8.7

**概述：**

这两周的进展包括：

1.掌握JNI的实现过程，即java平台调用底层C/C++代码。

2.学习了Android系统 Java 函数拦截的大体思想（bug太多，正在实现过程中）

**详解：**

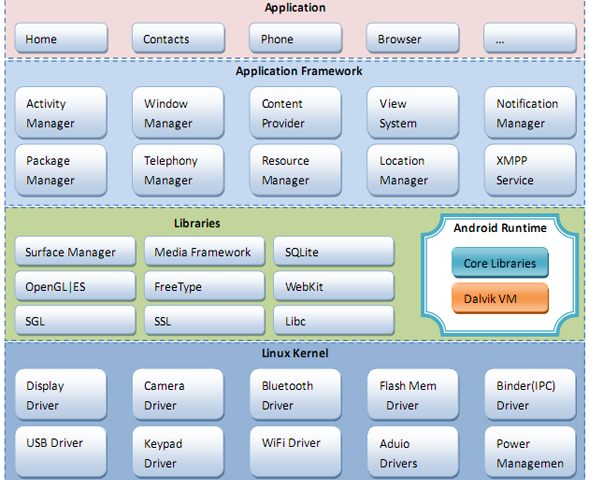
一.

JNI是什么？

JNI([Java](http://lib.csdn.net/base/java)Native Interface)意为[Java](http://lib.csdn.net/base/java)本地调用，它允许Java代码和其他语言写的代码进行交互，简单的说，一种在Java虚拟机控制下执行代码的标准机制。

为什么使用JNI（与本项目有什么关系吗）？

Android 应用开发以Java 为主，但也可以使用 C/C++语言，而且，由Android系统框架图可知，Android上层的application层和Application Framework层都是使用Java语言编写，底层包括Linux 内核层与系统运行库层都是由C/C++语言编写的，在这里我们将上层采用 Java语言编写的统称为 Java 层，下层采用 C/C++语言编写的统称为Native层。由分层架构所知，Native 层为上层Java 层提供服务，因此，当Java 代码要调用底层的 C/C++函数库就必须通过 Java的 JNI机制来实现。



Android系统框架图

总结一下流程：

编写静态方法（用java声明）-->编译生成class文件--->编译生成h文件---->编写C文件（用C/C++实现）

---->配置NDK---->配置so库---->在Activity调用（Java调用C/C++）。

具体实现：

1.创建一个名为HelloJni的android应用程序。

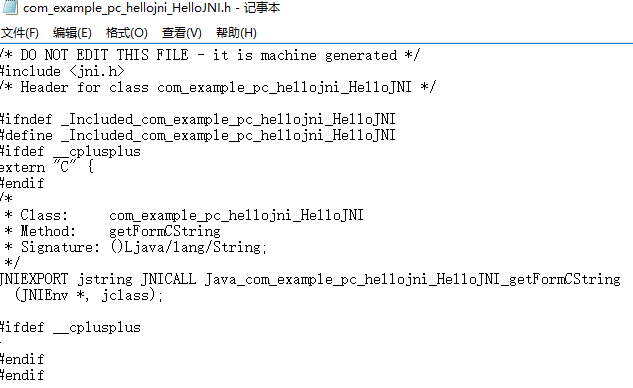
2.新建HelloJNI.Java;

3.利用javah生成c语言头文件。在执行javah命令之前，应先编译java文件为class文件，执行菜单栏上的Building->Make Project即可。然后打开Android Studio底部的Terminal执行命令：

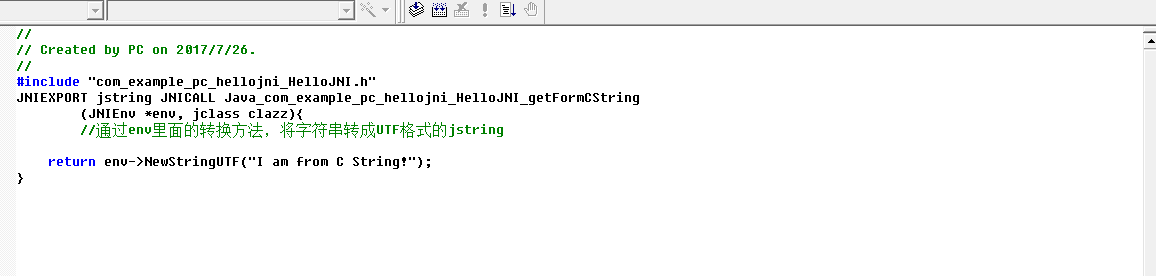
javah –d src\main\jni\ -classpath build\intermediates\classes\debug

<包名>.hellojni.HelloJNI

4.完成第三步之后将在jni目录下生成对应的.h头文件。

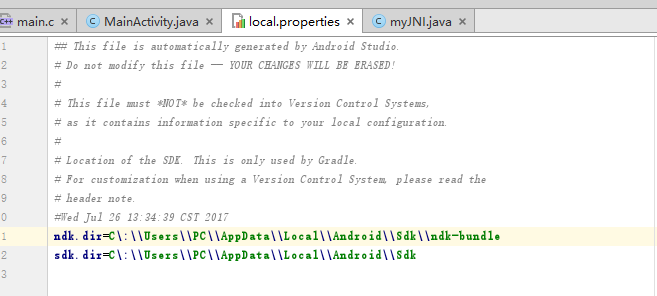


5.编写cpp文件，上面已经用javah工具生成了头文件。接下来我们要实现头文件里面的方法。在jni目录下新建文件HelloJNI.cpp:



6.配置NDK

打开Project的local.properties文件添加NDK路径



打开app Module的build.gradle文件，在defaultConfig节点里添加以下代码  
注意这里的moduleName，是我们在之前自己编写的类里面加载的so库名

ndk {

moduleName "JniTest"

ldLibs "log", "z", "m"

abiFilters "armeabi", "armeabi-v7a", "x86"

