

前言

[上一篇文章](#)分析了Obfuscapk，这一篇文章继续分析另一个apk加固工具advmp。

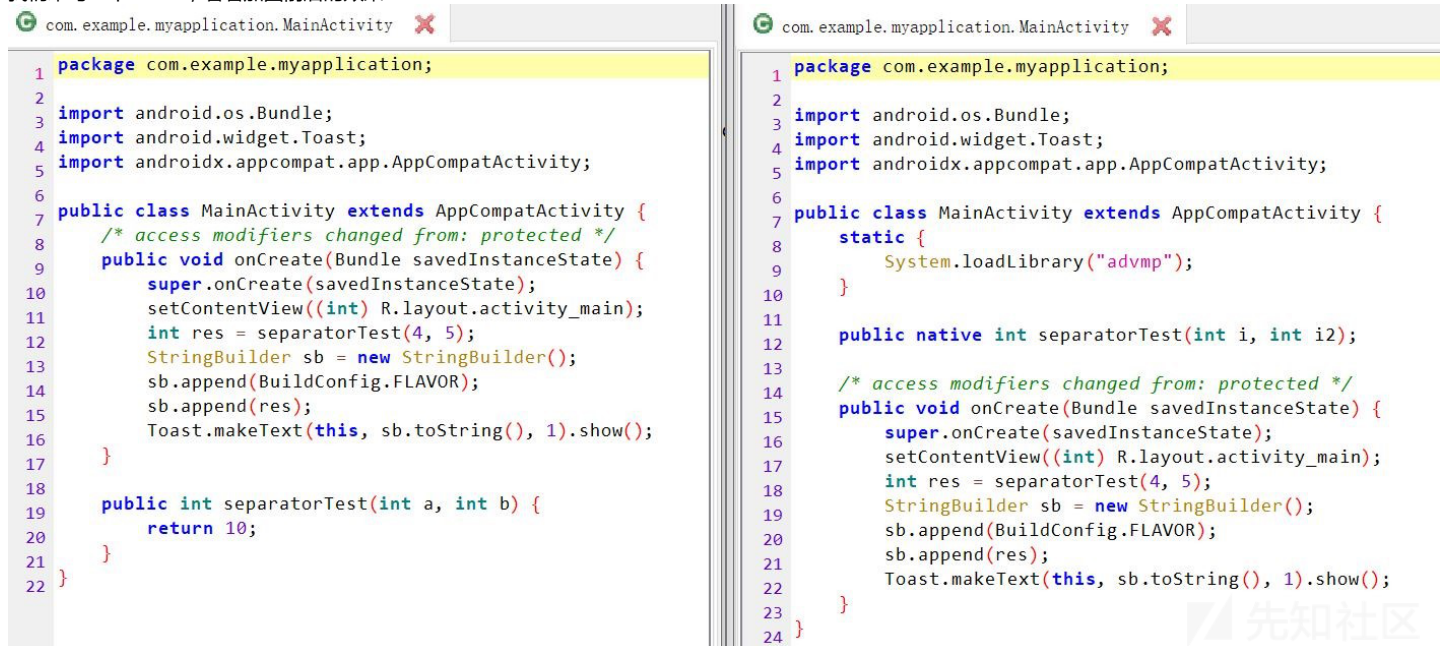
作者说明：<https://www.cnblogs.com/develop/p/4397397.html>

github地址：<https://github.com/chago/ADVMP>

简单来说，advmp参考dalvik虚拟机的解释器对字节码进行解释执行。代码结构如下。

- AdvmpTest：测试用的项目。
- base：Java写的工具代码。
- control-centre：Java写的控制加固流程的代码。
- separator：Java写的抽离方法指令，然后将抽离的指令按照自定义格式输出，并同时输出C文件的代码。
- template/jni：C写的解释器的代码。
- ycformat：Java写的用于保存抽取出的指令等数据的自定义的文件格式代码。

