手动绕过百度加固Debug.isDebuggerConnected反调试的方法

原创

Fly20141201 于 2017-10-14 22:29:55 发布 3699 收藏 4
分类专栏： Android逆向学习 Android系统安全和逆向分析研究 文章标签： Debug.isDebuggerConn 百度加固 反调试 dvmDbgIsDebuggerConn

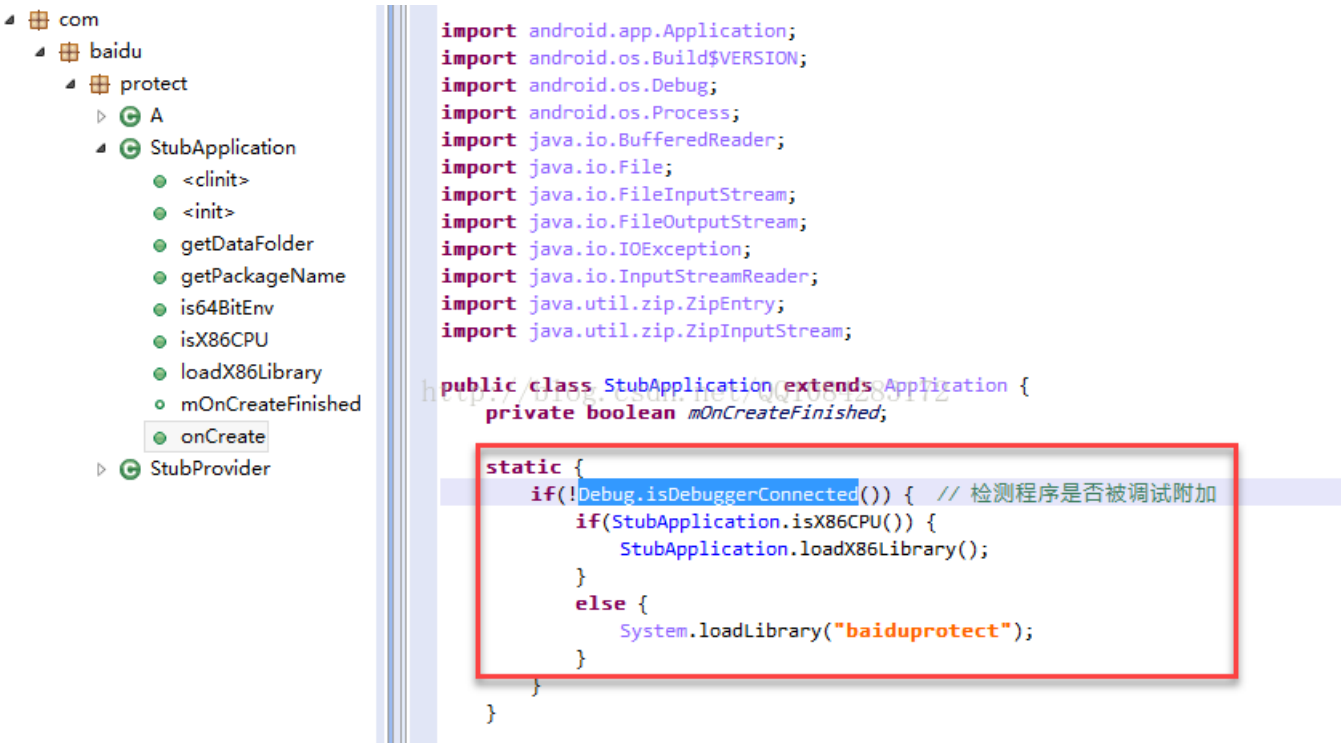
本文链接：https://blog.csdn.net/QQ1084283172/article/details/78237571