}

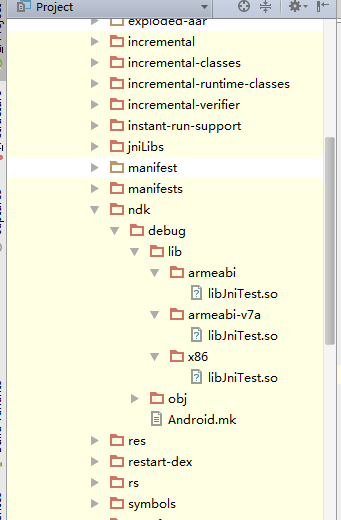
还要在gradle.properties里面加上这么一句话：

android.useDeprecatedNdk=true

7.生成SO库

完成以上步骤之后，我们rebuild一下就可以生成so库了

在项目的app\build\intermediates\ndk\debug\lib路径下



8. 、配置so库

在src\main下新建文件夹jniLIB，并将生成的SO文件拷贝到该文件夹下

9.在Activity中布局一个TextView用来显示字符串：

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

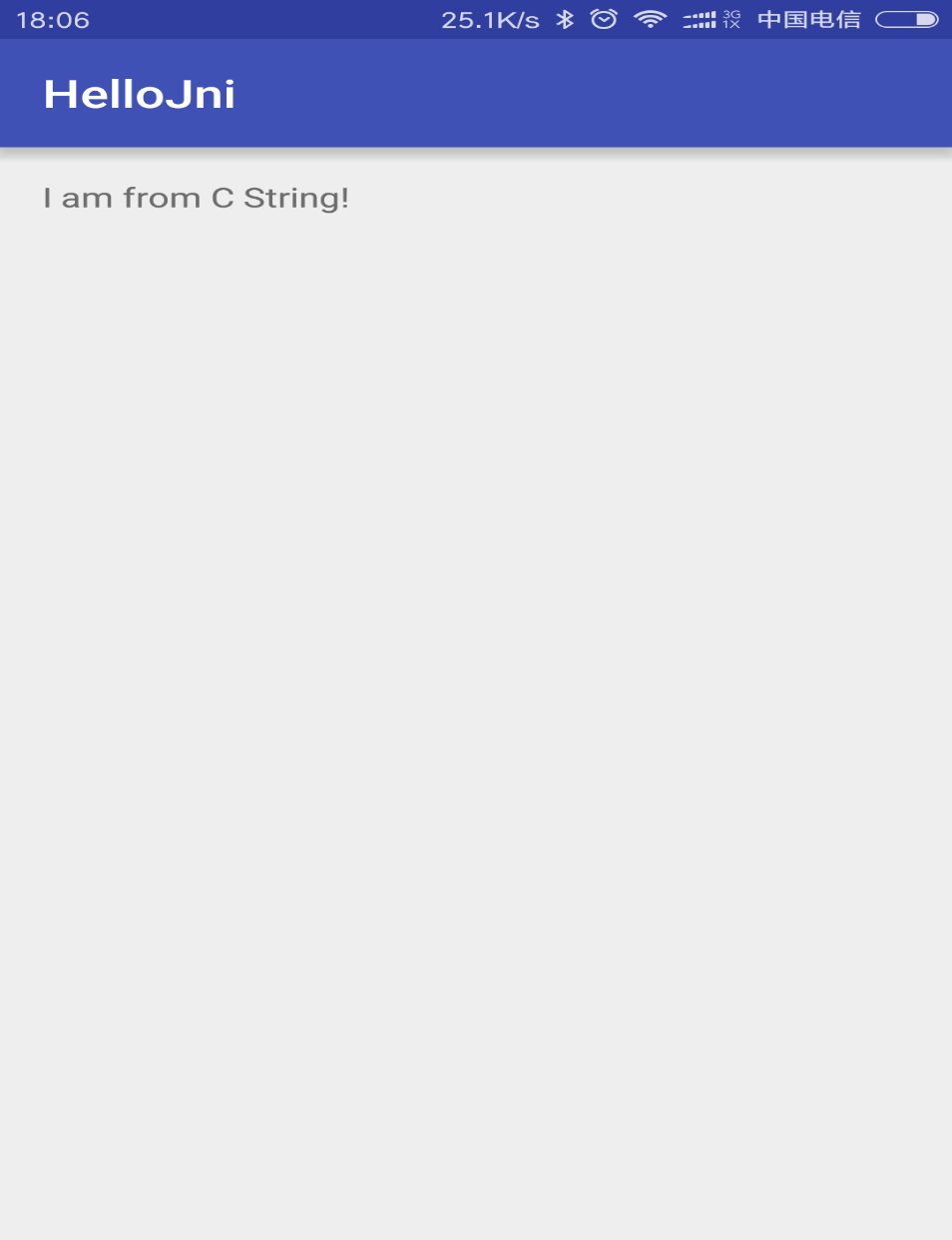
setContentView(R.layout.activity\_main);

TextView tv= (TextView) findViewById(R.id.text);

tv.setText(HelloJNI.getFormCString());

}

10.运行程序，不出现编译错误，手机上将会看到结果图：



反思：遇到的bug

1.生成的C函数参数是（JNIEnv , jobject）而不是（JNIEnv , jclass）

解决：这和java代码中对Native函数的声明有关，声明为static，这里的参数就是jclass,即代表该函数所在的类(如HelloJNI.getFromString()，这是jclass接收的是HelloJNI.class)。如果没有声明static，这里的参数就是jobject，表示调用者的实体对象。

2.报java.lang.UnsatisfiedLinkError错误；

解决：此类错误一般后面有详细的解释，此处列出常见的错误：

查看Logcat，显示Can’t load 64-bit .so on 32-bit platform 也就是说不能在32位环境中载入64位的so。把so移到armeabi-v8a 文件夹下即可。该文件夹存放的是arm64位so文件。

二.

我们将apk文件进行拆包，发现其中有一个classes.dex文件，这就是我们的工程代码最终编译生成的文件，也是Android上的可执行文件。D ex文件中保存的是Dalvik字节码，也就是说，我们编写的代码最终会被编译成Dalvik字节码，由Dalvik虚拟机解释执行。那么，Dalvik虚拟机是如何区分Java函数和Native函数的呢?