我们来写一个demo，看看加固前后的效果：



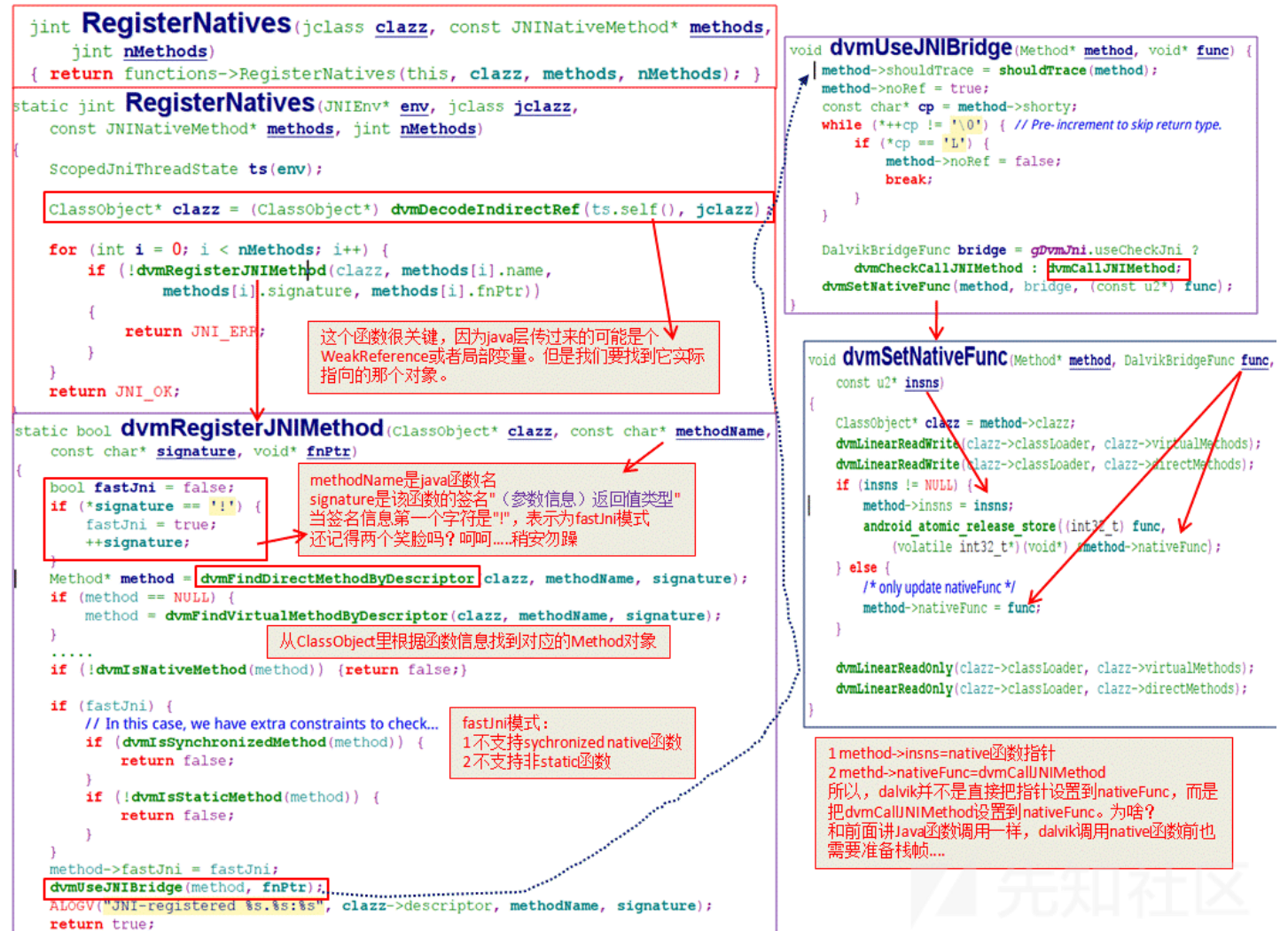
```
com.example.myapplication.MainActivity X
1 package com.example.myapplication;
2 import android.os.Bundle;
3 import android.widget.Toast;
4 import androidx.appcompat.app.AppCompatActivity;
5
6 public class MainActivity extends AppCompatActivity {
7     /* access modifiers changed from: protected */
8     public void onCreate(Bundle savedInstanceState) {
9         super.onCreate(savedInstanceState);
10        setContentView((int) R.layout.activity_main);
11        int res = separatorTest(4, 5);
12        StringBuilder sb = new StringBuilder();
13        sb.append(BuildConfig.FLAVOR);
14        sb.append(res);
15        Toast.makeText(this, sb.toString(), 1).show();
16    }
17
18    public int separatorTest(int a, int b) {
19        return 10;
20    }
21 }
22

com.example.myapplication.MainActivity X
1 package com.example.myapplication;
2 import android.os.Bundle;
3 import android.widget.Toast;
4 import androidx.appcompat.app.AppCompatActivity;
5
6 public class MainActivity extends AppCompatActivity {
7     static {
8         System.loadLibrary("advmp");
9     }
10
11    public native int separatorTest(int i, int i2);
12
13    /* access modifiers changed from: protected */
14    public void onCreate(Bundle savedInstanceState) {
15        super.onCreate(savedInstanceState);
16        setContentView((int) R.layout.activity_main);
17        int res = separatorTest(4, 5);
18        StringBuilder sb = new StringBuilder();
19        sb.append(BuildConfig.FLAVOR);
20        sb.append(res);
21        Toast.makeText(this, sb.toString(), 1).show();
22    }
23 }
24
```

