这是什么?

这两天在看有关于动态加载Dex hook的时候看到了一个很有意思的题，不仅代码分析量大，而且发现了大量的第三方库，甚至连wp的分析思路都看不懂。从一篇wp中得知这个题目和热补丁技术有关，了解之后觉得这个方面的思路和一些hook的实现很有相似之处，于是决定更多深入一些常见的热补丁修复手段。

基于方法替换的 Hook——xposed框架

dexposed基于于有名的xposed开源项目，是阿里淘宝使用的热补丁方案

只不过这个东西比较古早了，还是delvik虚拟机环境下的产物，之后也停止维护，对art的支持并不完善，dvm时代的很多机制和现在的主流框架都不一样。

xposed前情提要

如果了解过安卓应用的启动过程，应该会对zygote对新进程的孵化有一点印象：zygote是所有安卓应用进程的母进程，由init进程创建，在zygote启动时，他会预先加载安卓框架的公共资源，给所有应用共用——当启动一个应用时，zygote不会重新初始化进程，而是通过 fork()复制一个自身的进程，将自身的内存空间共享给新启动的应用进程，这样子既减少了加载共用资源的开销，也保证了各个应用之间初始化环境一致。

dexposed基于Xposed框架，简单介绍完zygote之后，xposed的热补丁方案就很容易理解了。正如之前所说，zygote会在每个应用进程孵化时共享java运行时库，那假如说我们让app_process（启动zygote的可执行程序）加载一个XposedBridge，他就会出现在每一个安卓应用程序中，从而实现对zygote进程和Dalvik虚拟机的控制。

> XposedBridge有一个私有的Native（JNI）方法hookMethodNative，这个方法也在app_process中使用。这个函数提供一个方法对象利用Java的Reflection机制来对内置方法覆写。

当然，Xposed的实现前提是需要root，并且会影响所有的app。

关于dexposed

把大象放进冰箱需要三步，同样dexposed的热补丁修复也可以概括为三步：找到需要替换的目标方法，处理方法（将java方法转换为native方法），回调java层处理。

第一步，找到目标方法对象。

已知要修改的方法类名和方法名，用反射就能找到method（findMethodExact方法），为了方便后续的操作，将方法的属性改为可访问。但是dexposed的思路是替换调用方法的指针，只找到方法是不够的，我们需要直接操作与虚拟机内部数据结构相关的信息，方法的对象本身。通过反射获取的method只是java层面的抽象，通过一系列的java api来操作，我们需要接触更加底层的数据结构。

dalvik虚拟机中对对象的引用管理采用了直接引用和间接引用两种方式，间接引用主要是为了支持垃圾回收机制，在移动对象在内存中的位置后可以通过修改间接指向对象的中间结构指针来保证引用有效。java层传递的对象大多是间接引用，具体通过`dvmDecodeIndirectRef`函数把间接引用转换为实际的ClassObject对象。

但ClassObject也不是我们最终的目的，dalvik虚拟机中维护着一个class方法表，保存了一个包含每个方法的数组，通过ClassObject可以知道对应Method对象在class方法表中的偏移slot，最后调用该方法。

> 方法表中的方法数组是更加底层的概念。在 Java 虚拟机中，每个类都有对应的方法表，方法表是一个结构体数组，每个结构体代表一个方法，包含了该类所有的方法信息，如方法的名称、参数类型、返回值类型、方法的字节码等。
>
> 在之后的art中每个类也有对应的方法表，用来存储类中定义的方法信息，包含了各种方法的元数据。

第二步，修改非native方法为native方法。

在方法表中还有一个指向实际方法实现的指针， 按理说修改了这个指针指向我们修复过的方法就能完成hook。但是java层有严格的访问控制机制，而native层更接近底层，有更强大的权限来操作内存和控制程序的执行流程，能更方便的实现方法的劫持。

dexposed通过hookMethodNative函数将方法标记为native，并将方法表结构体中的nativeFunc指针指向dexposedCallHandler函数。在原方法被调用时，实际调用的就是dexposedCallHandler函数

```
static void com_taobao_android_dexposed_DexposedBridge_hookMethodNative(JNIEnv* env, jclass clazz, jobject reflectedMethodIndirect,
        jobject declaredClassIndirect, jint slot, jobject additionalInfoIndirect) {
    // Usage errors?
    if (declaredClassIndirect == NULL || reflectedMethodIndirect == NULL) {
        dvmThrowIllegalArgumentException("method and declaredClass must not be null");
        return;
    }

    // Find the internal representation of the method
    ClassObject* declaredClass = (ClassObject*) dvmDecodeIndirectRef(dvmThreadSelf(), declaredClassIndirect);
    Method* method = dvmSlotToMethod(declaredClass, slot);
    if (method == NULL) {
        dvmThrowNoSuchMethodError("could not get internal representation for method");
        return;
    }

    if (dexposedIsHooked(method)) {
        // already hooked
        return;
    }
                                                    初始化DexposedHookInfo信息，这些信息都是从Java层传递过来的
                                                    http://blog.csdn.net/
    // Save a copy of the original method and other hook info
    DexposedHookInfo* hookInfo = (DexposedHookInfo*) calloc(1, sizeof(DexposedHookInfo));
    memcpy(hookInfo, method, sizeof(hookInfo->originalMethodStruct));
    hookInfo->reflectedMethod = dvmDecodeIndirectRef(dvmThreadSelf(), env->NewGlobalRef(reflectedMethodIndirect));
    hookInfo->additionalInfo = dvmDecodeIndirectRef(dvmThreadSelf(), env->NewGlobalRef(additionalInfoIndirect));

    // Replace method with our own code
    SET_METHOD_FLAG(method, ACC_NATIVE);
    method->nativeFunc = &dexposedCallHandler;
    method->insns = (const u2*) hookInfo;
    method->registersSize = method->insSize;
    method->outsSize = 0;

    if (PTR_gDvmJit != NULL) {
        // reset JIT cache
        MEMBER_VAL(PTR_gDvmJit, DvmJitGlobals, codeCacheFull) = true;
    }
}
```

这里是最核心的部分了，首先将需要hook的方法设置成native的，然后指定他nativeFunc为dexposedCallHandler函数

hookMethodNative还做了一件事：保存原始方法的相关信息，存放在DexposedHookInfo结构体中，后续如果有需要想调用原始方法的逻辑，就可以依据保存的信息实现。

第三步，回调java层调用

dexposedCallHandler函数主要做了两件事：

- 获取刚刚构造的DexposedInfo信息
- 调用Java层DexposedBridge.java中的handleHookedMethod方法

handleHookedMethod就是Dexposed框架中Java层最核心的一个方法，他主要做了三件事：

- 执行需要hook之前的所有回调方法beforeMethod
- 通过invokeOriginalMethodNative执行被hook的原生方法
- 执行需要hook之后的所有回调方法afterMethod

```
// call "before method" callbacks
int beforeIdx = 0;
do {
    try {
        ((XC_MethodHook) callbacksSnapshot[beforeIdx]).beforeHookedMethod(param);
    } catch (Throwable t) {
        log(t);

        // reset result (ignoring what the unexpectedly exiting callback did)
        param.setResult(null);
        param.returnEarly = false;
        continue;
    }

    if (param.returnEarly) {
        // skip remaining "before" callbacks and corresponding "after" callbacks
        beforeIdx++;
        break;
    }
} while (++beforeIdx < callbacksLength);

// call original method if not requested otherwise
if (!param.returnEarly) {
    try {
        param.setResult(invokeOriginalMethodNative(method, originalMethodId,
                additionalInfo.parameterTypes, additionalInfo.returnType, param.thisObject, param.args));
    } catch (InvocationTargetException e) {
        param.setThrowable(e.getCause());
    }
}

// call "after method" callbacks
int afterIdx = beforeIdx - 1;
do {
    Object lastResult =  param.getResult();
    Throwable lastThrowable = param.getThrowable();

    try {
        ((XC_MethodHook) callbacksSnapshot[afterIdx]).afterHookedMethod(param);
    } catch (Throwable t) {
        DexposedBridge.log(t);
```