在Dalvik虚拟机中，无论是Java函数还是Native函数，都是通过Method结构

体来描述的，在Android源码dalvik/vm/oo/Obj ect. h中是这样定义Method结构体的:

**struct Method{**

**ClassObject\* clazz;**

**u4 accessFlags**; /\*访问标志;低16位由spec定义（可以是u2)\*/

**………**/\*实际内容的代码/

**const u2\* insns;**/\*指令，在内存映射的.dex中\*/

………

**DalvikBridgeFunc nativeFunc;**

}

其中如果access Flags为ACC一ATIVE,则表明该函数为Native函数，否则为Java

，Dalvik虚拟机通过dvmIsNativeMethod函数判断一个函数Java函数还是Native

**INLINE bool dvmIsNativeMethod(const Method\* method){**

**return(method->accessFlags&ACC一ATIVE)!=0;**

**}**

对于Java函数，Dalvik虚拟机使用解释器来执行此函数;对于Native函数，

Dalvik虚拟机找到它的函数指针即method->nativeFunc，进行直接调用即可。

**如果我们能够获取某个函数对应的在dalvik虚拟中的结构体，把Java函数修改为Native函数，并且将nativeFunc指针设置为自定义的函数，那样就可以实现**

**拦截了，拦截完成之后，可以根据情况决定是否需要调用原来的Java函数，即可完成整个拦截过程。**

在Android中，dalvik虚拟机实现在libdvm. so中，libdvm. so导出了两个函数

dvmDecodeIndirectRef和dvmSlotToMethod，如果我们知道一个Java函数在它所属的Class里面的位置Slot，那么就可以通过它们获得该Java函数在Dalvik虚拟机内部所对应的Method结构体:

ClassObject\* declared\_ class=(ClassObject\*)dvmDecodeIndirectRef(dvmTheadself(),

declared\_ class\_ indirect);

Method\* method=dvmSlotToMethod(declared\_ class,slot);

if(method==NULL){

dvmThrowNoSuchMethodError("Failed to get internal representation for method");

return;

}

要想获得一个Java函数所属的Class以及它在该Class的位置Slot，我们首先要

知道：

Java函数的名称一methodName

Java函数的原型一prototype

Java函数的类名称一className

用来加载该Java类的C1assLoader一classLoader

然后通过以下步骤来获取Java函数所属的Class以及它在该Class的位置Slot

Step 1:

获得Class对象

Class<?> clazz=classLoader.loadClass(className);

Step 2:

获得Method对象

Method method=clazz.getDeclaredMethod(methodName,prototype);

Step 3:

获得clazz的Slot域描述

Field field=clazz.getDeclaredField("Slot");

Step 4:

获得method的slot

int slot=field.getInt(method);

Step 5:

将clazz和slot通过JNI传递到C/C++层，调用dvmDecodeIndirectRef和

dvmSlotToMethod

ClassObject\* declaredClass=(ClassObject\*)dvmDecodeIndirectRef (dvmThreadelfU， clazz);

Method\* method=dvmSlotToMethod(declaredClass, slot);

得到一个Java函数在Dalvik虚拟机内部所对应的Method结构体之后，就可以将

它设置为Native函数:

SETes METHOD一LAG(method, ACCes NATIVE);

method->nativeFunc=&xposedCallHandler;

method->insns=(const u2\*) hookInfo;

method->registersSize=method->insSize;

method->outsSize=0;

这样，我们就将完成了Java函数的拦截，函数最终执行的是

Java- method\_ call\_ intercepto:函数，但是也可以通过jni机制调用lava函数。

由此，我们便可以借助ptrace函数将so注入到目标进程，so完成目标进程目标函数method结构体的获取，并且将Java函数修改成native方法。

7.24

**概述**：

这两周的进展包括：

1.学习了android的权限机制，实现具有简单动态权限申请的app。

2.掌握LINUX系统上的基本操作。

3.实现了基于LINUX的hook技术。

**详解：**

一.

Android系统采用了权限机制来限制应用程序所具有的权限。为了预防应用任意

使用权限访问与应用不相关的数据对用户以及系统造成不良影响，以及防止恶意应用使用敏感权限获取用户隐私等，Android系统为应用程序分配了有限的访问权限。应用程序如果要访问一些敏感权限，必须在Android Manifest.xml文件中进行主动申请，而且在运行的时候系统还会对其申请的权限进行检查。应用在安装的时候，系统会向用户展示应用所申请的权限，如果用户同意授权便进行安装。

Android系统权限分为几个保护级别。需要了解的两个最重要保护级别是正常权限和危险权限:

（1）正常权限:涵盖应用需要访问其沙盒外部数据或资源，但对用户隐私或其他应用操作风险很小的区域。这些权限在应用安装时授予，运行时不再询问用户。

（2）危险权限:涵盖应用需要涉及用户隐私信息的数据或资源，或者可能对用户存储的数据或其他应用的操作产生影响的区域。

Android系统对所有的危险权限进行了分组，称为权限组 。属于同一组的危险权限将自动合并授予，用户授予应用某个权限组的权限，则应用将获得该权限组下的所有权限（前提是相关权限在 AndroidManifest.xml 中有声明）。

