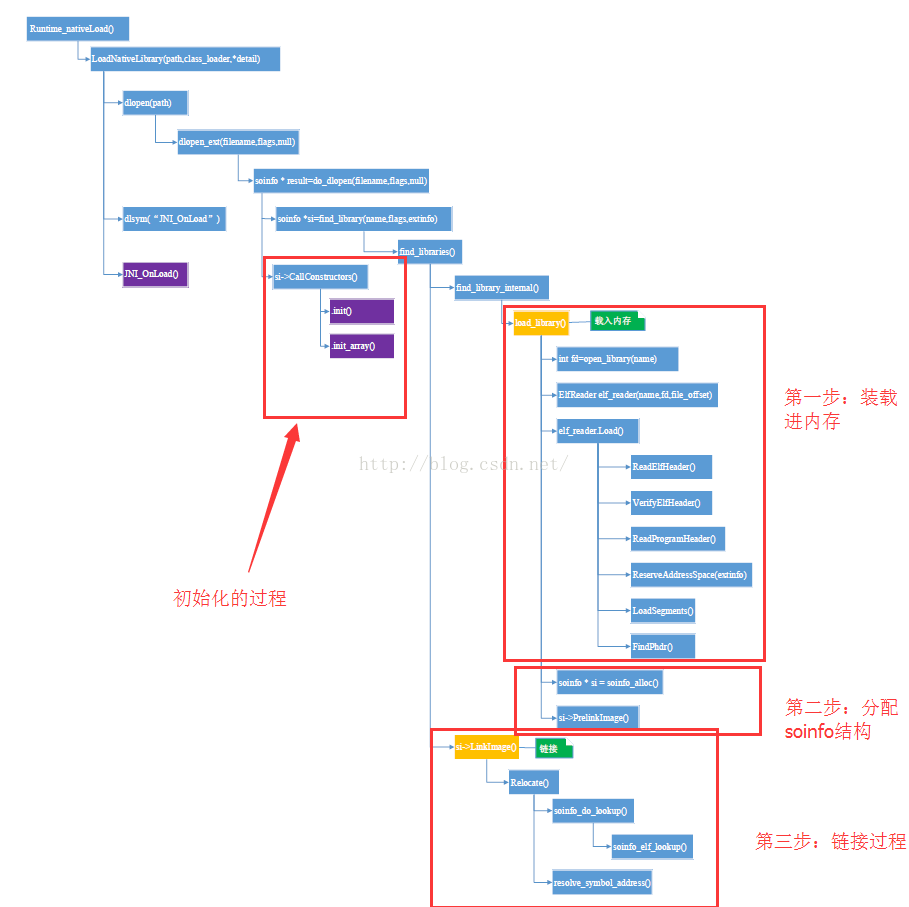
查找的过程是，BaseClassLoader类 的 findClass 方法，调用 DexPathList类 的 findClass 方法，然后调用刚才说到的返回的 DexFile 对象的 loadClassBinaryName 方法。在 loadClassBinaryName 方法中把查找的类方法名称、类加载器、cookie 值，作为三个参数传进了最后的 defineClass 方法，这个方法调用了 native 函数来返回一系列调用想要查找的 class。然后就是 loadClass 和 defineClass 的操作。

#### 6.2.2.1 乐固hook mmap execve execv

dlpen函数调用：OpenDexFile->openDexFileNative->OpenDexFilesFromOat->LoadDexFiles->OpenDexFile->DexFile::Open->OatFileBase::OpenOatFile-> OatFileBase::OpenOatFile->Dlopen

# 7 linker

[ELF Linker学习篇（一）关于ELF文件装载进内存\_雪一梦的博客-CSDN博客](https://blog.csdn.net/feibabeibei_beibei/article/details/52986326)



# 8 汇编

[AndroidSecNotes/ARM 寄存器简介.md at master · JnuSimba/AndroidSecNotes · GitHub](https://github.com/JnuSimba/AndroidSecNotes/blob/master/Android%E9%80%86%E5%90%91%E5%9F%BA%E7%A1%80/ARM%20%E5%AF%84%E5%AD%98%E5%99%A8%E7%AE%80%E4%BB%8B.md)

# 9 JNI

1. svc系统调用 ： svc 0x0

[Android 6.0 JNI原理分析 和 Linux系统调用(syscall)原理\_dk\_work的博客-CSDN博客](https://blog.csdn.net/q1183345443/article/details/62227664)

# 10 动态调试

## 10.1 调试启动

$adb shell am start -D -n package/activitiyname

$jdb -connect com.sun.jdi.SocketAttach:hostname=localhost,port=8700

Setprop：

/data/local/tmp/mprop

setprop ro.debuggable 1

/data/local/tmp/mprop -r

getprop ro.debuggable #查看当前属性

## 10.2 ida脚本

linker下断点： Import mycustom

跳转到目标地址：idc.jumpto(module.base + 0x13548)

# 11 Online disassembler

[disasm.pro | Online Assembler and Disassembler](https://disasm.pro/)

[shell-storm | Online Assembler and Disassembler](http://shell-storm.org/online/Online-Assembler-and-Disassembler/)