一、前言

这是*Anti-Unidbg*的第二篇，初步讨论 *Unidbg* 的 *JNI* 调用问题。

二、描述

Unicorn无法跑dex（至少目前没看到开源的方案），更别提跑通完整的Android FrameWork，那么当Native通过JNI这座桥梁和JAVA世界做交互时，Unidbg如何处理呢？

可以简单理解为Hook后打补丁，Unidbg预先补了一些基础的、对系统类和方法的访问，其余交由用户处理。实际过程复杂很多，详见 *unidbg-android/src/main/java/com/github/unidbg/linux/android/dvm/AbstractJni.java* ，但不影响这样一个事实

> Unidbg对进程的JAVA世界一无所知

Anti-Unidbg的一半武器，都由它贡献。（而另一半武器，则来自Unidbg对目标SO以外的Native世界缺少了解，这是后话）。比如文章一的问题，换个角度理解就是Unidbg不知道JAVA世界里添加了环境变量，所以让我们成功设置了一个陷阱。

本篇要讲什么呢——**FindClass**

```
jclass FindClass(JNIEnv *env, const char *name);
```

FindClass 用于查找类名为name的JAVA类，如果找到就返回jclass引用，没有就返回null并抛出异常。这个API有一个常见的作用——Anti 脱壳机。

比如开发者想检测YouPK（一款基于ART的主动调用的脱壳机），于是他阅读了YouPK代码，发现其中存在这么一个关键类

https://github.com/Youlor/unpacker/blob/01747abf5ce01b52a2349e7360fa7db577125129/android-7.1.2_r33/frameworks/base/core/java/andrc

```
145 import libcore.io.IoUtils;
146 import libcore.net.event.NetworkEventDispatcher;
147 import dalvik.system.CloseGuard;
148 import dalvik.system.VMDebug;
149 import dalvik.system.VMRuntime;
150 import org.apache.harmony.dalvik.ddmc.DdmVmInternal;
151
152 //patch by Youlor
153 //++++++++++++++++++++++++++++
154 import cn.youlor.Unpacker;
155 //++++++++++++++++++++++++++++
156
157 final class RemoteServiceException extends AndroidRuntimeException {
158     public RemoteServiceException(String msg) {
159         super(msg);
160     }
161 }
162
163 /**
164  * This manages the execution of the main thread in an
165  * application process, scheduling and executing activities,
166  * broadcasts, and other operations on it as the activity
167  * manager requests.
```

所以他认为，如果FindClass 查找到了这个类，就说明样本运行在YouPK脱壳机环境中。这合理吗？这很合理。

那么我们就得想怎么新瓶灌旧酒，让这个API在Anti-Unidbg上也发挥出一些作用了。

> Unidbg对进程的JAVA世界一无所知

那它怎么判断目标样本里是否存在FindClass所寻找的类呢？我想你应该也猜到了——默认有。

FIndClass是为了后续使用这个类，所以凡是找某个类，它应该是存在的。否则干嘛？想报异常崩溃嘛？

这个逻辑大部分时候是对的，代价就是我们可以利用这一点来检测Unidbg。

来看下面的代码，FindClass 查找一些不可能存在的类，并做对应的异常处理防止代码崩溃影响业务本身逻辑。

```
#include <jni.h>
#include <string>

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_findmyclass_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject mainactivity /* this */) {

    const char *result;
    jclass fake_clazz = env->FindClass("my/fake/class");
    bool exc = env->ExceptionCheck();
    if(exc){
        // 清除异常，不要让进程崩溃
        env->ExceptionClear();
    }
    if(fake_clazz == nullptr){
        result = "everything ok";
    } else{
        result = "unidbg detect";
    }
    return env->NewStringUTF(result);
}
```