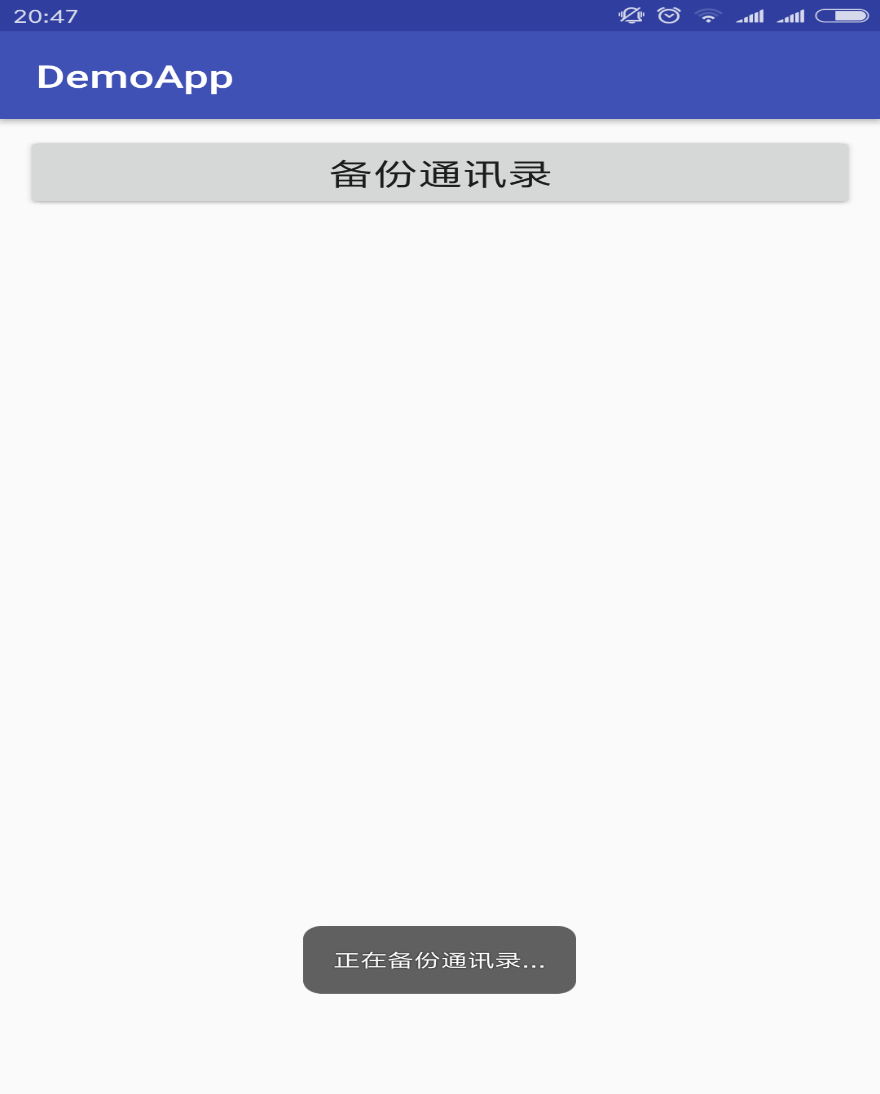
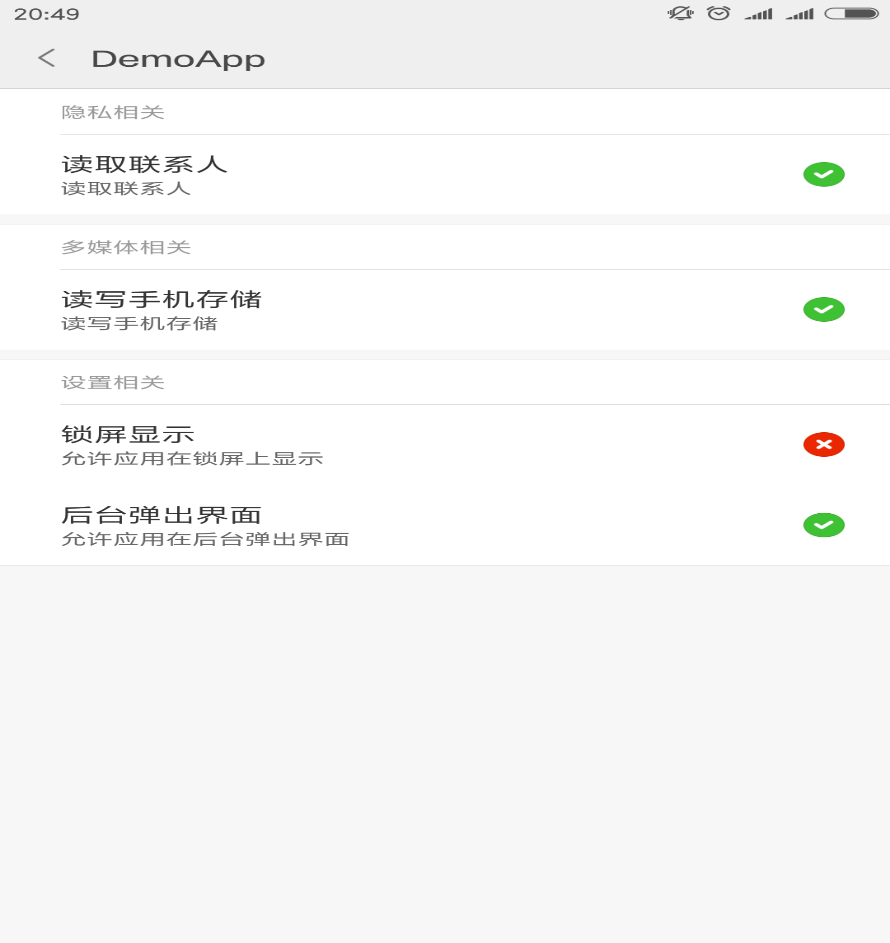
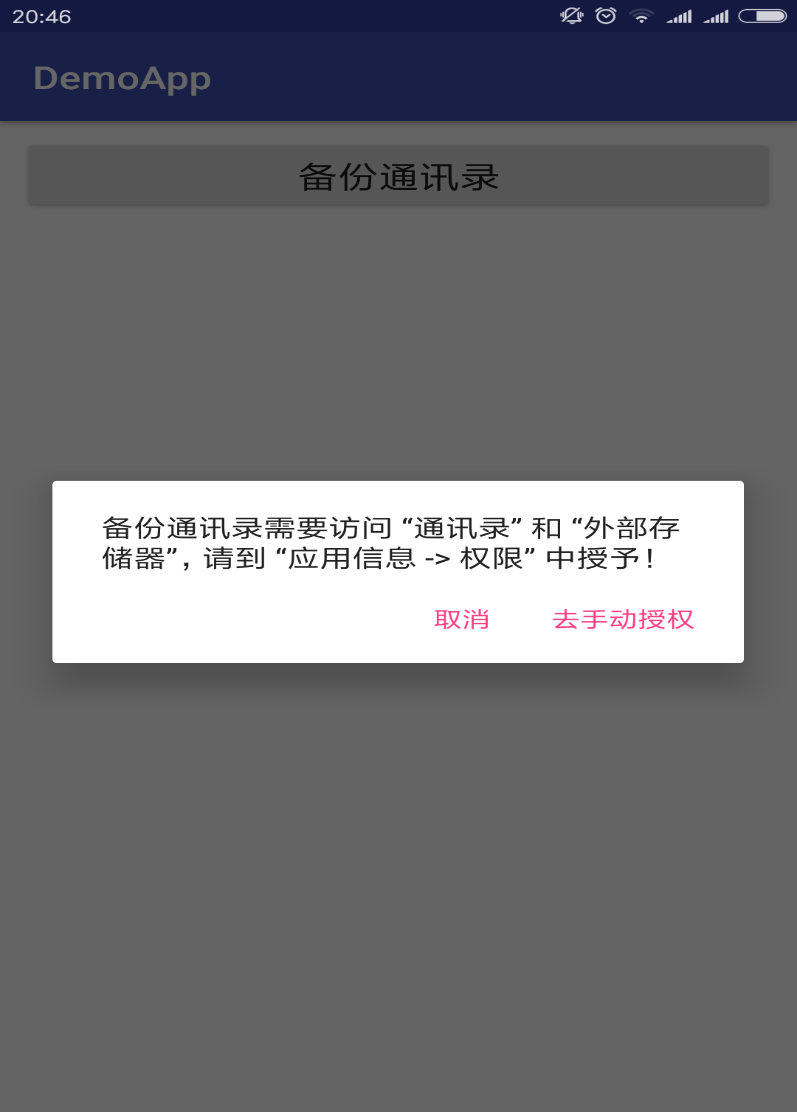
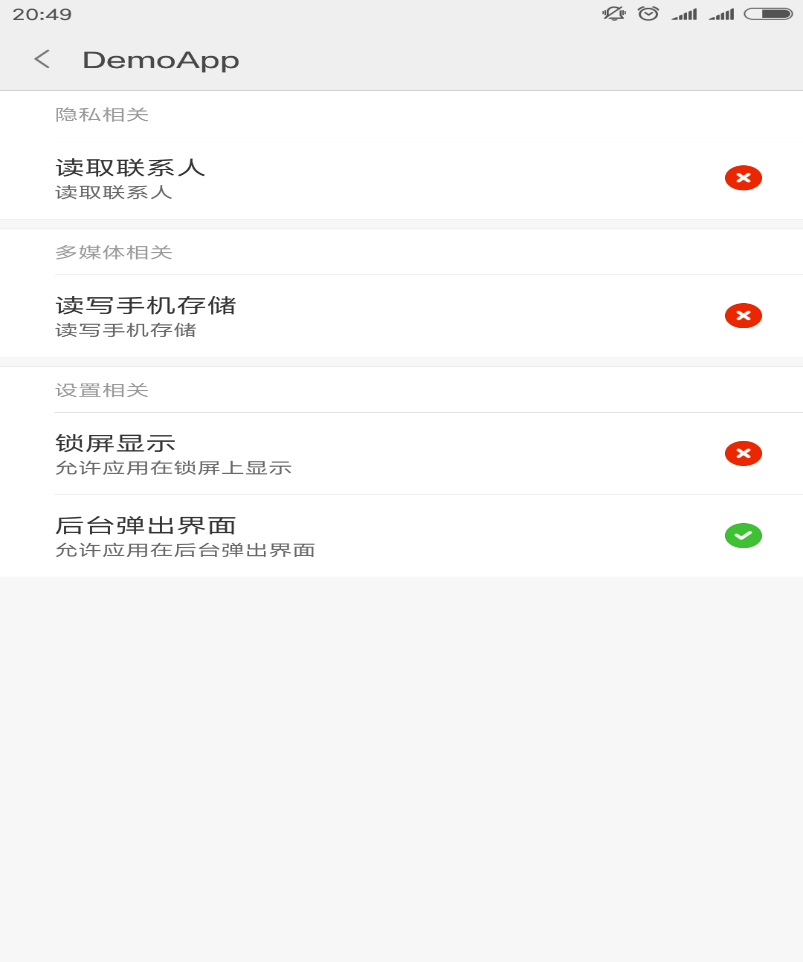
（1）点击一个按钮，如果有相应权限就进行备份通讯录操作；