执行hook方法之前的一些回调方法

执行原来的方法

执行hook方法之后的一些回调方法

其中执行原生方法是可选的：如果想要完全替换，可以不调用原来的java方法。beforeMethod和afterMethod都是开发者自定义的，其中beforeMethod会默认调用replaceHookedMethod，我们通过实现它来代替原方法的调用；除此之外，beforeMethod允许开发者在原方法执行前进行一些准备工作或条件判断等操作，afterMethod允许开发者在这个阶段进行一些清理工作、结果处理或数据统计等操作。

dexposed之后——andfix

> 从AOP的角度来看，这是效率的大幅度提升。首先它不需要任何编译器的插桩或者代码改写，对正常运行不引入任何性能开销。这是AspectJ之类的框架没法比拟的优势。其次，对所改写方法的性能开销也极低（微秒级），基本可以忽略不计。这就决定了它是一个不可多得的AOP解决方案，虽然仅限Android下，而且有些限制和兼容性制约。
>
> 作者：Oasis Feng
> 链接：https://www.zhihu.com/question/31894163/answer/81365637
> 来源：知乎
> 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

dexposed的思路总的来说就是通过劫持java虚拟机中class表中的方法入口来实现对方法的修改/替换。与它的先辈xposed相比，它无需hook，**先后通过反射机制、间接引用转换、偏移访问方法表来获取方法调用指针，将该方法的类型改变为`native`并且链接到一个通用的 Native Dispatch 方法完成热补丁修复。**

然而到了android5.0，ART替换Dalvik成了唯一的运行时环境，dexposed的思路就变得困难：

- art环境中方法内联优化会将方法的代码直接嵌入调用它的方法中，这让替换方法入口的hook方式不再可行
- 后续引入了aot和jit机制，aot将dex文件预编译成机器码，但是dexposed基于dalvik的hook机制对方法的动态修改是基于字节码进行的，直接修改变得困难；jit则会导致方法入口不固定，跑着跑着入口就变了

- 除此之外还有很多很多问题，每一个安卓版本的ART变化都是天翻地覆，这使得dexposed对新安卓版本的支持愈发难以支持

但是尽管如此，还是有为dexposed续命的尝试：

https://weishu.me/2017/11/23/dexposed-on-art/

在dexposed正式开源之后，阿里巴巴后脚又开发了一个热修复工具：andfix。andfix的实现原理和dexposed及其相像，同样是通过替换方法结构体来实现hook，但是比dexposed更加彻底，后者是仅仅替换方法入口nativefunc，而前者会通过memcpy直接复制ArtMethod结构体的所有字节，确保补丁方法的结构体和原方法完全一致。

```
extern void __attribute__ ((visibility ("hidden"))) dalvik_replaceMethod(
        JNIEnv* env, jobject src, jobject dest) {
    //clazz为被替换的类
    jobject clazz = env->CallObjectMethod(dest, jClassMethod);
    //clz 为被替换的类对象
    ClassObject* clz = (ClassObject*) dvmDecodeIndirectRef_fnPtr(
            dvmThreadSelf_fnPtr(), clazz);
    //将类状态设置为装载完毕
    clz->status = CLASS_INITIALIZED;
    //得到指向新方法的指针
    Method* meth = (Method*) env->FromReflectedMethod(src);
    //得到指向需要修复的目标方法的指针
    Method* target = (Method*) env->FromReflectedMethod(dest);

    //新方法指向目标方法，实现方法的替换
    meth->clazz = target->clazz;
    meth->accessFlags |= ACC_PUBLIC;
    meth->methodIndex = target->methodIndex;
    meth->jniArgInfo = target->jniArgInfo;
    meth->registersSize = target->registersSize;
    meth->outsSize = target->outsSize;
    meth->insSize = target->insSize;
    meth->prototype = target->prototype;
    meth->insns = target->insns;
    meth->nativeFunc = target->nativeFunc;
}
```

更值得一提的是，andfix支持art，在art中它会修改对应的ArtMethod结构体和entry_point_from_quick_compiled_code入口指针。

相比dexposed，andifx专注于紧急bug修复，更加轻便，而dexposed则着重于线上调试，比如方法埋点、逆向分析、方法前后调整。

但是作为native解决方案，始终存在着稳定性与art兼容性问题。dexposed和andfix上一次更新都已经是10年前的事了。

从instantrun到robust

从instantRun开始说起

如果说dexposed是Native流派的，InstantRun和接下来要说的robust就是Java流派。

InstantRun是android studio 2.0 stable集成的即时编译技术。可以依据代码修改的程度和类型进行合适的模式对应用进行更新，加快了修改代码后的部署时间和开发效率。在对代码修改程度小时，甚至可以立即生效，不用重启Activity或者应用。

InstantRun是通过插桩技术实现的。

> 关于插桩技术，此事在csapp中亦有提及。
>
> 如果你读过csapp的链接部分应该有一点印象，其中提到的库打桩技术就是一种插桩。库打桩可以通过创建一个和目标函数一模一样的包装函数，来欺骗系统调用包装函数而不是目标函数

插桩的概念则更为广泛，通过针对编译后字节码插入额外的代码逻辑（让我想起了frida inline的实现原理，同样也是在字节码的汇编层面更改对代码实现劫持）来添加额外的功能，比如统计代码执行次数和执行时间，还有实现应用的热修复和动态更新。

和前文的替换方法的思路相比较，如果我们预先在代码中设置判断：如果没有更改代码片段，则执行原逻辑；如果更改了代码，则执行我们的自定义逻辑，这样子是不是就不用担心方法入口的种种问题了？

由此我们可以提出两个问题：在不打断程序执行的情况下，这个判断的字段如何于我们修改了代码之后被更改？进入了我们自定义逻辑的代码段，如何把我们修复过后的代码“塞”进正在执行的程序？解决了这两个问题，就能理解instantrun的核心思路了。

来看这个例子（取自深入理解Instant Run——原理篇Instant-run是Android Studio 2.0开始引入的新特性，它 - 掘金）

这是原正常编译的activity：

```
package com.example.wuyi.instantruntest;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends Activity implements OnClickListener {
    private TextView mTv;
    private int count = 0;

    public MainActivity() {
    }

    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        this.mTv = new TextView(this);
        this.mTv.setText("click me!");
        this.mTv.setOnClickListener(this);
```

```
        this.setContentView(this.mTv);
    }

    private void toBeFix() {
        Toast.makeText(this, "origin count: " + this.count, 0).show();
    }

    public void onClick(View v) {
        this.toBeFix();
        ++this.count;
    }
}
```

这是instantrun进行插桩处理过后的activity：

```
package com.example.wuyi.instantruntest;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.TextView;
import android.widget.Toast;
import com.android.tools.ir.runtime.IncrementalChange;
import com.android.tools.ir.runtime.InstantReloadException;

public class MainActivity extends Activity implements OnClickListener {
    private TextView mTv;
    private int count;
    public static final long serialVersionUID = -3671979505056694483L;
    public static volatile transient com.android.tools.ir.runtime.IncrementalChange $change;

    public MainActivity() {
        IncrementalChange var1 = $change;
        if (var1 != null) {
            Object[] var10001 = (Object[])var1.access$dispatch("init$args.
([Lcom/example/wuyi/instantruntest/MainActivity;[Ljava/lang/Object;)Ljava/lang/Object;", new
Object[]{null, new Object[0]});
            Object[] var2 = (Object[])var10001[0];
            this(var10001, (InstantReloadException)null);
            var2[0] = this;
            var1.access$dispatch("init$body.(Lcom/example/wuyi/instantruntest/MainActivity;
[Ljava/lang/Object;)V", var2);
        } else {
            super();
            this.count = 0;
        }
    }

    public void onCreate(Bundle savedInstanceState) {
        IncrementalChange var2 = $change;
        if (var2 != null) {
```

```
            var2.access$dispatch("onCreate.(Landroid/os/Bundle;)V", new Object[]{this,
savedInstanceState});
        } else {
            super.onCreate(savedInstanceState);
            this.mTv = new TextView(this);
            this.mTv.setText("click me!");
            this.mTv.setOnClickListener(this);
            this.setContentView(this.mTv);
        }
    }

    private void toBeFix() {
        IncrementalChange var1 = $change;
        if (var1 != null) {
            var1.access$dispatch("toBeFix.()V", new Object[]{this});
        } else {
            Toast.makeText(this, "origin count: " + this.count, 0).show();
        }
    }

    public void onClick(View v) {
        IncrementalChange var2 = $change;
        if (var2 != null) {
            var2.access$dispatch("onClick.(Landroid/view/View;)V", new Object[]{this, v});
        } else {
            this.toBeFix();
            ++this.count;
        }
    }

    MainActivity(Object[] var1, InstantReloadException var2) {
        String var3 = (String)var1[1];
        switch(var3.hashCode()) {
        case -1230767868:
            super();
            return;
        case -669279916:
            this();
            return;
        default:
            throw new InstantReloadException(String.format("String switch could not find
'%s' with hashcode %s in %s", var3, var3.hashCode(),
"com/example/wuyi/instantruntest/MainActivity"));
        }
    }
}
```

我们可以发现这些变化：

- 在MainActivity类的开头声明了一个com.android.tools.ir.runtime.IncrementalChange类型的$change变量（这个字段是插件gradle添加的）
- 在之后的原方法中都会有一个判断$change是否为空，如果为空则执行原始代码，如果不为空则调用access$dispatch，执行补丁类的代码

$change是由instantrun工具自动生成的，access$dispatch是他唯一的方法。

我们提出的问题可以转换为：$change是如何在我们改动代码之后被修改的?