代码做了三件事

- 找my/fake/class这个不存在的类
- 异常处理，让进程不崩溃，继续往下运行
- 如果FindClass 找到了类，走A逻辑，否则走B逻辑

看一下真机运行情况

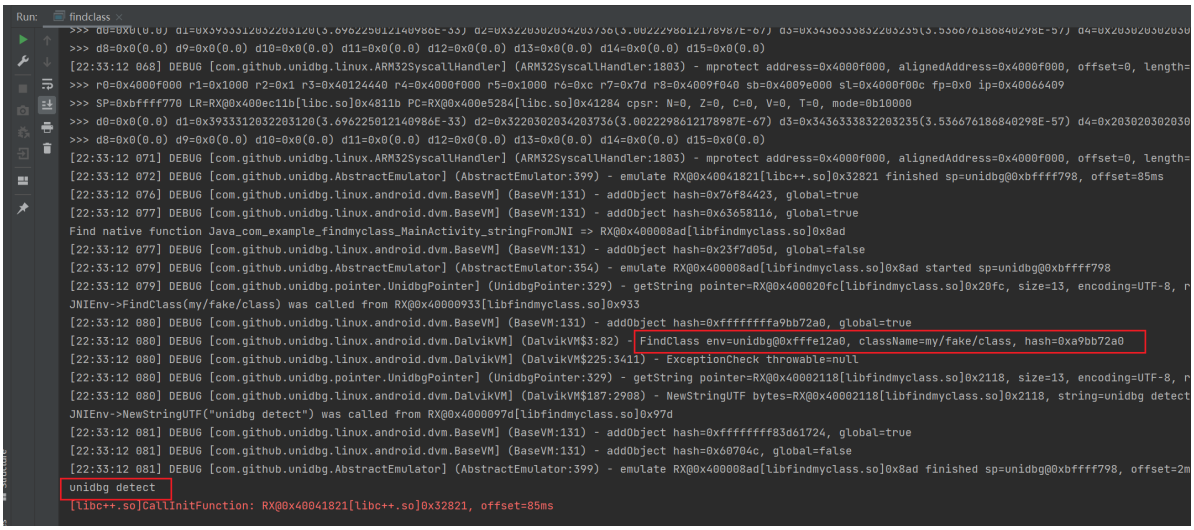
三、Unidbg模拟执行

findclass.java

```
package com.antiUnidbg;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Module;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
import com.github.unidbg.linux.android.dvm.*;
import com.github.unidbg.memory.Memory;
import com.github.unidbg.virtualmodule.android.AndroidModule;

import java.io.File;
```

```java
public class findclass {
    private final AndroidEmulator emulator;
    private final VM vm;
    private final Module module;

    findclass() {
        emulator = AndroidEmulatorBuilder
                .for32Bit()
                .build();
        final Memory memory = emulator.getMemory();
        memory.setLibraryResolver(new AndroidResolver(23));
        vm = emulator.createDalvikVM(new File("unidbg-
android/src/test/resources/findclass/app-debug.apk"));
        DalvikModule dm = vm.loadLibrary(new File("unidbg-
android/src/test/resources/findclass/libfindmyclass.so"), true);
        module = dm.getModule();
        vm.setVerbose(true); // 打印日志
    };

    public static void main(String[] args) {
        findclass demo = new findclass();
        demo.call();
    }

    public void call(){
        DvmClass dvmClass =
vm.resolveClass("com/example/findmyclass/MainActivity");
        String methodSign = "stringFromJNI()Ljava/lang/String;";
        DvmObject<?> dvmObject = dvmClass.newObject(null);

        StringObject obj = dvmObject.callJniMethodObject(emulator, methodSign);
        System.out.println(obj.getValue());
    }
}
```

运行



可以发现，我们甜蜜的陷阱已经构造成功了，成功让Unidbg和真机结果产生了差异，而且几乎可以认定，执行环境就是Unidbg。

有一些办法可以让它更隐蔽

- FindClass 查找数十个类，绝大多数都正常，只有一个类是不存在的，浑水摸鱼。
- 并不是查找一个不存在的类，而是查找一个当前时机尚未加载的类，或者当前加载器无法加载的类，这样会更有迷惑性。

四、Anti Anti-Unidbg

最朴素的办法——对目标函数做JNItrace，同时打开Unidbg JNI相关日志，耐心仔细的逐条对比，JNITrace 和 Jtrace可以完成这个任务。在Unidbg中怎么表示出“目标类不存在”这样一种状态呢？从Unidbg代码中我们可以看出，设计者其实考虑过这个问题。

```
Pointer _FindClass = svcMemory.registerSvc(new ArmSvc() {
    @Override
    public long handle(Emulator<?> emulator) {
        RegisterContext context = emulator.getContext();
        Pointer env = context.getPointerArg(0);
        Pointer className = context.getPointerArg(1);
        String name = className.getString(0);

        boolean notFound = notFoundClassSet.contains(name);
        if (verbose) {
            if (notFound) {
                System.out.printf("JNIEnv->FindNoClass(%s) was called from
%s%n", name, context.getLRPointer());
            } else {
                System.out.printf("JNIEnv->FindClass(%s) was called from %s%n",
name, context.getLRPointer());
            }
        }

        if (notFound) {
            throwable =
resolveClass("java/lang/NoClassDefFoundError").newObject(name);
            return 0;
        }

        DvmClass dvmClass = resolveClass(name);
        long hash = dvmClass.hashCode() & 0xffffffffL;
        if (log.isDebugEnabled()) {
            log.debug("FindClass env=" + env + ", className=" + name + ",
hash=0x" + Long.toHexString(hash));
        }
        return hash;
    }
});
```

使用者可以手动添加标注一个类为不存在

```
package com.antiUnidbg;

import com.github.unidbg.AndroidEmulator;
import com.github.unidbg.Module;
import com.github.unidbg.linux.android.AndroidEmulatorBuilder;
import com.github.unidbg.linux.android.AndroidResolver;
```

```java
import com.github.unidbg.linux.android.dvm.*;
import com.github.unidbg.memory.Memory;
import com.github.unidbg.virtualmodule.android.AndroidModule;

import java.io.File;

public class findclass {
    private final AndroidEmulator emulator;
    private final VM vm;
    private final Module module;

    findclass() {
        emulator = AndroidEmulatorBuilder
                .for32Bit()
                .build();
        final Memory memory = emulator.getMemory();
        memory.setLibraryResolver(new AndroidResolver(23));
        vm = emulator.createDalvikVM(new File("unidbg-
android/src/test/resources/findclass/app-debug.apk"));
        DalvikModule dm = vm.loadLibrary(new File("unidbg-
android/src/test/resources/findclass/libfindmyclass.so"), true);
        module = dm.getModule();
        vm.setVerbose(true); // 打印日志

        // 添加类到《不存在的类列表》
        vm.addNotFoundClass("my/fake/class");
    };

    public static void main(String[] args) {
        findclass demo = new findclass();
        demo.call();
    }

    public void call(){
        DvmClass dvmClass =
vm.resolveClass("com/example/findmyclass/MainActivity");
        String methodSign = "stringFromJNI()Ljava/lang/String;";
        DvmObject<?> dvmObject = dvmClass.newObject(null);

        StringObject obj = dvmObject.callJniMethodObject(emulator, methodSign);
        System.out.println(obj.getValue());
    }
}
```

一切就顺利了

五、尾声

除此之外，getMethodID，getStaticMethodID等JNI方法也可以制造同样陷阱，毕竟Unidbg也没办法知道某个类是否有某个方法，所以只能同样选择相信它**有**。