（2）如果没有相应的权限，则向用户申请权限；

（3）如果用户授权通过，则继续进行备份通讯录操作；

（4）如果用户拒绝授权，则弹出对话框引导用户跳转到应用权限管理界面手动授权。

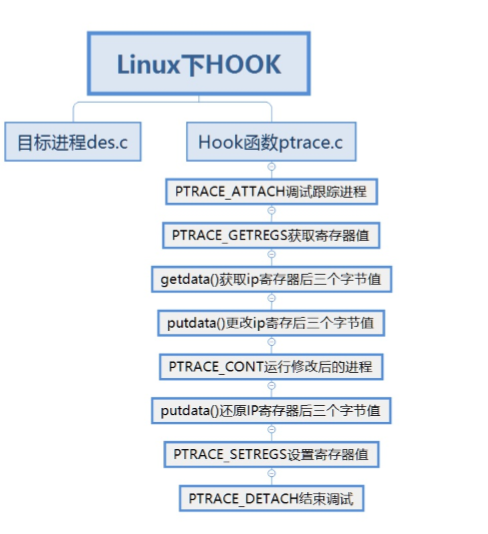
效果图如下：



源代码见附件。

二.linux hook的学习：

由于Android系统与Linux系统具有较强的联系，相应的实现也十分类似，故首先实现Linux下的Hook技术用做前期练习。在Linux系统中利用Ptrace()函数可以很轻松的对相应的进程进行修改和调试。本次实现的大致流程如本次实现的大致流程如图:



Ptrace()函数介绍

1.函数原型:

long ptrace(enum \_\_ptrace\_request request,pid\_t pid, void \*addr,void \*data);

ptrace 提供了一种父进程可以控制子进程运行，并可以检查和改变它的核心image。它主要用于实现断点 调试。一个被跟踪的进程运行中，直到发生一个信号。则进程被中止，并且通知其父进程。在进程中止的状态下，进程的内存空间可以被读写。父进程还可以使子进程继续执行，并选择是否是否忽略引起中止的

信号。

(1). 第一个参数决定了ptrace的行为，其值有：

a). ptrace\_attach: 跟踪指定 pid 进程。pid表示被跟踪进程。被跟踪进程将成为当前进程的子进程，并进 入中止状态

b). ptrace\_getregs: 读取寄存器值，pid表示被跟踪的子进程，data为用户变量地址用于返回读到的数 据。此功能将读取所有 17个基本寄存器的值。

c). ptrace\_setregs: 设置寄存器值，pid表示被跟踪的子进程，data为用户数据地址。此功能将设置所有 17个基本寄存器的值。

d). ptrace\_detach: 结束跟踪。 pid表示被跟踪的子进程。结束跟踪后被跟踪进程将继续执行.

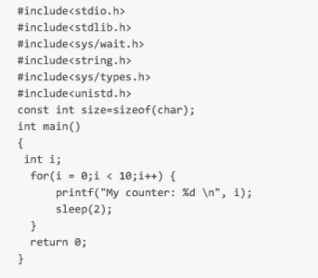
e). ptrace\_cont: 继续执行。pid表示被跟踪的子进程，signal为 0则忽略引起调试进程中止的信号，若不 为0则继续处理信号 signal。

(2). 第二个参数决定了ptrace所要操作的进程，第三个参数指定了操作数据的地址，第四个参数指定了数据的长度.

