一、前言

在这篇中我们聚焦于如何利用*GetMethodID*方法构造一个隐蔽的初始化函数

二、描述

上篇我们注意到，GetMethodID()会给未初始化的类做初始化，这可以帮助我们设计出一个基于初始化缺失带来的Anti-Unidbg。

> `GetMethodID()` causes an uninitialized class to be initialized.
>
> GetMethodID()会给未初始化的类做初始化。

初始化指的是目标函数无法单独运行，需要在前面先执行某个或数个函数。这些函数会对目标函数造成某种持久化，不得不执行它们，最常见的初始化函数，在星球前面的《Unidbg初始化问题》五节视频中已经讲过了。但前面的那类初始化方式不够隐蔽，我们可以通过Frida Native Call + Hook 的方式准确的找到这些初始化函数，本文讨论如何更隐蔽的做初始化。

> GetMethodID()会给未初始化的类做初始化。

我们先理解和印证这一点，先看JAVA代码

MainActivity.java

```
package com.example.getmethodidexample1;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.widget.TextView;

import com.example.getmethodidexample1.databinding.ActivityMainBinding;

public class MainActivity extends AppCompatActivity {

    // Used to load the 'getmethodidexample1' library on application startup.
    static {
        System.loadLibrary("getmethodidexample1");
    }

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        // Example of a call to a native method
        TextView tv = binding.sampleText;
        tv.setText(stringFromJNI());
```

```
    }

    /**
     * A native method that is implemented by the 'getmethodidexample1' native
library,
     * which is packaged with this application.
     */
    public native String stringFromJNI();
}
```

testclass.java

```
package com.example.getmethodidexample1;

import android.util.Log;

public class testclass {

    static {
        Log.i("lilac", "class init ");
    }

    public void call(){

    }
}
```

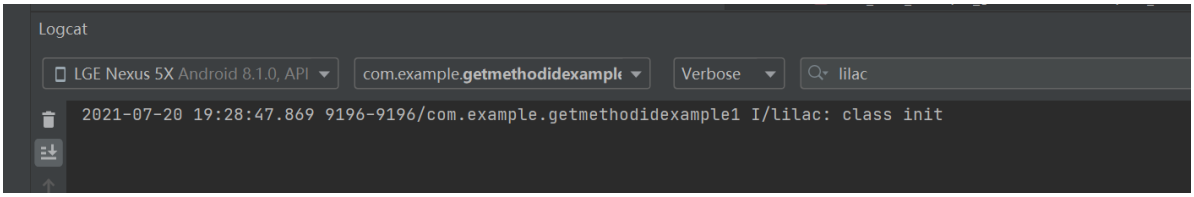
natiive-lib.cpp

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

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_getmethodidexample1_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject /* this */) {
    jclass context_clz = env-
>FindClass("com/example/getmethodidexample1/testclass");
    env->GetMethodID(context_clz, "call", "()V");

    return env->NewStringUTF("");
}
```

运行



这和我们所预期的一样，静态代码块在类初始化中被调用。如果我们在这段静态代码块中做某些操作（必须能影响到目标函数的运行），那不就是一种隐蔽且有效的Anti-Unidbg了吗？

事实上，这种操作也可以在目标函数之前的任意时机，或者发生在目标函数通过JNI与JAVA交互，call的某个JAVA方法中。只是相比*GetMethodID*这个时机，隐蔽性差一些，但如果DEX经过较好的保护，没办法脱壳查看其具体内容，那每个时机点都很隐蔽。

所以问题的关键就在于**某种能影响到目标函数的某种操作**，最好是JNITrace和Unidbg都注意不到的某种操作。

下面介绍两种可行的方向，首先我们得承认，没有哪个方案完美无缺，只有组合拳才能做到比较好的对抗效果。

2.1 陷阱1的设计

首先看一下基础代码，我们创建了一个文本文件，内容是12345，并在native中读取文本内容。

MainActivity.java

```
package com.example.readfile;

import androidx.appcompat.app.AppCompatActivity;

import android.annotation.SuppressLint;
import android.os.Bundle;
import android.widget.TextView;

import com.example.readfile.databinding.ActivityMainBinding;

import java.io.File;
import java.io.FileOutputStream;
import java.util.Objects;

public class MainActivity extends AppCompatActivity {

    static {
        System.loadLibrary("readfile");
        initFile();
    }

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        // Example of a call to a native method
        TextView tv = binding.sampleText;
        tv.setText(stringFromJNI());
    }