首先，$change字段的更新是通过反射和动态加载完成的。在修改代码并apple changes后，首先由gradle插件识别被修改的类和方法，生成包含新实现的代理类。

> 代理类就是在原始类名后添加$override后缀，比如MainActivity$override。说白了`MainActivity$override`基本上就是MainActivity的副本，主要改动的地方就是`toBeFix`方法中的文案，同时没有为了支持补丁机制的插桩代码。

代理类会被编译打包成dex文件，再和资源补丁一起打包位zip文件，并由android studio通过adb推送到应用数据目录（如/data/data/com.example.app/files/instant-run/）

在接收到补丁之后，潜伏在应用中的InstantRunRuntime就会在补丁文件路径下检测到补丁文件，创建一个DexClassLoader加载补丁Dex，并在之后通过反射获取到原始类的$change字段，创建好补丁类（就是之前提到的代理类）实例之后将其赋值给$change字段。

其实以上对应的是instantrun的热启动方式，也就是在对原始代码修改较小的时候，最理想、最快捷的补丁方式

- **热启动**：只能修改方法体，不能新增 / 删除字段、方法或类，此时不用对应用做任何变动就能跑通补丁
- **冷启动触发**：如果变更超出热启动范围，Instant Run 会触发**温启动**（重启 Activity）或**冷启动**（重新安装应用）。

相较于热启动，其他两种启动方式的区别就是在重启应用的时候新建一个自定义的ClassLoader加载所有的dex文件（包括补丁dex）。

但是他们都只能在原始代码的框架下缝缝补补，不能新增或删除类、字段、方法签名，在AS4.0版本instantRun也被Apply Changes替代。

关于robust

了解了instantRun，理解robust就很简单了。robust和instantRun一样，也是通过代码插桩和动态加载实现的热补丁。

首先，robust也是通过一个类似于$change的字段来判断有没有修复补丁的——changeQuickRedirect

```
public class MainActivity extends AppCompatActivity {
    public static ChangeQuickRedirect changeQuickRedirect;
    public void onBackPressed() {
        if (PatchProxy.isSupport(new Object[0], this, changeQuickRedirect, false, 1, new
Class[0], Void.TYPE)) {
            PatchProxy.accessDispatch(new Object[0], this, changeQuickRedirect, false, 1,
new Class[0], Void.TYPE);
        } else {
            super.onBackPressed();
            finish();
        }
    }
}
```

这是截取了一个robust补丁应用的一部分，编译阶段会在类中插桩一个changeQuickRedirect字段，然后在每个方法中插入补丁检查逻辑查看changeQuickRedirect是否非空。patchproxy中封装了获取当前classname和methodname的逻辑，如果检查需要hook通过则通过accessDispatch方法执行补丁方法的具体逻辑。

对于生成加载补丁的部分，有很多方面和instantRun不同。首先，补丁类是开发人员手动编写的，而不是像instantRun通过Gradle自动生成一个代理类；其实除此之外，开发人员还需要实现一个控制类——所谓控制类，就是之前提到那个要检测非空的changeQuickRedirect。补丁类包含了修改后方法的具体实现逻辑，而补丁控制类会检测方法是否需要热修复，并引导调用补丁类的方法

```
public class MainActivityPatchControl implements ChangeQuickRedirect {
    @Override
    public boolean isSupport(String methodSignature, Object[] args) {
        // 解析方法签名，判断是否为需要修复的方法
        String methodId = methodSignature.split(":")[3];
        return "1".equals(methodId); // 假设方法编号1对应getText()
    }

    @Override
    public Object accessDispatch(String methodSignature, Object[] args) {
        // 获取原始类实例
        MainActivity target = (MainActivity) args[args.length - 1];

        // 创建并调用补丁类方法
        return new MainActivityPatch(target).RobustPublicgetText();
    }
}
```

在调用补丁类方法的过程中，采用的是反射调用的方法，而不是直接将代理类直接创建实例赋值——robust通过给方法编号标识，再在补丁控制类中通过编号映射到具体的补丁方法。

robust的补丁包除了补丁类和补丁控制类之外，还有一个补丁包说明类，用来获取所有补丁对象：

```
public class PatchesInfoImpl implements PatchesInfo
 {
  public List getPatchedClassesInfo()
   {
     ArrayList localArrayList = new ArrayList();
     localArrayList.add(new  PatchedClassInfo("com.xxx.android.robustdemo.MainActivity",
"com.bytedance.robust.patch.MainActivityPatchControl"));
     com.meituan.robust.utils.EnhancedRobustUtils.isThrowable = false;
     return localArrayList;
   }
  }
```

接着还是由Gradle生成dex文件，由服务器（当然这里不是android studio了）推送给设备下的补丁目录中，比如/data/data/com.example.app/files/patch.jar。在应用启动时，由PatchExecutor类读取指定路径下的.jar包，通过DexClassLoader加载好补丁Dex，读取PatchInfoImp的类信息利用反射将控制类赋值给changeQuickRedirect。这一段神似instantRun，只是许多工作需要自己编写补丁执行类去做而不是gradle自动帮忙解决了。

```
public class PatchExecutor extends Thread {
    @Override
```

```
    public void run() {
        ...
        applyPatchList(patches);
        ...
    }
    /**
     * 应用补丁列表
     */
    protected void applyPatchList(List<Patch> patches) {
        ...
        for (Patch p : patches) {
            ...
            currentPatchResult = patch(context, p);
            ...
            }
    }
    protected boolean patch(Context context, Patch patch) {
        ...
        DexClassLoader classLoader = new DexClassLoader(patch.getTempPath(),
context.getCacheDir().getAbsolutePath(),
                null, PatchExecutor.class.getClassLoader());
        patch.delete(patch.getTempPath());
        ...
        try {
            patchsInfoClass =
classLoader.loadClass(patch.getPatchesInfoImplClassFullName());
            patchesInfo = (PatchesInfo) patchsInfoClass.newInstance();
            } catch (Throwable t) {
             ...
        }
        ...
        for (PatchedClassInfo patchedClassInfo : patchedClasses) {
            ...
            try {
                oldClass = classLoader.loadClass(patchedClassName.trim());
                Field[] fields = oldClass.getDeclaredFields();
                for (Field field : fields) {
                    if (TextUtils.equals(field.getType().getCanonicalName(),
ChangeQuickRedirect.class.getCanonicalName()) &&
TextUtils.equals(field.getDeclaringClass().getCanonicalName(), oldClass.getCanonicalName()))
{
                        changeQuickRedirectField = field;
                        break;
                    }
                }
                ...
                try {
                    patchClass = classLoader.loadClass(patchClassName);
                    Object patchObject = patchClass.newInstance();
                    changeQuickRedirectField.setAccessible(true);
                    changeQuickRedirectField.set(null, patchObject);
                    } catch (Throwable t) {
                    ...
```

```
                }
            } catch (Throwable t) {
                ...
            }
        }
        return true;
    }
}
```

其实相比instantRun，Robust还有一点更值得说说的是：它支持新增方法或者字段。

除了补丁类、补丁控制类、补丁说明类，robust的补丁包中还还还有一个补丁管理类，其中通过静态Map存储新增字段

```
// 补丁包中的PatchManager.java
public class PatchManager {
    // 静态Map：存储所有实例的新增字段
    private static final Map<Object, Map<String, Object>> FIELD_MAP = new HashMap<>();

    // 获取字段值
    public static Object getField(Object instance, String fieldName) {
        Map<String, Object> fields = FIELD_MAP.get(instance);
        return fields != null ? fields.get(fieldName) : null;
    }
}
```