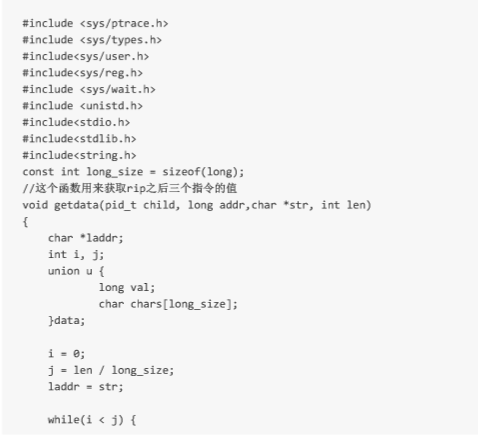
三.应用ptrace函数

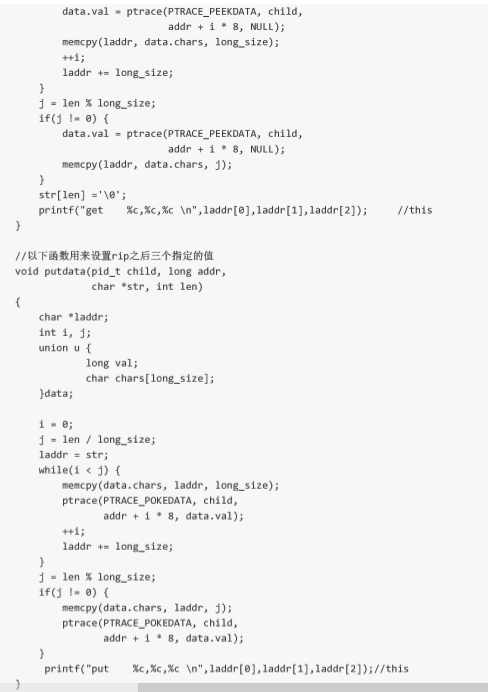
利用ptrace函数将目标函数暂停。

1.目标函数进程des.c



2.ptrace利用函数ptrace.c







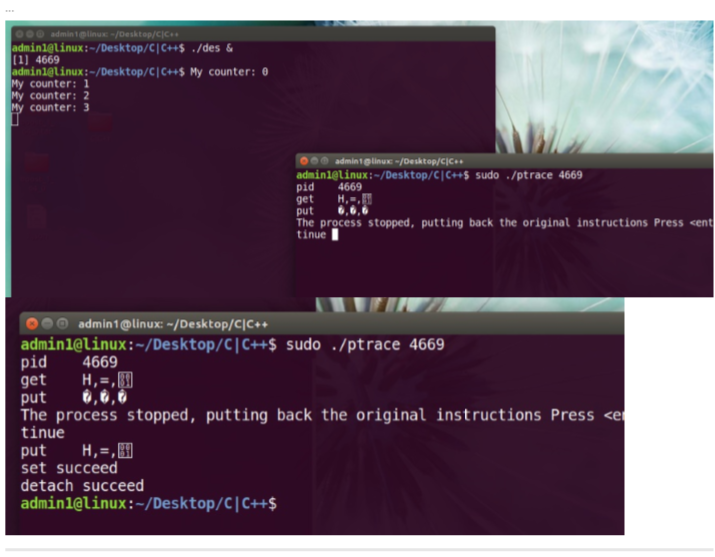
return 0

}

3.测试

gcc -o des des.c gcc -o ptrace ptrace.c

在终端中输入 ./des & 使得des在后台运行并且返回相应的pid，再在终端中输入sudo ./ptrace pid(des的pid)即可发现ptrace函数成功 修改了des函数的代码逻辑。在未运行ptrace函数时des函数会顺利运行直至结束，但是运行ptrace函数后des函数会发生中断。至此结束ptrace函数的学习.