Android逆向学习 同时被 2 个专栏收录
58 篇文章 6 订阅
订阅专栏


Android系统安全和逆向分析研究
72 篇文章 59 订阅
订阅专栏

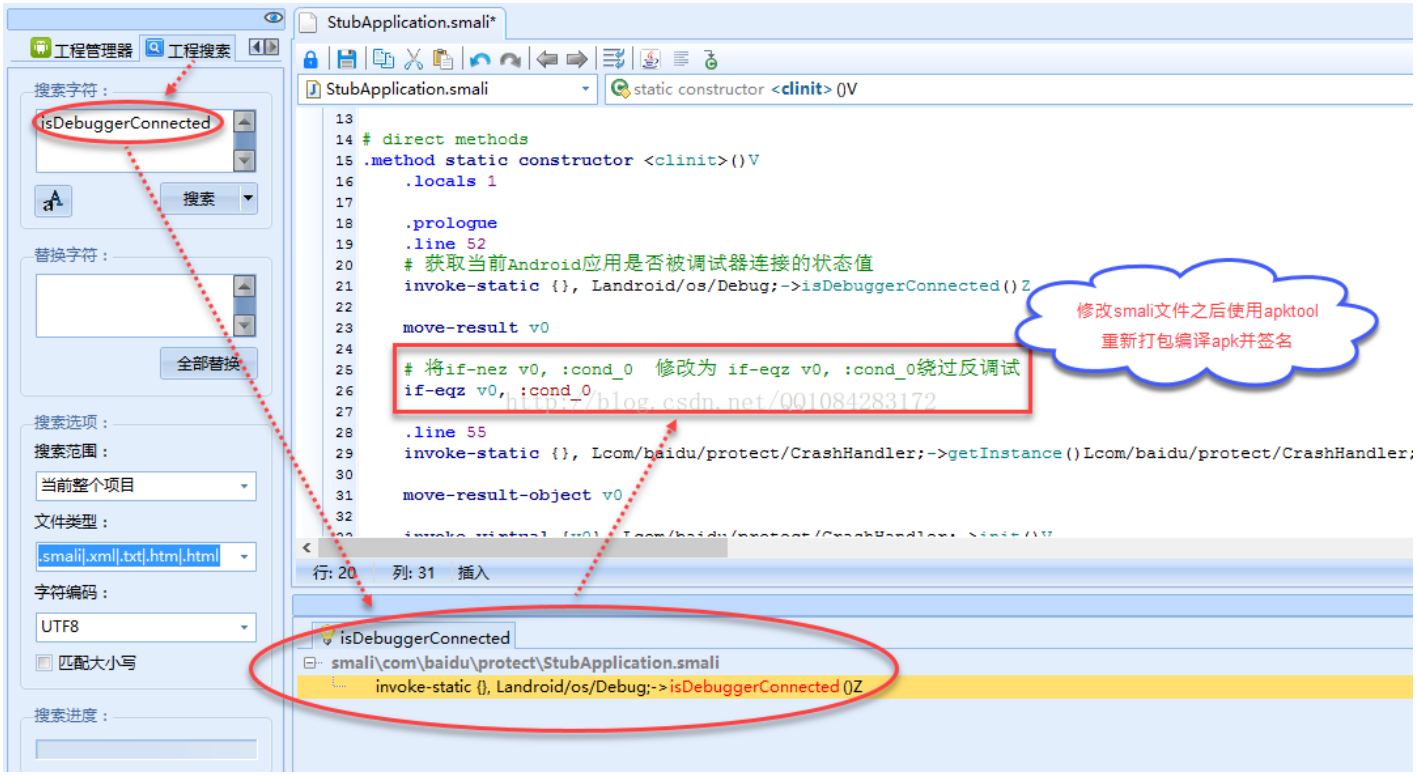
本文博客地址：http://blog.csdn.net/qq1084283172/article/details/78237571

1.调用Debug.isDebuggerConnected函数这种反调试的Android加固比较少，主要是因为Debug.isDebuggerConnected这种方法的反调试作用不大，过掉也比较容易。百度的Android应用加固使用了调用Debug.isDebuggerConnected函数检测程序被调试的反调试方法。