在被应用加载后，静态map就一直存储在应用进程的内存中。当需要时，原始类会通过代理方法应用静态map

```
// 原始类（在基线APK中）
public class MainActivity {
    // 代理方法：访问新增字段
    public String getNewField() {
        return (String) PatchManager.getField(this, "newField");
    }
}
```

之前也提到，由于补丁类和补丁控制类都是手动编写的，所以自然可以直接在补丁类中实现新增方法并将其编号，再在补丁控制类中映射到具体实现。

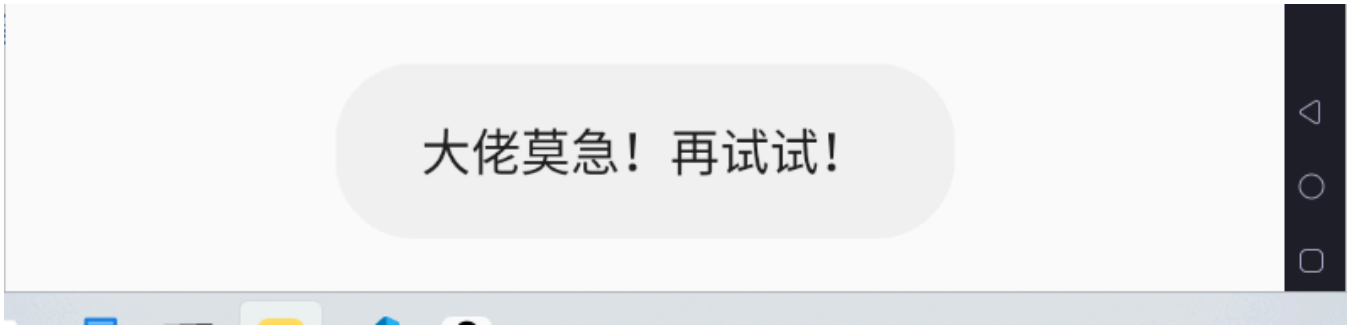
说了这么多，刚好这里有一道robust技术相关的ctf赛题！

DDCTF2018 安卓逆向 Hello Baby Dex

（其实是因为看不懂这道题才来学的补丁和hook技术原理，为了这碟醋包的饺子）

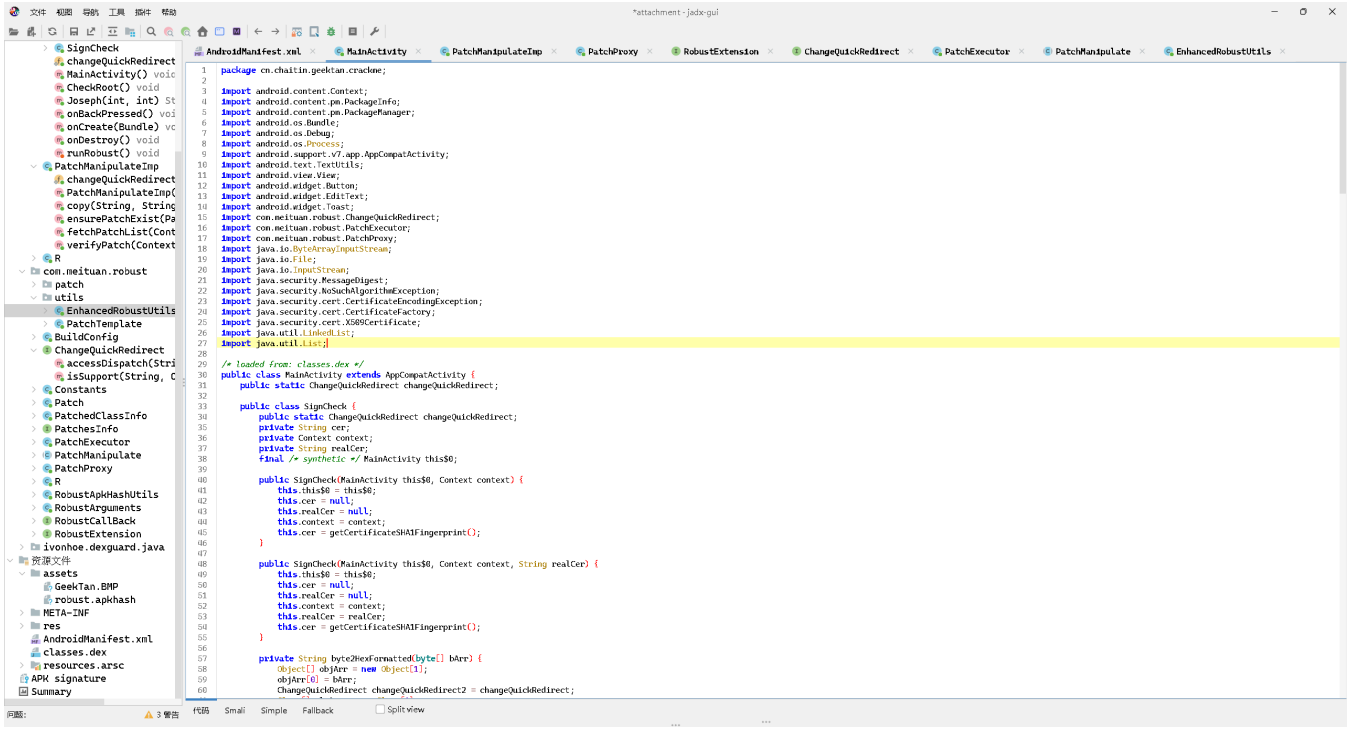
这道题给了一个apk文件，扔模拟器里打开看看


雷电模拟器
crackme
4:35
crackme
1234
确认
按键
加量
减量
全屏
截图
多开
安装
设置
更多



就是一个简单的要求输入密码的界面

用jadx打开，好家伙样子不大MainActivity还不小



看到导入中包含com.meituan.robust，类下面第一条就是public static ChangeQuickRedirect changeQuickRedirect;可以确定会有robust热补丁

程序的主要部分在于onCreate方法：

```
public void onCreate(Bundle bundle) {
        Object[] objArr = new Object[1];
        objArr[0] = bundle;
        ChangeQuickRedirect changeQuickRedirect2 = changeQuickRedirect;
        Class[] clsArr = new Class[1];
        clsArr[0] = Bundle.class;
        if (PatchProxy.isSupport(objArr, this, changeQuickRedirect2, false, 3, clsArr,
Void.TYPE)) {
            Object[] objArr2 = new Object[1];
            objArr2[0] = bundle;
            ChangeQuickRedirect changeQuickRedirect3 = changeQuickRedirect;
            Class[] clsArr2 = new Class[1];
            clsArr2[0] = Bundle.class;
            PatchProxy.accessDispatch(objArr2, this, changeQuickRedirect3, false, 3,
clsArr2, Void.TYPE);
            return;
```

```
        }
        final String Joseph = Joseph(1, 2);
        super.onCreate(bundle);
        setContentView(R.layout.activity_main);
        runRobust();
        if (!new SignCheck(this, this,
"1B:D0:4A:9D:B5:A9:84:93:7E:79:27:9C:6C:C4:14:AB:DD:B0:75:7F").check()) {
        }
        if (Debug.isDebuggerConnected()) {
        }
        final EditText editText = (EditText) findViewById(R.id.input_text);
        ((Button) findViewById(R.id.check_btn)).setOnClickListener(new
View.OnClickListener() { // from class: cn.chaitin.geektan.crackme.MainActivity.1
            public static ChangeQuickRedirect changeQuickRedirect;

            @Override // android.view.View.OnClickListener
            public void onClick(View view) {
                Object[] objArr3 = new Object[1];
                objArr3[0] = view;
                ChangeQuickRedirect changeQuickRedirect4 = changeQuickRedirect;
                Class[] clsArr3 = new Class[1];
                clsArr3[0] = View.class;
                if (!PatchProxy.isSupport(objArr3, this, changeQuickRedirect4, false, 18,
clsArr3, Void.TYPE)) {
                    if (TextUtils.isEmpty(editText.getText()) ||
!editText.getText().toString().equals("DDCTF{" + Joseph + "}")) {
                        Toast.makeText(MainActivity.this, "大佬莫急！再试试！", 0).show();
                        return;
                    } else {
                        Toast.makeText(MainActivity.this, "恭喜大佬！密码正确！", 0).show();
                        return;
                    }
                }
                Object[] objArr4 = new Object[1];
                objArr4[0] = view;
                ChangeQuickRedirect changeQuickRedirect5 = changeQuickRedirect;
                Class[] clsArr4 = new Class[1];
                clsArr4[0] = View.class;
                PatchProxy.accessDispatch(objArr4, this, changeQuickRedirect5, false, 18,
clsArr4, Void.TYPE);
            }
        });
    }
```