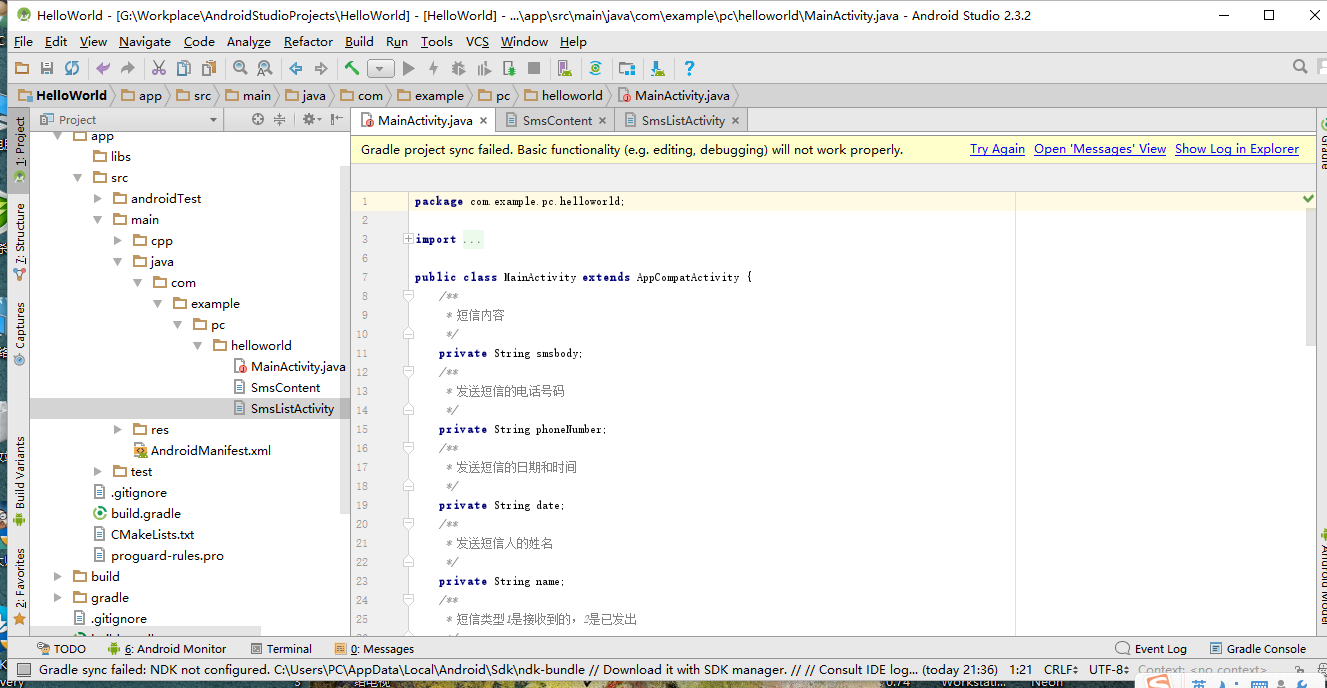


6-7

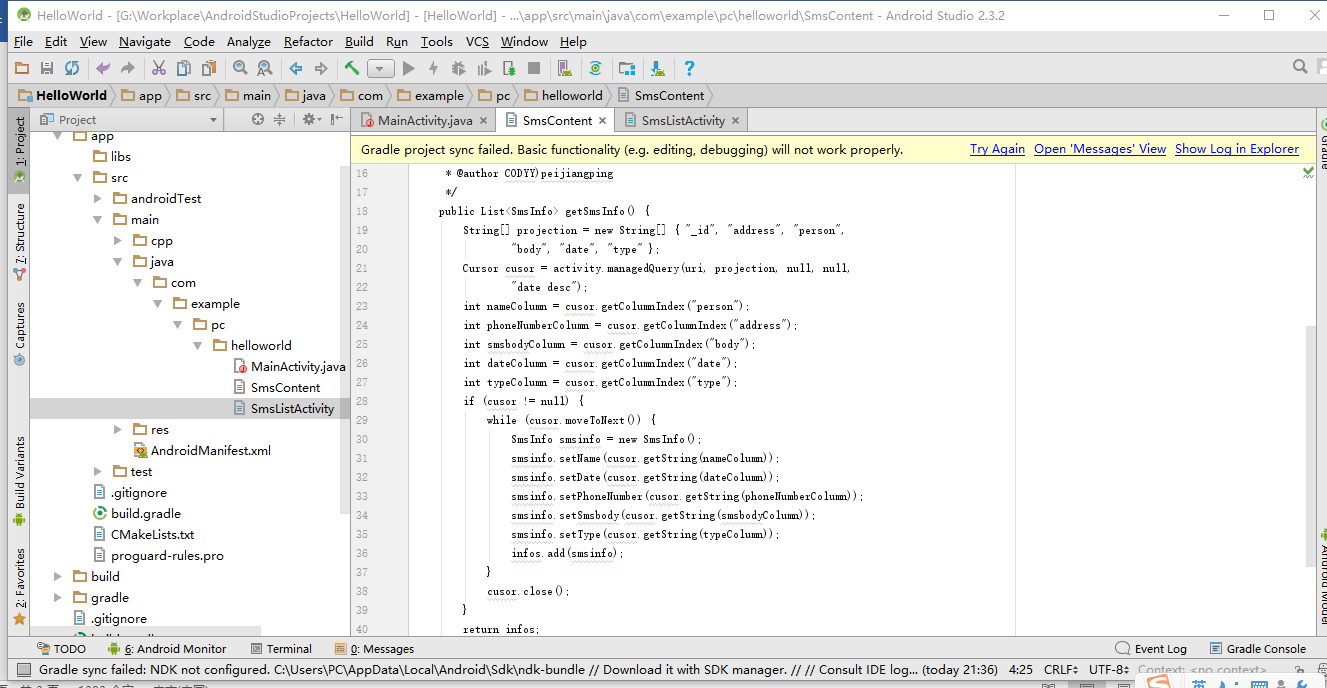
王振兴：过去两周我首先熟悉和上手了Android studio，包括一个新建工程项目的Product目录结构，怎样调试编译好的程序，及Genymotion模拟器的使用等。

为了熟悉操作，最近正在做一个读取短信信息的程序，一下是部分代码：

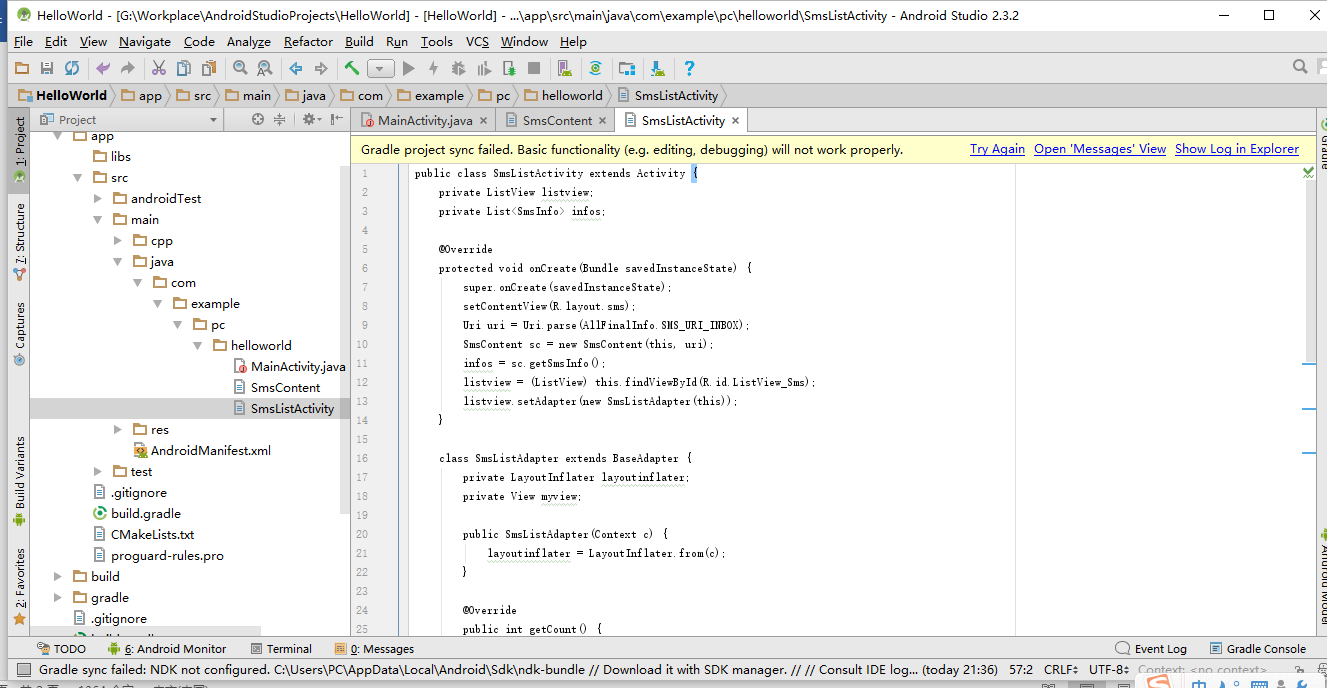
主程序：



读取信息内容的类：



显示短信内容的类：



由于这部分基础较少，有时间我就会多练手，尽量赶上进度。

同时，对于手机应用程序行为的监控，我看了一篇讲行为拦截实现方式的博客：

<http://blog.csdn.net/jiangwei0910410003/article/details/39346151>

大体内容是如何拦截可以获取系统信息的进程（主要是system\_server），如何拦截哪些应用在使用系统服务，以及如何将其拦截到的信息传递到上层然后进行显示的三种方式。