2.绕过基于Debug.isDebuggerConnected函数检测进行反调试的方法整理：

【1】.对基于Debug.isDebuggerConnected函数检测进行反调试的百度加固的Android应用使用Apktool工具进行解包处理，在解包后的所有smali文件中全局搜索关键字符串“**isDebuggerConnected**”，查找到Debug.isDebuggerConnected函数检测反调试的smali汇编代码的位置，修改smali代码检测位置处的**判断条件** 绕过Debug.isDebuggerConnected函数检测的反调试，使用Apktool工具对解包修改后的百度加固的Android应用的smali文件进行重新打包和签名处理，推荐使用AndroidKiller工具进行这所有的操作。



【2】.在 **Dalvik模式下** 进行百度加固的Andorid应用的动态so库文件调试时，使用IDA脚本IDC文件 Hook VMDebug.isDebuggerConnected函数的Native层实现函数dvmDbgIsDebuggerConnected，修改dvmDbgIsDebuggerConnected函数的返回值（基于VMDebug.isDebuggerConnected函数的Nativev层的函数dvmDbgIsDebuggerConnected）并且dvmDbgIsDebuggerConnected函数在libdvm.so库文件中还是导出函数，具体的原理参考《在百度加固中正确使用ida的姿势》。

在百度的加固中会使用这句话来判断本程序是否被调试

if (!Debug.isDebuggerConnected())

这样对于ida使用者而言,网上的公开调试方式so的方式就不可以用了,怎么办呢?

闲来没事分析了一下,调用的是下面这个函数

public static boolean isDebuggerConnected(){

return VMDebug.isDebuggerConnected();

}

在libcore\dalvik\src\main\java\dalvik\system\VMDebug.java下：

public static native boolean isDebuggerConnected()

发现是一个native函数，那就hook它过调试检测。

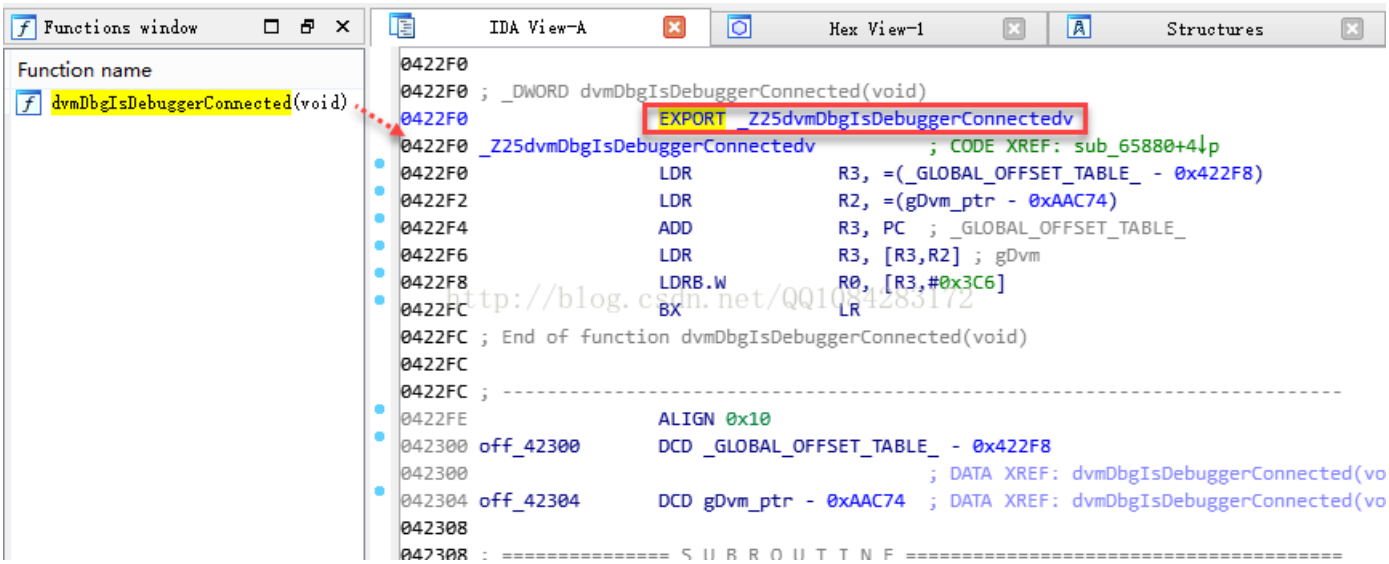
Hook dvmDbgIsDebuggerConnected函数的IDA脚本（至于脚本中，第一次r0寄存器的值为1的时候为什么不改成0，需要参考一下Android源码的实现才能理解）。

```
from idaapi import *
from idc import *

debug_addr = LocByName("_Z25dvmDbgIsDebuggerConnectedv")
end = FindFuncEnd(debug_addr) - 0x02
count = 0;

class DumpHook(DBG_Hooks):
    def dbg_bpt(self,tid,ea):
        global count
        r0 = GetRegValue('r0')
        if r0 == 1:
            count = count + 1
            if count == 2:
                SetRegValue(0,"r0")
        ResumeProcess()
        return 0

AddBpt(end)
debug = DumpHook()
debug.hook()

print "hook"
```