可以看到方法开头就有一个isSupport的判断，和我们了解到的不同的是他没有检测changeQuickRedirect是否非空，而是直接检查是否需要热补丁，判断通过再调用accessdispatch，之后执行完自定义逻辑直接返回。的确先判断changeQuickRedirect非空到了控制类中还要进行一次isSupport有些多余了，在这里PatchProxy就是实现changeQuickRedirect的补丁控制类。

具体内容；之后我们点击“确认”，onClick就会触发，在onClick中仍然会将robust热补丁的判断逻辑执行一遍，但原始逻辑是检测我们输入的内容和由Joseph组成的flag是否相同，而Joseph由Joseph()方法生成

```
public String Joseph(int i, int i2) {
    Object[] objArr = new Object[2];
    objArr[0] = new Integer(i);
    objArr[1] = new Integer(i2);
    ChangeQuickRedirect changeQuickRedirect2 = changeQuickRedirect;
    Class[] clsArr = new Class[2];
    clsArr[0] = Integer.TYPE;
    clsArr[1] = Integer.TYPE;
    if (PatchProxy.isSupport(objArr, this, changeQuickRedirect2, false, 6, clsArr, String.class)) {
        Object[] objArr2 = new Object[2];
        objArr2[0] = new Integer(i);
        objArr2[1] = new Integer(i2);
        ChangeQuickRedirect changeQuickRedirect3 = changeQuickRedirect;
        Class[] clsArr2 = new Class[2];
        clsArr2[0] = Integer.TYPE;
        clsArr2[1] = Integer.TYPE;
        return (String) PatchProxy.accessDispatch(objArr2, this, changeQuickRedirect3, false, 6, clsArr2, String.class);
    }
    List<Integer> list = new LinkedList<>();
    list.add(Integer.valueOf(R.id.ALT));
    list.add(Integer.valueOf(R.id.always));
    list.add(Integer.valueOf(R.id.basic));
    list.add(Integer.valueOf(R.id.beginning));
    list.add(Integer.valueOf(R.id.check_btn));
    list.add(Integer.valueOf(R.id.chains));
    list.add(Integer.valueOf(R.id.end));
    list.add(Integer.valueOf(R.id.useLogo));
    list.add(Integer.valueOf(R.id.CTRL));
    list.add(Integer.valueOf(R.id.never));
    list.add(Integer.valueOf(R.id.packed));
    list.add(Integer.valueOf(R.id.ifRoom));
    list.add(Integer.valueOf(R.id.input_text));
    while (list.size() > 1) {
        i = ((i + i2) - 1) % list.size();
        list.remove(i);
    }
    return String.valueOf((list.get(0).intValue() << 2) & (list.get(0).intValue() >> 1));
}
```

这是原始类中Joseph的实现，但是如果判断热补丁的逻辑通过了，onClick的执行逻辑可能就完全不同，我们得先判断是否应用了热补丁机制

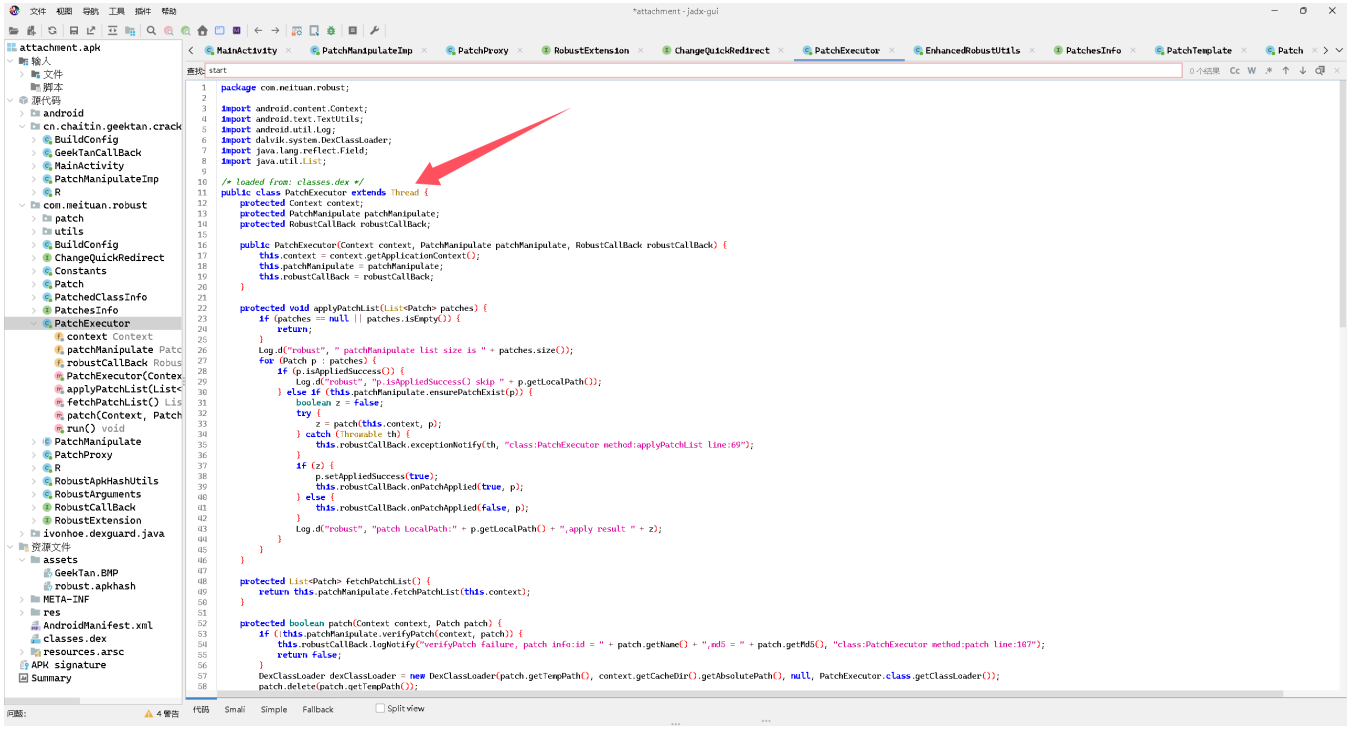
看到这里，我们回顾一下robust热补丁的总体逻辑：

负责读取补丁文件并加载补丁的是PatchExecutor类，同时也是他为每个changeQuickRedirect赋值，因此只要找到PatchExecutor就能判断出是否加载了补丁并定位出补丁文件位置

我们回头去看runRobust的具体实现：

```
147
148     private void runRobust() {
149         if (PatchProxy.isSupport(new Object[0], this, changeQuickRedirect, false, 4, new Class[0], Void.TYPE)) {
150             PatchProxy.accessDispatch(new Object[0], this, changeQuickRedirect, false, 4, new Class[0], Void.TYPE);
151         } else {
152             new PatchExecutor(getApplicationContext(), new PatchManipulateImp(), new GeekTanCallBack()).start();
153         }
154     }
155
```

同样是检测是否加载了补丁然后执行自定义逻辑，只不过如果没有检测到，就会新建一个PatchExecutor实例并调用.start()方法去加载热补丁，我们可以判断出这就是补丁机制生效的入口。进入PatchExecutor看看：



神奇的是，在PatchExecutor类中并没有start()方法，反而有一个相当具有嫌疑的run()方法，这是为什么？

```java
@Override // java.lang.Thread, java.lang.Runnable
public void run() {
    try {
        List<Patch> patches = fetchPatchList();
        applyPatchList(patches);
    } catch (Throwable t) {
        Log.e("robust", "PatchExecutor run", t);
        this.robustCallBack.exceptionNotify(t, "class:PatchExecutor,method:run,line:36");
    }
}
```