    /**
     * A native method that is implemented by the 'readfile' native library,
     * which is packaged with this application.
     */
    public native String stringFromJNI();

    public static void initFile(){
```

```
        String sourceString = "12345"; //待写入字符串
        byte[] sourceByte = sourceString.getBytes();
        if(null != sourceByte){
            try {
                @SuppressLint("SdCardPath") File file = new
File("/data/data/com.example.readfile/test.txt");     //文件路径（路径+文件名）
                if (!file.exists()) {  //文件不存在则创建文件，先创建目录
                    File dir = new
File(Objects.requireNonNull(file.getParent()));
                    dir.mkdirs();
                    file.createNewFile();
                }
                FileOutputStream outStream = new FileOutputStream(file);    //文件
输出流用于将数据写入文件
                outStream.write(sourceByte);
                outStream.close(); //关闭文件输出流
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
}
```

native-lib.cpp

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

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_readfile_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject /* this */) {
    // 提前在该位置新建一个文件，内容是12345
    const char *path = "/data/data/com.example.readfile/test.txt";

    //打开
    FILE *fp = fopen(path, "r");
    //读取
    char buff[50];//缓冲
    fgets(buff,50,fp);
    //关闭
    fclose(fp);

    return env->NewStringUTF(buff);
}
```

接下来我们新建两个类，A类有一个changeFile方法，将内容修改成ABCDE，B类有一个recoveryFile方法，将内容改回12345。

A.java

```
package com.example.readfile;
import android.annotation.SuppressLint;

import java.io.BufferedWriter;
import java.io.File;
```

```
import java.io.FileWriter;
import java.io.IOException;

public class A {

    public static void changeFile() {
        // 写文件
        @SuppressLint("SdCardPath") File file = new
File("/data/data/com.example.readfile/test.txt");
        try {
            FileWriter writer = new FileWriter(file, false);
            BufferedWriter bw = new BufferedWriter(writer);
            bw.write("ABCDE");
            bw.newLine();
            bw.close();
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

    };
}
```

B.java

```
package com.example.readfile;

import android.annotation.SuppressLint;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class B {
    public static void recoveryFile() {
        // 写文件
        @SuppressLint("SdCardPath") File file = new
File("/data/data/com.example.readfile/test.txt");
        try {
            FileWriter writer = new FileWriter(file, false);
            BufferedWriter bw = new BufferedWriter(writer);
            bw.write("12345");
            bw.newLine();
            bw.close();
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

    };
}
```

接下来我们修改一下*MainActivity.java*，让它在目标函数执行前后进行修改和复原

```
package com.example.readfile;
```

```java
import androidx.appcompat.app.AppCompatActivity;

import android.annotation.SuppressLint;
import android.os.Bundle;
import android.widget.TextView;

import com.example.readfile.databinding.ActivityMainBinding;

import java.io.File;
import java.io.FileOutputStream;
import java.util.Objects;

public class MainActivity extends AppCompatActivity {

    static {
        System.loadLibrary("readfile");
        initFile();
    }

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        // Example of a call to a native method
        TextView tv = binding.sampleText;
        A.changeFile();
        tv.setText(stringFromJNI());
        B.recoveryFile();
    }

    /**
     * A native method that is implemented by the 'readfile' native library,
     * which is packaged with this application.
     */
    public native String stringFromJNI();

    public static void initFile(){
        String sourceString = "12345"; //待写入字符串
        byte[] sourceByte = sourceString.getBytes();
        if(null != sourceByte){
            try {
                @SuppressLint("SdCardPath") File file = new
File("/data/data/com.example.readfile/test.txt");     //文件路径（路径+文件名）
                if (!file.exists()) {  //文件不存在则创建文件，先创建目录
                    File dir = new
File(Objects.requireNonNull(file.getParent()));
                    dir.mkdirs();
                    file.createNewFile();
                }
                FileOutputStream outStream = new FileOutputStream(file);   //文件
输出流用于将数据写入文件
                outStream.write(sourceByte);
```

```
                outStream.close(); //关闭文件输出流
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
}
```

目标函数中所读取的内容即”ABCDE“

在Unidbg使用者补环境时，提示文件访问/data/data/com.example.readfile/test.txt，于是他从手机中导出test.txt文件，但因为不知道要先执行changeFile方法，所以传入文件始终是错的。除此之外，我们可以将容易对比的文本文件换成一张图片，changeFile只修改其中一个微不足道的字节，这样的话，Unidbg使用者很难意识到这么点细微的差别。