李宗泰：我看了队友的报告，大多是不懂的。太多的知识需要学习。上次布置关于短信权限的任务后，我先去看了一下Android是如何写一个首发短信的机制。发现收发短信需要在后台启动service，就回头学习了一下service。如果想要获取发送短信成功和对方收到短信的消息，就涉及到broadcast Receiver的监听功能，就又了解了一下这个组件。Broadcast和service实现细节不再赘述。Android在程序方面实现短信收发和读取功能是比较简单的，因为他已经帮我们集成了类SmsManager，准备好了好用的函数，我们只需要了解形参然后加以应用，但我们还不知道他内部是怎么实现的，这一点值得警惕，后期可以重新关注。有一个比较好玩的地方是函数sendtextmessage（）的形参pendingintent，该函数用于发送短信。pendingIntent是一种特殊的Intent，主要的区别在于，Intent的执行立刻的，而pendingIntent的执行不是立刻的。pendingIntent执行的操作实质上是参数传进来的Intent的操作，有意思的是使用pendingIntent的目的在于它所包含的Intent的操作的执行是需要满足某些条件的。还有一个类sharedpreference值得关注，他在Android中被用来记录应用，以及用户喜好，主要是可以用来保存简单的数据类型。随着我的了解，我发现不论干什么，都需要在manifest中注册相应的权限，不注册是不能实现相应功能的。我想你要读取短信，就必须注册短信权限，那么问题就在于如何读取manifest中注册了什么权限（个人想法）。现在就不知道如何读取权限，然后最近有3门考试，进度就停在这里。希望老师给予指导。

盛可俊：基于API HOOK来实现检测应用查看短信

安全相关软件经常要实现的功能有API HOOK 和APP Control，为了实现这两个功能，最常用的方法就是写driver，在kernel中拦截检查相应的调用。

例如，手机平台中很多应用都会发短信，如果阻止某些应用发短信而允许另一些应用发短信。而发短信的操作并不是由每个应用直接调用的，它们把发送请求发给一个叫SmsService的服务进程，由这个服务进程再调用系统API来发短信。当我们在Kernel里面拦截到这个API的时候，发现调用者都是SmsService，无法区分原始请求者，因而无法做到有目的的拦截。

为了解决上述等问题，需要HOOK SmsService。这时候需要利用到Ptrace,许多安全软件都是通过Ptrace来实现API HOOK和APP Control。

实现过程分为以下四步：

1)用ptrace函数attach上目标进程；

2)让目标进程的执行流程跳转到mmap函数来分配一小段内存空间；

3)把一段机器码拷贝到目标进程中刚分配的内存中去；

（我们可在这里实现添加代码提醒用户有软件正在调用查看短信接口API从而实现监测）

4)最后让目标进程的执行流程跳转到注入的代码执行。

（这里可以设置相应的MESSAGE提醒用户允许还是拒绝，如果允许，则子进程继续运行，如果拒绝则直接退出，将子进程杀死）。

Ptrace:

ptrace系统调用从名字上看是用于进程跟踪的，它提供了父进程可以观察和控制其子进程执行的能力，并允许父进程检查和替换子进程的内核镜像(包括寄存器)的值。其基本原理是: 当使用了ptrace跟踪后，所有发送给被跟踪的子进程的信号(除了SIGKILL)，都会被转发给父进程，而子进程则会被阻塞，这时子进程的状态就会被系统标注为TASK\_TRACED。而父进程收到信号后，就可以对停止下来的子进程进行检查和修改，然后让子进程继续运行。

Ptrace函数具体介绍见：

http://blog.csdn.net/myarrow/article/details/9617673

http://www.cnblogs.com/tangr206/articles/3094358.html

http://blog.csdn.net/dajian790626/article/details/7781709

那么Partce目标进程的具体代码是：

ptrace(PTRACE\_ATTACH, pid, NULL, 0 );

在继续操作前，需要先把目标进程的寄存器先保存起来，这样完成注入后，恢复目标进程的寄存器，目标进程就能不受影响继续执行了。

structpt\_regs old\_regs;

ptrace(PTRACE\_GETREGS, pid, NULL, &old\_regs);

以上是这几天的收获，具体了解了如何实现软件的功能，以及第一步的相应操作。

刘翰鸣：我学习到了一些android平台检测手机权限的工具和接口，针对安卓平台的漏洞进行分析，通过调用接口实现app手机权限扫描，对android平台提供的接口实现精确分析，利用漏洞检测软件接口的恶意代码，如读取联系人SD卡信息。在Android平台中，Intent是应用程序之间进行通信的最常用的方式之一。目前，我用Intent Sniffer工具实现监控运行时路由的广播Intent，在系统上的应用程序之间发送的Intent。它不监控显式广播的Intent，而是默认为无优先权的广播。应用中有一个选项可以查看最近任务的Intent（GET\_TASKS），当用来启动Activity的Intent是可以让如Intent Sniffer这样具有GET\_TASKS权限的应用程序访问时。这个工具也能够针对那些基于应用反射和动态审查安装程序的Intent来动态升级扫描的Action和Category。我用的另一个工具Intent Fuzzer是一个针对Intent的Fuzzer。我用这个工具可以发现能够导致系统崩溃的bug，以及设备、应用程序或者是定制平台的运行中的问题。这个工具能够针对一个简单组件或者是所有安装组件进行fuzz测试。它也适用于BroadcastReceiver，但针对Service只有较少的覆盖，Service通常更加广泛地应用Binder接口而不是针对IPC的Intent。该工具只能针对一个Activity进行fuzz测试，一次不能针对所有的Activity进行测试。另外，也能应用这个接口来启动Instrumentation，虽然列出了ContentProvider，但是它们不是一个基于Intent的IPC机制，因此并不能应用该工具进行fuzz测试。