代码中默认对separatorTest方法进行加固。可以看到加固后separatorTest方法已经变成一个native方法，并且加载了advmp.so。

dalvik虚拟机的解释器

首先参考infoQ上几篇文章回顾下dalvik虚拟机的解释器。我们知道可以通过RegisterNatives函数注册native方法，最后调用到dvmSetNativeFunc函数中将Method的native



在dvmCallMethodV函数中将ins指向第一个参数，如果不是静态方法将this指针放入ins，根据后面参数的类型依次将后面的参数放入ins。

```

clazz = callPrep(self, method, obj, false);
if (clazz == NULL)
    return;

/* "ins" for new frame start at frame pointer plus locals */
ins = ((u4*)self->interpSave.curFrame) +
    (method->registersSize - method->insSize);

//ALOGD("  FP is %p, INs live at >= %p", self->interpSave.curFrame, ins);

/* put "this" pointer into in0 if appropriate */
if (!dvmIsStaticMethod(method)) {
#ifdef WITH_EXTRA_OBJECT_VALIDATION
    assert(obj != NULL && dvmIsHeapAddress(obj));
#endif
    *ins++ = (u4) obj;
    verifyCount++;
}

while (*desc != '\0') {
    switch (*(desc++)) {
        case 'D': case 'J': {
            u8 val = va_arg(args, u8);
            memcpy(ins, &val, 8);          // EABI prevents direct store
            ins += 2;
            verifyCount += 2;
            break;
        }
        case 'F': {
            /* floats were normalized to doubles; convert back */
            float f = (float) va_arg(args, double);
            *ins++ = dvmFloatToU4(f);
            verifyCount++;
            break;
        }
    }
}

```

如果调用的是native方法就会进入Method的nativeFunc指向的dvmCallJNIMethod函数；如果调用的是java方法就会进入dvmInterpret函数。

```

if (dvmIsNativeMethod(method)) {
    TRACE_METHOD_ENTER(self, method);
    /*
     * Because we leave no space for local variables, "curFrame" points
     * directly at the method arguments.
     */
    (*method->nativeFunc)((u4*)self->interpSave.curFrame, pResult,
                          method, self);
    TRACE_METHOD_EXIT(self, method);
} else {
    dvmInterpret(self, method, pResult);
}

```

下面我们来分析dvmInterpret函数。dvmInterpret函数选择解释器(这里以Portable解释器为例)，调用dvmInterpretPortable函数。dvmInterpretPortable函数通过DEFIN handlerTable[kNumPackedOpcodes]数组。数组元素通过H宏定义，比如H(OP_RETURN_VOID)展开后得到&&op_OP_RETURN_VOID，表示op_OP_RETURN_VOID的


```

void dvmInterpretPortable(Thread* self)
{
    DvmDex* methodClassDex; // curMethod->clazz-
    JValue retval;
    const Method* curMethod; // method we're interp
    const u2* pc; // program counter
    u4* fp; // frame pointer
    u2 inst; // current instruction
    u4 ref; // 16 or 32-bit quantity fetched dir
    u2 vsrc1, vsrc2, vdst; // usually used for register in
    /* method call setup */
    const Method* methodToCall;
    bool methodCallRange;
    /* static computed goto table */
    DEFINE_GOTO_TABLE(handlerTable);
    curMethod = self->interpSave.method;
    pc = self->interpSave.pc;
    fp = self->interpSave.curFrame;
    retval = self->interpSave.retval; /* only need for kInterpEntryRe
    methodClassDex = curMethod->clazz->pDvmDex;
}

```

```

#define DEFINE_GOTO_TABLE(_name) \
    static const void* _name[kNumPackedOpcodes] = {
    /* BEGIN(libdex- goto- table); GENERATED AUTOM
    H(OP_NOP),
    H(OP_MOVE),
    H(OP_MOVE_FROM16),
    H(OP_MOVE_16),
    H(OP_MOVE_WIDE),
    H(OP_MOVE_WIDE_FROM16),
    H(OP_MOVE_WIDE_16),
    H(OP_MOVE_OBJECT),
    H(OP_MOVE_OBJECT_FROM16),
    H(OP_MOVE_OBJECT_16),
    H(OP_MOVE_RESULT),
    H(OP_MOVE_RESULT_WIDE),
    H(OP_MOVE_RESULT_OBJECT),
    H(OP_MOVE_EXCEPTION),
    H(OP_RETURN_VOID),

