这篇文章主要讲解如何利用动态代理技术Hook掉系统的AMS服务，来实现拦截Activity的启动流程，这种hook原理方式来自**DroidPlugin**。代码量不是很多，为了更容易的理解，需要掌握JAVA的反射，动态代理技术，以及Activity的启动流程。

一、寻找Hook点的原则

Android中主要是依靠分析系统源码类来做到的，首先我们得找到被Hook的对象，我称之为Hook点；什么样的对象比较好Hook呢？一般来说，静态变量和单例变量是相对不容易改变，是一个比较好的hook点，而普通的对象有易变的可能，每个版本都不一样，处理难度比较大。我们根据这个原则找到所谓的Hook点。

二、寻找Hook点

通常点击一个Button就开始Activity跳转了，这中间发生了什么，我们如何Hook,来实现Activity启动的拦截呢?

```
public void start(View view) {
    Intent intent = new Intent(this, OtherActivity.class);
    startActivity(intent);
}
```

我们的目的是要拦截startActivity方法，跟踪源码，发现最后启动Activity是由Instrumentation类的execStartActivity做到的。其实这个类相当于启动Activity的中间者，启动Activity中间都是由它来操作的

```
public ActivityResult execStartActivity(
            Context who, IBinder contextThread, IBinder token, Activity target,
            Intent intent, int requestCode, Bundle options) {
    IApplicationThread whoThread = (IApplicationThread) contextThread;
    ....
            try {
        intent.migrateExtraStreamToClipData();
        intent.prepareToLeaveProcess(who);
        //通过ActivityManagerNative.getDefault()获取一个对象，开始启动新的Activity
        int result = ActivityManagerNative.getDefault()
                        .startActivity(whoThread, who.getBasePackageName(),
intent,

 intent.resolveTypeIfNeeded(who.getContentResolver()),
                                token, target != null ? target.mEmbeddedID :
null,
                                requestCode, 0, null, options);
        checkStartActivityResult(result, intent);
    }
    catch (RemoteException e) {
        throw new RuntimeException("Failure from system", e);
    }
    return null;
}
```

对于ActivityManagerNative这个东东，熟悉Activity/Service启动过程的都不陌生

```
public abstract class ActivityManagerNative extends Binder implements
IActivityManager
```

继承了Binder，实现了一个IActivityManager接口，这就是为了远程服务通信做准备的"Stub"类，一个完整的AID L有两部分，一个是个跟服务端通信的Stub,一个是跟客户端通信的Proxy。ActivityManagerNative就是Stub,阅读源码发现在ActivityManagerNative 文件中还有个ActivityManagerProxy，这里就多不扯了。

```
static public IActivityManager getDefault() {
    return gDefault.get();
}
```

ActivityManagerNative.getDefault()获取的是一个IActivityManager对象，由IActivityManager去启动Activity，IActivityManager的实现类是ActivityManagerService，ActivityManagerService是在另外一个进程之中，所有Activity 启动是一个跨进程的通信的过程，所以真正启动Activity的是通过远端服务ActivityManagerService来启动的。

```
private static final Singleton<IActivityManager> gDefault = new
Singleton<IActivityManager>() {
    protected IActivityManager create() {
        IBinder b = ServiceManager.getService("activity");
        if (false) {
            Log.v("ActivityManager", "default service binder = " + b);
        }
        IActivityManager am = asInterface(b);
        if (false) {
            Log.v("ActivityManager", "default service = " + am);
        }
        return am;
    }
```

其实gDefalut借助Singleton实现的单例模式，而在内部可以看到先从ServiceManager中获取到AMS远端服务的Binder对象，然后使用asInterface方法转化成本地化对象，我们目的是拦截startActivity,所以改变IActivityManager对象可以做到这个一点，这里gDefault又是静态的，根据Hook原则，这是一个比较好的Hook点。

三、Hook掉startActivity，输出日志

我们先实现一个小需求，启动Activity的时候打印一条日志，写一个工具类HookUtil。

```
public class HookUtil {

    private Class<?> proxyActivity;

    private Context context;

    public HookUtil(Class<?> proxyActivity, Context context) {
        this.proxyActivity = proxyActivity;
        this.context = context;
    }

    public void hookAms() {

        //一路反射，直到拿到IActivityManager的对象
```

```
        try {
            Class<?> ActivityManagerNativeClss =
Class.forName("android.app.ActivityManagerNative");
            Field defaultFiled =
ActivityManagerNativeClss.getDeclaredField("gDefault");
            defaultFiled.setAccessible(true);
            Object defaultValue = defaultFiled.get(null);
            //反射SingleTon
            Class<?> SingletonClass = Class.forName("android.util.Singleton");
            Field mInstance = SingletonClass.getDeclaredField("mInstance");
            mInstance.setAccessible(true);
            //到这里已经拿到ActivityManager对象
            Object iActivityManagerObject = mInstance.get(defaultValue);


            //开始动态代理，用代理对象替换掉真实的ActivityManager，瞒天过海
            Class<?> IActivityManagerIntercept =
Class.forName("android.app.IActivityManager");

            AmsInvocationHandler handler = new
AmsInvocationHandler(iActivityManagerObject);

            Object proxy =
Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(), new
Class<?>[]{IActivityManagerIntercept}, handler);

            //现在替换掉这个对象
            mInstance.set(defaultValue, proxy);


        } catch (Exception e) {
            e.printStackTrace();
        }
    }


    private class AmsInvocationHandler implements InvocationHandler {

        private Object iActivityManagerObject;

        private AmsInvocationHandler(Object iActivityManagerObject) {
            this.iActivityManagerObject = iActivityManagerObject;
        }

        @Override
        public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {

            Log.i("HookUtil", method.getName());
            //我要在这里搞点事情
            if ("startActivity".contains(method.getName())) {
                Log.e("HookUtil","Activity已经开始启动");
                Log.e("HookUtil","小弟到此一游！！！");
            }
            return method.invoke(iActivityManagerObject, args);
        }
    }
}
```