在 **dalvik虚拟机模式** 下，VMDebug.isDebuggerConnected函数最终调用的是Native函数 **dvmDbgIsDebuggerConnected**。

```
358 /*
359  * static boolean isDebuggerConnected()
360  *
361  * Returns "true" if a debugger is attached.
362  */
363 static void Dalvik_dalvik_system_VMDebug_isDebuggerConnected(const u4* args,
364     JValue* pResult)
365 {
366     UNUSED_PARAMETER(args);
367
368     RETURN_BOOLEAN(dvmDbgIsDebuggerConnected());
369 }
370
```


函数**dvmDbgIsDebuggerConnected**是 **libdvm.so**库文件 中的**导出函数**。



在 **art虚拟机模式** 下，VMDebug.isDebuggerConnected函数最终调用的是Native函数**art::Dbg::IsDebuggerActive** 。

```
116
117 static jboolean VMDebug_isDebuggerConnected(JNIEnv*, jclass) {
118   return Dbg::IsDebuggerActive();
119 }
120
```


函数**art::Dbg::IsDebuggerActive**是 **libart.so**库文件 中的**导出函数**。

```
.text:0006F4B4 ; =============== S U B R O U T I N E =======================================
.text:0006F4B4
.text:0006F4B4
.text:0006F4B4 ; _DWORD art::Dbg::IsDebuggerActive(art::Dbg *__hidden this)
.text:0006F4B4                 EXPORT _ZN3art3Dbg16IsDebuggerActiveEv
.text:0006F4B4 _ZN3art3Dbg16IsDebuggerActiveEv         ; CODE XREF: art::JDWP::JdwpState::LastDebuggerActivity(void)+E↓p
.text:0006F4B4                                         ; art::Dbg::IsDebuggerActive(void)↓j ...
.text:0006F4B4                 LDR     R3, =(byte_214BFC - 0x6F4BA)
.text:0006F4B6                 ADD     R3, PC  ; byte_214BFC
.text:0006F4B8                 LDRB    R0, [R3]
.text:0006F4BA                 BX      LR
.text:0006F4BA ; End of function art::Dbg::IsDebuggerActive(void)
.text:0006F4BA
```

【3】.解包百度加固的Android应用，把百度加固的 libbaiduprotect.so 库文件单独拿出来，自己编写一个**dalvik虚拟机模式下**的 loader程序 调用百度加固的 libbaiduprotect.so 库文件中的JNI_Onload函数bypass掉壳代码和反调试，具体的方法可以参考看雪论坛的文章《百度加固逆向分析》。

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

int main()
{
    JavaVM* vm;
    JNIEnv* env;
    jint res;