```

H(OP_RETURN_VOID) →
&&op_OP_RETURN_VOID

define H(_op) &&op_##_op
define **HANDLE_OPCODE**(_op) op_##_op:

HANDLE_OPCODE(OP_RETURN_VOID) →
op_OP_RETURN_VOID:

之后dvmInterpretPortable函数中调用了FINISH(0)取第一条指令并执行。移动PC，然后获取对应指令的操作码到inst。根据inst获取该指令的操作码(一条指令包含操作码和

```

methodToCall = (const Method*) - 1;
FINISH(0); /* fetch and exe
/* --- start of opcodes --- */
/* File: c/ OP_NOP.cpp */
HANDLE_OPCODE(OP_NOP)
    FINISH(1);
OP_END
/* File: c/ OP_MOVE.cpp */
HANDLE_OPCODE(OP_MOVE /*vA, vB*/)
    vdst = INST_A(inst);
    vsrc1 = INST_B(inst);
    ILOGV("| move%s v%d,v%d %s(v%d=0x%08x)",
        (INST_INST(inst) == OP_MOVE) ? "" : "- object",
        kspacing, vdst, GET_REGISTER(vsrc1));
    SET_REGISTER(vdst, GET_REGISTER(vsrc1));
    FINISH(1);
OP_END

```

```

# define FINISH(_offset) {
    ADJUST_PC(_offset);
    inst = FETCH(0);
    if (self->interpBreak.ctl.subMode) {
        dvmCheckBefore(pc, fp, self);
    }
    goto *handlerTable[INST_INST(inst)];
}

```

File:表示这段代码从哪个文件里搞过来的。目录dalvik/vm/interp/c
HANDLE_OPCODE: 定义goto label
注意，后面一大段代码其实都位于dvmInterpretPortable这个函数里
OP_END: 表示这段goto label结束无实际意义，该宏为空定义

ADJUST_PC(0): 调整PC的位置

```

# define ADJUST_PC(_offset) do {
    pc += _offset;
    EXPORT_EXTRA_PC();
} while (false)

```

FETCH(0): 获取当前PC指令的前个字节，并保存到inst变量中
Inst类型为u2。即两个字节。也就是说，dex中指令的前2个字节为操作码

```

# define FETCH(_offset) (pc[_offset])

```

取出指令的操作码，然后goto到对应的label
这就是portable模式下java函数执行的秘密了....

Inst已经在FINISH(0)中取得了，代表当前的java操作码，INST_A,INST_B则是从这条指令里取出操作数

向下移动PC，以指向接下来的指令
根据不同的操作符，偏移值也不同，OP_MOVE是1，还有2,3的

ADVMP原理

了解了dalvik虚拟机的解释器原理之后我们就可以理解ADVMP加壳的过程了。

插入System.loadLibrary指令加载advmp.so：

```
TypeDescription classDesc = AndroidManifestHelper.findFirstClass(new File(mApkUnpackDir, "AndroidManifest.xml"));
InstructionInsert01 instructionInsert01 = new InstructionInsert01(new File(mApkUnpackDir, "classes.dex"), classDesc);
instructionInsert01.insert();
```

先知社区

抽取separatorTest方法的代码写到classes.yc文件中，生成新的separatorTest方法(native)。yc文件为自定义格式，保存了抽取出来的方法指令/访问标志/参数个数等等信息

```
public Method rewrite(@NonNull Method value) {
    if (mConfigHelper.isValid(value)) {
        mSeparatedMethod.add(value);
        // 抽取代码。
        YcFormat.SeparatorData separatorData = new YcFormat.SeparatorData();
        separatorData.methodIndex = mSeparatedMethod.size();
        separatorData.accessFlag = value.getAccessFlags();
        separatorData.paramSize = value.getParameters().size();
        separatorData.registerSize = value.getImplementation().getRegisterCount();

        separatorData.paramShortDesc = new StringItem();
        separatorData.paramShortDesc.str = MethodHelper.genParamsShortDesc(value).getBytes();
        separatorData.paramShortDesc.size = separatorData.paramShortDesc.str.length;

        separatorData.insts = MethodHelper.getInstructions((DexBackedMethod) value);
        separatorData.instSize = separatorData.insts.length;
        separatorData.size = 4 + 4 + 4 + 4 + 4 + separatorData.paramShortDesc.size + 4 + (separatorData.instSize * 2) + 4;
        mSeparatedMethod.add(separatorData);

        // 下面这么做的目的是要把方法的name删除，否则生成的dex安装的时候会有这个错误：INSTALL_FAILED_DEXOPT。
        List<? extends MethodParameter> oldParams = value.getParameters();
        List<ImmutableMethodParameter> newParams = new ArrayList<>();
        for (MethodParameter mp : oldParams) {
            newParams.add(new ImmutableMethodParameter(mp.getType(), mp.getAnnotations(), null));
        }

        // 生成一个新的方法。
        return new ImmutableMethod(value.getDefiningClass(), value.getName(), newParams, value.getReturnType(), value.getAccessFlags());
    }
}
```