结合之后的流程分析，我觉得start()实际上就是调用了run()。可这是怎么实现的？

其实我们可以发现，PatchExecutor是继承自Thread的——这就是原因，start()和run()都在Thread中实现

- **调用`start()`**：
  - **作用**：启动一个新线程，并让 JVM 在新线程中执行`run()`方法。
  - **线程上下文**：`start()`本身在调用线程（如主线程）中执行，但`run()`在新线程中执行。
- **调用`run()`**：
  - **作用**：直接在当前线程中执行`run()`方法的代码，**不会创建新线程**。
  - **线程上下文**：`run()`和调用者在同一个线程中执行。

实际上真正的流程是这样的：

1. 主线程调用`executor.start()`。
2. JVM 创建新线程（例如`Thread-0`）。
3. 新线程开始执行`executor.run()`。
4. 主线程继续执行后续代码（直接退出了runRobust()方法）。

为什么要特意让PatchExecutor继承Thread，从而新建一个线程去执行补丁加载的过程？

> 在热修复场景中，补丁加载通常是**耗时操作**，需要在后台线程执行，避免阻塞主线程。因此：

- **使用`start()`**：补丁加载在后台线程进行，主线程继续响应 UI 交互。
- **使用`run()`**：主线程会被阻塞，导致应用无响应（ANR）。

如果忽略了这个细节就很难搞清楚补丁机制的具体调用过程。解决完了这个问题，继续跟进fetchPatchList方法

```
protected List<Patch> fetchPatchList() {
    return this.patchManipulate.fetchPatchList(this.context);
}
```

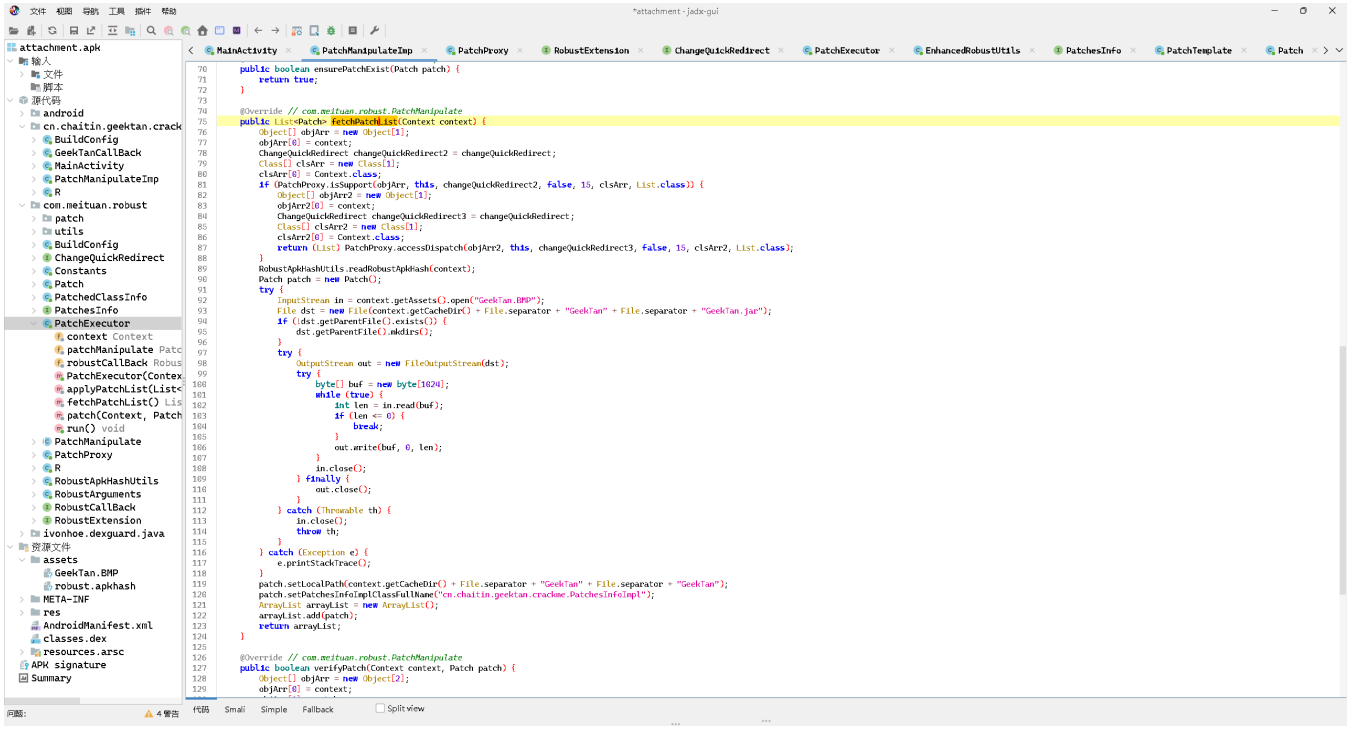
```
public class PatchExecutor extends Thread {
    protected Context context;
    protected PatchManipulate patchManipulate;
    protected RobustCallBack robustCallBack;

    public PatchExecutor(Context context, PatchManipulate patchManipulate, RobustCallBack robustCallBack) {
        this.context = context.getApplicationContext();
        this.patchManipulate = patchManipulate;
        this.robustCallBack = robustCallBack;
    }

    protected void applyPatchList(List<Patch> patches) {
        if (patches == null || patches.isEmpty()) {
            return;
        }
        Log.d("robust", " patchManipulate list size is " + patches.size());
        for (Patch p : patches) {
            if (p.isAppliedSuccess()) {
                Log.d("robust", "p.isAppliedSuccess() skip " + p.getLocalPath());
            } else if (this.patchManipulate.ensurePatchExist(p)) {
                boolean z = false;
                try {
                    z = patch(this.context, p);
                } catch (Throwable th) {
                    this.robustCallBack.exceptionNotify(th, "class:PatchExecutor method:applyPatchList line:69");
                }
                if (z) {
                    p.setAppliedSuccess(true);
                    this.robustCallBack.onPatchApplied(true, p);
                } else {
                    this.robustCallBack.onPatchApplied(false, p);
                }
                Log.d("robust", "patch LocalPath:" + p.getLocalPath() + ",apply result " + z);
            }
        }
    }

    protected List<Patch> fetchPatchList() {
        return this.patchManipulate.fetchPatchList(this.context);
    }
```

fetchPatchList会调用patchManipulate对象的同名方法，patchManipulate又是在构造方法中作为第二个参数传入的。我们回顾会发现第二个参数是一个PatchManipulateImp实例，继续跟进PatchManipulate类查看其中的fetchPatchList是怎么实现的