除此之外，如果分析者仔细分析目标函数周围的代码，可能会发现我们所做的操作，因此可以利用GetMethodID/GetStaticMethodID 等JNI函数会对未初始化的类做初始化这一点特性，把我们的小操作隐藏到Native中调用。

看一下修改后的代码

A.java

```
package com.example.readfile;
import android.annotation.SuppressLint;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class A {

    static {
        changeFile();
    }

    public static void changeFile() {
        // 写文件
        @SuppressLint("SdCardPath") File file = new
File("/data/data/com.example.readfile/test.txt");
        try {
            FileWriter writer = new FileWriter(file, false);
            BufferedWriter bw = new BufferedWriter(writer);
            bw.write("ABCDE");
            bw.newLine();
            bw.close();
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

    };

    public void call(){

    };
}
```

B.java

```
package com.example.readfile;

import android.annotation.SuppressLint;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;

public class B {

    static {
        recoveryFile();
    }

    public static void recoveryFile() {
        // 写文件
        @SuppressLint("SdCardPath") File file = new
File("/data/data/com.example.readfile/test.txt");
        try {
            FileWriter writer = new FileWriter(file, false);
            BufferedWriter bw = new BufferedWriter(writer);
            bw.write("12345");
            bw.newLine();
            bw.close();
            writer.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

    };

    public void call(){

    };
}
```

MainActivity.java

```
package com.example.readfile;

import androidx.appcompat.app.AppCompatActivity;

import android.annotation.SuppressLint;
import android.os.Bundle;
import android.widget.TextView;

import com.example.readfile.databinding.ActivityMainBinding;

import java.io.File;
import java.io.FileOutputStream;
import java.util.Objects;
```

```
public class MainActivity extends AppCompatActivity {

    static {
        System.loadLibrary("readfile");
        initFile();
    }

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        binding = ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());

        // Example of a call to a native method
        TextView tv = binding.sampleText;
        tv.setText(stringFromJNI());
    }

    /**
     * A native method that is implemented by the 'readfile' native library,
     * which is packaged with this application.
     */
    public native String stringFromJNI();

    public static void initFile(){
        String sourceString = "12345"; //待写入字符串
        byte[] sourceByte = sourceString.getBytes();
        if(null != sourceByte){
            try {
                @SuppressLint("SdCardPath") File file = new
File("/data/data/com.example.readfile/test.txt");     //文件路径（路径+文件名）
                if (!file.exists()) {  //文件不存在则创建文件，先创建目录
                    File dir = new
File(Objects.requireNonNull(file.getParent()));
                    dir.mkdirs();
                    file.createNewFile();
                }
                FileOutputStream outStream = new FileOutputStream(file);   //文件
输出流用于将数据写入文件
                outStream.write(sourceByte);
                outStream.close(); //关闭文件输出流
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };
}
```

最后是关键的*native-lib.cpp*，我们利用两个GetMethodID实现对A以及B静态代码块的调用，让changeFile和

recoveryFile的操作更加隐蔽。

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

```
#include <string>

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_getmethodidexample0_MainActivity_stringFromJNI(
        JNIEnv* env,
        jobject mainactivity /* this */) {

    jclass context_clz = env->FindClass("android/content/Context");
    jmethodID methodID_getPackageName = env->GetMethodID(context_clz,
"getPackageName", "()Ljava/lang/String;");

    auto packageName = (jstring)env->CallObjectMethod(mainactivity,
methodID_getPackageName);

    const char* c_package_name = env->GetStringUTFChars(packageName, nullptr);
    return env->NewStringUTF(c_package_name);
}
```

如果分析者不了解可以这么做，那他会被迷惑和戏耍。

2.2 陷阱2的设计

陷阱1有所限制，要求有个文件作为中间载体，更自由的方式是在某个时机（比如上述的GetMethodID）加载另一个SO，我们叫它 monitor.so，它的功能是修改目标SO中的内存，比如某个全局变量的值，这个值会改变目标函数的执行流。

别忘了，Unidbg不仅对目标进程的jAVA世界一无所知，在Native的层面上，也无法感知目标SO及其依赖外的其他SO。

三、尾声

本节只提供了一些原型，安全开发人员可以根据业务和自身所长，继续开发更好的"无感知陷阱"。