先知社区

将separatorTest方法对应的C代码和注册该方法的C代码写到advmp_separator.cpp文件中，然后将该文件中的代码合并到定义了JNI_OnLoad方法的avmp.cpp文件中，如

```
jint separatorTest (JNIEnv* env, jobject thiz, jint a, jint b)
{
    jvalue result = BWdvmInterpretPortable(gAdvmp.ycFile->GetSeparatorData(0), env, thiz, a, b);
    return result.i;
}

bool registerNatives0(JNIEnv* env)
{
    const char* classDesc = "com/example/myapplication/MainActivity";
    const JNINativeMethod methods[] =
    {{ "separatorTest", "(II)I", (void*)separatorTest },,};
    jclass clazz = env->FindClass(classDesc);
    if (!clazz) { MY_LOG_ERROR("can't find class:%s!", classDesc); return false; }
    bool bRet = false;
    if ( JNI_OK == env->RegisterNatives(clazz, methods, array_size(methods)) ) { bRet = true; }
    else { MY_LOG_ERROR("classDesc:%s, register method fail.", classDesc); }
    env->DeleteLocalRef(clazz);
    return bRet;
}

void registerFunctions(JNIEnv* env)
{
    if (!registerNatives0(env))
    {
        MY_LOG_ERROR("register method fail."); return;
    }
}
```

先知社区

JNI_OnLoad中首先调用registerFunctions函数然后释放并解析yc文件：

```

JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* env = NULL;

    if (vm->GetEnv((void **)&env, JNI_VERSION_1_4) != JNI_OK) {
        return JNI_ERR;
    }

    // 注册本地方法。
    registerFunctions(env);

    // 获得apk路径。
    gAdvmp.apkPath = GetAppPath(env);
    MY_LOG_INFO("apk path: %s", gAdvmp.apkPath);

    // 释放yc文件。
    gAdvmp.ycSize = ReleaseYcFile(gAdvmp.apkPath, &gAdvmp.ycData);
    if (0 == gAdvmp.ycSize) {
        MY_LOG_WARNING("release Yc file fail!");
        goto _ret;
    }

    // 解析yc文件。
    gAdvmp.ycFile = new YcFile;
    if (!gAdvmp.ycFile->parse(gAdvmp.ycData, gAdvmp.ycSize)) {
        MY_LOG_WARNING("parse Yc file fail.");
        goto _ret;
    }

_ret:
    return JNI_VERSION_1_4;
}

```



在separatorTest方法对应的C代码中调用了BwdvmInterpretPortable函数，第一个参数为从yc文件中得到的data，BwdvmInterpretPortable函数首先要做类似于dvmCal

```

// 获得参数寄存器个数。
size_t paramRegCount = getParamRegCount(separatorData);

// 设置参数寄存器的值。
if (isStaticMethod(separatorData)) {
    startIndex = separatorData->registerSize - separatorData->paramSize;
} else {
    startIndex = separatorData->registerSize - separatorData->paramSize;
    fp[startIndex++] = (u4)thiz;
}
for (int i = startIndex, j = 0; j < separatorData->paramSize; j++ ) {
    if ('D' == separatorData->paramShortDesc.str[i] || 'J' == separatorData->paramShortDesc.str[i]) {
        fp[i++] = params[j].j & 0xFFFFFFFF;
        fp[i++] = (params[j].j >> 32) & 0xFFFFFFFF;
    } else {
        fp[i++] = params[j].i;
    }
}

pc = separatorData->insts;

/* static computed goto table */
DEFINE_GOTO_TABLE(handlerTable);

// 抓取第一条指令。
FINISH(0);

/*--- start of opcodes ---*/

```



参考资料

1. <https://www.infoq.cn/profile/1279859>

点击收藏 | 0 关注 | 1

[上一篇：windows样本高级静态分析之识...](#) [下一篇：jsonp的一些安全问题](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)