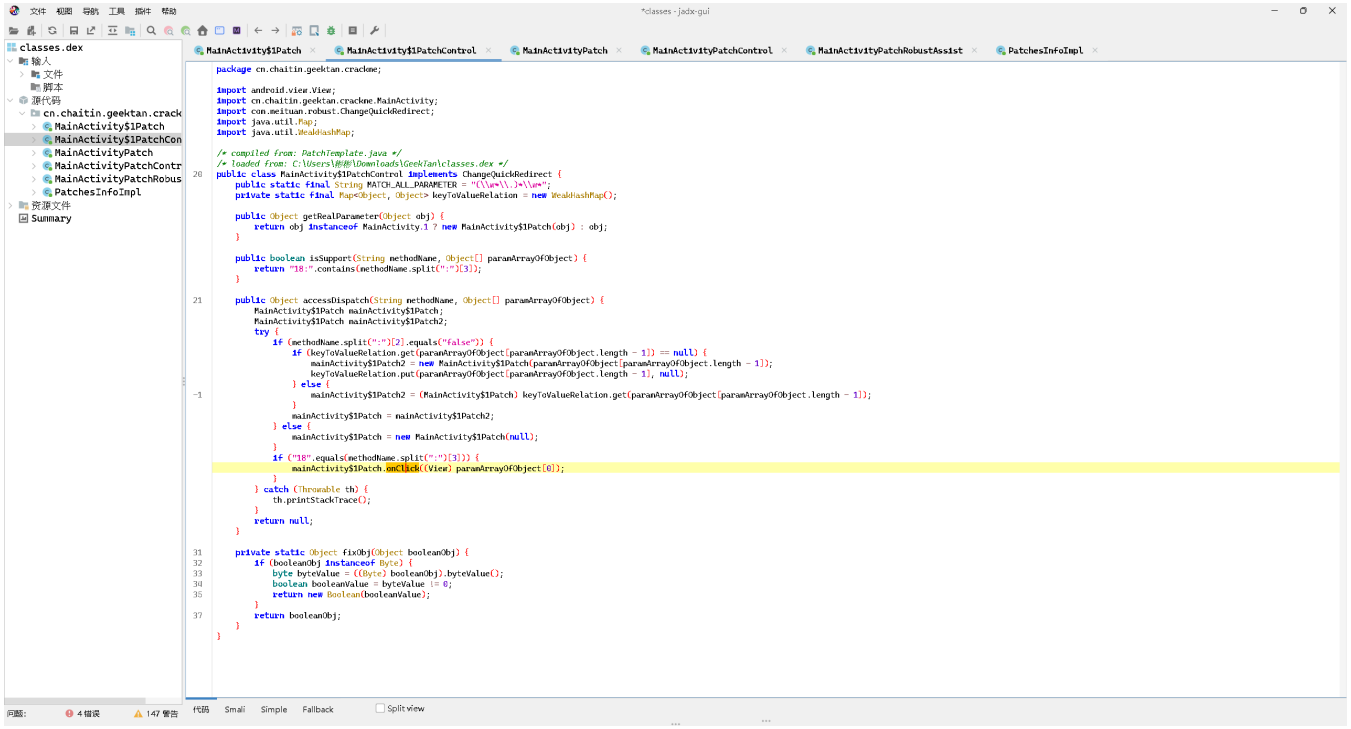
有别于之前我们对robust框架中了解到的服务器下发的方式，应用会读取一个叫做"GeekTan.BMP"的本地文件，将其更名为"GeekTan.jar"，很明显这就是补丁包文件。接着通过其中的补丁说明类文件PatchesInfoImpl读取补丁列表并返回，就完成了拉取补丁包、读取补丁列表的过程。

重新回到run()方法，在获取到补丁列表之后直接执行applyPatchList()方法

```
protected void applyPatchList(List<Patch> patches) {
    if (patches == null || patches.isEmpty()) {
        return;
    }
    Log.d("robust", " patchManipulate list size is " + patches.size());
    for (Patch p : patches) {
        if (p.isAppliedSuccess()) {
            Log.d("robust", "p.isAppliedSuccess() skip " + p.getLocalPath());
        } else if (this.patchManipulate.ensurePatchExist(p)) {
            boolean z = false;
            try {
                z = patch(this.context, p);
            } catch (Throwable th) {
                this.robustCallBack.exceptionNotify(th, "class:PatchExecutor method:applyPatchList line:69");
            }
            if (z) {
                p.setAppliedSuccess(true);
                this.robustCallBack.onPatchApplied(true, p);
            } else {
                this.robustCallBack.onPatchApplied(false, p);
            }
            Log.d("robust", "patch LocalPath:" + p.getLocalPath() + ",apply result " + z);
        }
    }
}
```

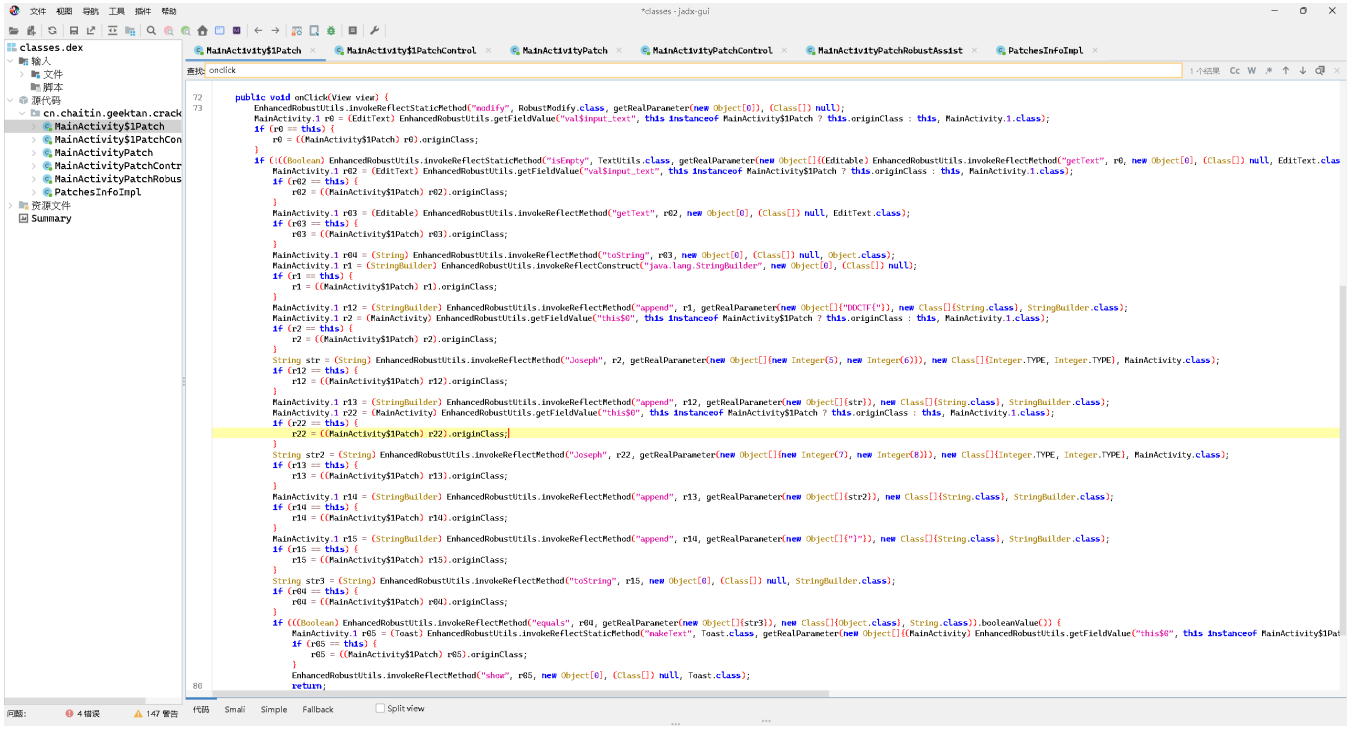
这个方法会遍历刚刚获取的补丁列表，并执行patch方法。patch方法代码段比较长，简述就是通过加载器dexclassloader将所有补丁加载到内存中，然后遍历所有的changeQuickRedirectField字段并将其赋值。

至此，我们了解了这个应用中补丁加载的全过程，之后在正常的程序流程中都会执行判断是否应用热补丁->进入补丁类的自定义逻辑的老一套。终于到了最激动人心的一段，我们看看补丁包GeekTan.BMP中具体是怎么样的



补丁包中有两套MainActivity的补丁类和补丁控制类，我们只关心onClick()的实现流程。和之前对robust框架的了解一样，补丁控制类通过方法编号来调用具体方法。在MainActivity$1PatchControl中能找到onClick()重写实现的调用流程，在MainActivityPatchControl中则有onCreate和Joseph的调用流程