结合注释应该很容易看懂，在Application中配置一下

```
public class MyApplication extends Application {

    @Override
    public void onCreate() {
        super.onCreate();
        HookUtil hookUtil=new HookUtil(SecondActivity.class, this);
        hookUtil.hookAms();
    }
}
```

看看执行结果：

可以看到，我们成功的Hook掉了startActivity，输出了一条日志。有了上面的基础，现在我们开始来点有用的东西，Activity不用在清单文件中注册，就可以启动起来，这个怎么搞呢？

四、无需注册，启动Activity

如下，TargetActivity没有在清单文件中注册，怎么去启动TargetActivity？

```
public void start(View view) {
    Intent intent = new Intent(this, TargetActivity.class);
    startActivity(intent);
}
```

这个思路可以是这样，上面已经拦截了启动Activity流程，在invoke中我们可以得到启动参数intent信息，那么就在这里，我们可以自己构造一个假的Activity信息的intent，这个Intent启动的Activity是在清单文件中注册的，当真正启动的时候（ActivityManagerService校验清单文件之后），用真实的Intent把代理的Intent在调换过来，然后启动即可。

首先获取真实启动参数intent信息

```
@Override
  public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable {
    if ("startActivity".contains(method.getName())) {
        //换掉
        Intent intent = null;
        int index = 0;
        for (int i = 0; i < args.length; i++) {
            Object arg = args[i];
            if (arg instanceof Intent) {
                //说明找到了startActivity的Intent参数
                intent = (Intent) args[i];
                //这个意图是不能被启动的，因为Acitivity没有在清单文件中注册
                index = i;
```

```
            }
        }
        //伪造一个代理的Intent，代理Intent启动的是proxyActivity
        Intent proxyIntent = new Intent();
        ComponentName componentName = new ComponentName(context, proxyActivity);
        proxyIntent.setComponent(componentName);
        proxyIntent.putExtra("oldIntent", intent);
        args[index] = proxyIntent;
    }
    return method.invoke(iActivityManagerObject, args);
}
```

有了上面的两个步骤,这个代理的Intent是可以通过ActivityManagerService检验的，因为我在清单文件中注册过

```
<activity android:name=".ProxyActivity" />
```

为了不启动ProxyActivity，现在我们需要找一个合适的时机，把真实的Intent换过了来，启动我们真正想启动的Activity。看过Activity的启动流程的朋友，我们都知道这个过程是由Handler发送消息来实现的，可是通过Handler处理消息的代码来看，消息的分发处理是有顺序的，下面是Handler处理消息的代码:

```
public void dispatchMessage(Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}
```

handler处理消息的时候，首先去检查是否实现了callback接口，如果有实现的话，那么会直接执行接口方法，然后才是handleMessage方法，最后才是执行重写的handleMessage方法，我们一般大部分时候都是重写了handleMessage方法,而ActivityThread主线程用的正是重写的方法，这种方法的优先级是最低的，我们完全可以实现接口来替换掉系统Handler的处理过程。

```
public void hookSystemHandler() {
    try {
        Class<?> activityThreadClass =
Class.forName("android.app.ActivityThread");
        Method currentActivityThreadMethod =
activityThreadClass.getDeclaredMethod("currentActivityThread");
        currentActivityThreadMethod.setAccessible(true);
        //获取主线程对象
        Object activityThread = currentActivityThreadMethod.invoke(null);
        //获取mH字段
        Field mH = activityThreadClass.getDeclaredField("mH");
        mH.setAccessible(true);
        //获取Handler
        Handler handler = (Handler) mH.get(activityThread);
        //获取原始的mCallBack字段
```

```
            Field mCallBack = Handler.class.getDeclaredField("mCallback");
            mCallBack.setAccessible(true);
            //这里设置了我们自己实现了接口的CallBack对象
            mCallBack.set(handler, new ActivityThreadHandlerCallback(handler)) ;
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
```

自定义Callback类

```
private class ActivityThreadHandlerCallback implements Handler.Callback {
    private Handler handler;
    private ActivityThreadHandlerCallback(Handler handler) {
        this.handler = handler;
    }
    @Override
            public Boolean handleMessage(Message msg) {
        Log.i("HookAmsUtil", "handleMessage");
        //替换之前的Intent
        if (msg.what ==100) {
            Log.i("HookAmsUtil","lauchActivity");
            handleLauchActivity(msg);
        }
        handler.handleMessage(msg);
        return true;
    }
    private void handleLauchActivity(Message msg) {
        Object obj = msg.obj;
        //ActivityClientRecord
        try{
            Field intentField = obj.getClass().getDeclaredField("intent");
            intentField.setAccessible(true);
            Intent proxyInent = (Intent) intentField.get(obj);
            Intent realIntent = proxyInent.getParcelableExtra("oldIntent");
            if (realIntent != null) {
                proxyInent.setComponent(realIntent.getComponent());
            }
        }
        catch (Exception e){
            Log.i("HookAmsUtil","lauchActivity falied");
        }
    }
}
```

最后在application中注入

```
public class MyApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();
        //这个ProxyActivity在清单文件中注册过，以后所有的Activitiy都可以用ProxyActivity
无需声明，绕过监测
        HookAmsUtil hookAmsUtil = new HookAmsUtil(ProxyActivity.class, this);
        hookAmsUtil.hookSystemHandler();
        hookAmsUtil.hookAms();
    }
}
```

执行，点击MainActivity中的按钮成功跳转到了TargetActivity。