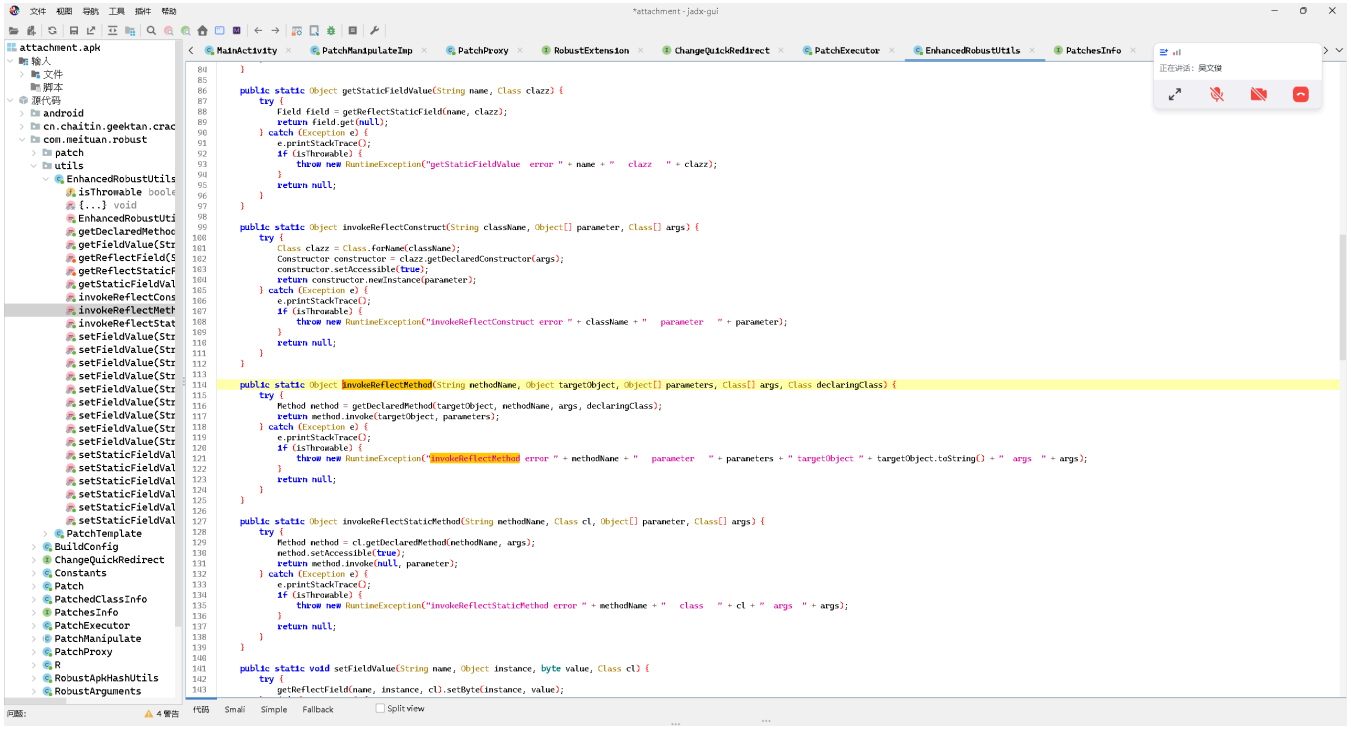
```
String str = methodName.split(":")[3];
            if ("3".equals(str)) {
                mainActivityPatch.onCreate((Bundle) paramArrayOfObject[0]);
            }
            if ("6".equals(str)) {
                return mainActivityPatch.Joseph(((Integer)
paramArrayOfObject[0]).intValue(), ((Integer) paramArrayOfObject[1]).intValue());
            }
/*另一个补丁控制类文件MainActivityPatchControl中的代码片段，其余都一样*/
```



然而打开补丁类文件，全他妈是invokeReflectMethod的对原始方法的反射调用，好家伙robust补丁类就这样重写方法实现的。但是作为一名饱经锻炼的ctf选手，眼睛里应该只能容下对解题有用的部分——粗略看一下，这个流程应该和原方法是大致相同的，可能是在Joshph的具体实现中做了手脚。我们直接看最后判断输入对错的实现

```
 if (((Boolean) EnhancedRobustUtils.invokeReflectMethod("equals", r04, getRealParameter(new
Object[]{str3}), new Class[]{Object.class}, String.class)).booleanValue()) {
                MainActivity.1 r05 = (Toast)
EnhancedRobustUtils.invokeReflectStaticMethod("makeText", Toast.class, getRealParameter(new
Object[]{(MainActivity) EnhancedRobustUtils.getFieldValue("this$0", this instanceof
MainActivity$1Patch ? this.originClass : this, MainActivity.1.class), "恭喜大佬！密码正确！ ",
new Integer(0)}), new Class[]{Context.class, CharSequence.class, Integer.TYPE});
                if (r05 == this) {
                    r05 = ((MainActivity$1Patch) r05).originClass;
                }
                EnhancedRobustUtils.invokeReflectMethod("show", r05, new Object[0],
(Class[]) null, Toast.class);
                return;
            }
        }
        MainActivity.1 r06 = (Toast)
EnhancedRobustUtils.invokeReflectStaticMethod("makeText", Toast.class, getRealParameter(new
Object[]{(MainActivity) EnhancedRobustUtils.getFieldValue("this$0", this instanceof
MainActivity$1Patch ? this.originClass : this, MainActivity.1.class), "大佬莫急！再试试！ ", new
Integer(0)}), new Class[]{Context.class, CharSequence.class, Integer.TYPE});
```

好家伙，而且这个EnhancedRobustUtils.invokeReflectMethod是在原始应用文件中实现的



这意味着我们通过frida hook的时候可以不用考虑如何hook dex动态加载类，通过拦截invokeReflectMethod过滤第一个参数为Joseph和equals就能直接一把梭

```
Java.perform(function(){
    let EnhancedRobustUtils = Java.use("com.meituan.robust.utils.EnhancedRobustUtils");
    EnhancedRobustUtils["invokeReflectMethod"].implementation = function (methodName, targetObject, parameters, args, declaringClass) {
        let result = this["invokeReflectMethod"](methodName, targetObject, parameters, args, declaringClass);
        if(methodName == "Joseph"){
            console.log(`EnhancedRobustUtils.invokeReflectMethod is called: methodName=${methodName}, targetObject=${targetObject}, parameters=${parameters}, args=${args}, declaringClass=${declaringClass}`);
            console.log(`EnhancedRobustUtils.invokeReflectMethod ${methodName} result=`+result);
        }
        if(methodName == "equals"){
            console.log(`EnhancedRobustUtils.invokeReflectMethod is called: methodName=${methodName}, targetObject=${targetObject}, parameters=${parameters}, args=${args}, declaringClass=${declaringClass}`);
            console.log(`EnhancedRobustUtils.invokeReflectMethod ${methodName} result=`+result);
        }
        return result;
    };
})
```

```
marcel@Flag-Fish-Tournament:~/android_re/baby_dex$ frida -U crackme -l _agent.js
     ____
    / _  |   Frida 15.2.2 - A world-class dynamic instrumentation toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)

[Android Emulator 5554::crackme ]-> EnhancedRobustUtils.invokeReflectMethod is cal
led: methodName=Joseph, targetObject=cn.chaitin.geektan.crackme.MainActivity@7e9cb
12, parameters=5,6, args=int,int, declaringClass=class cn.chaitin.geektan.crackme.
MainActivity
EnhancedRobustUtils.invokeReflectMethod Joseph result=29360128
EnhancedRobustUtils.invokeReflectMethod is called: methodName=Joseph, targetObject
=cn.chaitin.geektan.crackme.MainActivity@7e9cb12, parameters=7,8, args=int,int, de
claringClass=class cn.chaitin.geektan.crackme.MainActivity
EnhancedRobustUtils.invokeReflectMethod Joseph result=29362176
EnhancedRobustUtils.invokeReflectMethod is called: methodName=equals, targetObject
=1234, parameters=DDCTF{2936012829362176}, args=class java.lang.Object, declaringC
lass=class java.lang.String
EnhancedRobustUtils.invokeReflectMethod equals result=false
```

读取equals的对比字符串就能看到flag，原来重写的方法是调用了两次Joseph，将两次结果拼接得到flag，的确流程和原onClick有一点点不一样，出题人心机叵测呀

Reference

热补丁技术解析-CSDN博客

Android 中免 Root 实现 Hook 的 Dexposed 实现原理解析以及如何实现应用的热修复今天我们来看一 - 掘金

Android热补丁技术—dexposed原理简析(阿里Hao) - linghu_java - 博客园

我为 Dexposed 续一秒：论 ART 上运行时 Method AOP 实现 — 小专栏

AndFix - 热修复方案原理分析AndFix 是阿里开源的一种在线 bug 热修复的方案，当线上应用出现紧急 Bug - 掘金

Instant Run 浅析 | Jason's Blog

从Instant run谈Android替换Application和动态加载机制 | w4lle's Notes

深入理解Instant Run——原理篇Instant-run是Android Studio 2.0开始引入的新特性，它 - 掘金

Android热更新方案Robust - 美团技术团队

Android热补丁之Robust原理解析(一) | w4lle's Notes

Robust 2.0：支持Android R8的升级版热修复框架 - 美团技术团队

DDCTF 2018 Android WriteUp - SecPulse.COM | 安全脉搏

[原创]进阶Frida--Android逆向之动态加载dex Hook（三）（上篇）-Android安全-看雪-安全社区|安全招聘|kanxue.com