    JavaVMInitArgs vm_args;
    JavaVMOption options[1];
    options[0].optionString = "-Djava.class.path=.";
    vm_args.version=0x00010002;
    vm_args.options=options;
    vm_args.nOptions =1;
    vm_args.ignoreUnrecognized=JNI_TRUE;

    printf("[+] dlopen libdvm.so\n");
 // RTLD_LAZY RTLD_NOW
    void *handle = dlopen("/system/lib/libdvm.so", RTLD_LAZY);
    if(!handle) {

  printf("[-] dlopen libdvm.so failed!!\n");
  return 0;
    }

    // 先创建一个java虚拟机。因为JNI_ONload函数参数第一个参数为JavaVM。
    typedef int (*JNI_CreateJavaVM_Type)(JavaVM**, JNIEnv**, void*);
    JNI_CreateJavaVM_Type JNI_CreateJavaVM_Func = (JNI_CreateJavaVM_Type)dlsym(handle, "JNI_CreateJavaVM");
    if(!JNI_CreateJavaVM_Func) {

  printf("[-] dlsym failed\n");
  return 0;
    }

 // 创建java虚拟机
    res = JNI_CreateJavaVM_Func(&vm, &env, &vm_args)
    void* si = dlopen("/data/local/tmp/libbaiduprotect.so", RTLD_LAZY);
    if(si == NULL) {

  printf("[-] dlopen err!\n");
  return 0;
    }

    typedef jint (*FUN)(JavaVM* vm, void* res);
    FUN func_onload = (FUN)dlsym(si, "JNI_OnLoad");
 // 将断点下在了这里可以正好获取到JNI_Onload的函数地址。
    if(func_onload==NULL)
        return 0;

 // 调用JNI_Onload函数
    func_onload(vm,NULL);
    return 0;
}
```

关于Android系统上dalvik虚拟机模式下java虚拟机的创建方法，可以参考书籍《Android框架揭秘》102页~110页中，关于dalvik虚拟机模式下java虚拟机创建的代码分析。

```
#include <jni.h>                                                    ❶

int main()
{
    JNIEnv *env;
    JavaVM *vm;
    JavaVMInitArgs vm_args;
    JavaVMOption options[1];
    jint res;
    jclass cls;
    jmethodID mid;
    jstring jstr;
    jclass stringClass;
    jobjectArray args;

    // 1. 生成 Java 虚拟机选项
    options[0].optionString = "-Djava.class.path=."                 ❷
    vm_args.version = 0x00010002;
    vm_args.options = options;
    vm_args.nOptions = 1;
    vm_args.ignoreUnrecognized = JNI_TRUE;

    // 2. 生成 Java 虚拟机
    res = JNI_CreateJavaVM(&vm, (void**)&env, &vm_args);            ❸

    // 3. 查找并加载类
    cls = (*env)->FindClass(env, "InvocationApiTest");

    // 4. 获取 main() 方法的 ID                                      ❹
    mid = (*env)->GetStaticMethodID(env, cls, "main", "([Ljava/lang/String;)V");

    // 5. 生成字符串对象，用作 main ( ) 方法的参数
    jstr = (*env)->NewStringUTF(env, "Hello Invocation API!!");     ❺
    stringClass = (*env)->FindClass(env, "java/lang/String");
    args = (*env)->NewObjectArray(env, 1, stringClass, jstr);

    // 6. 调用 main ( ) 方法
    (*env)->CallStaticVoidMethod(env, cls, mid, args);              ❻

    // 7. 销毁 Java 虚拟机
    (*vm)->DestroyJavaVM(vm);                                       ❼
}
```

Dalvik虚拟机模式下，java虚拟机的创建可以参考Andorid 4.4.4 r1的源码文件 **/dalvik/vm/Jni.cpp** 中的代码。

http://androidxref.com/4.4.4_r1/xref/dalvik/vm/Jni.cpp#3424

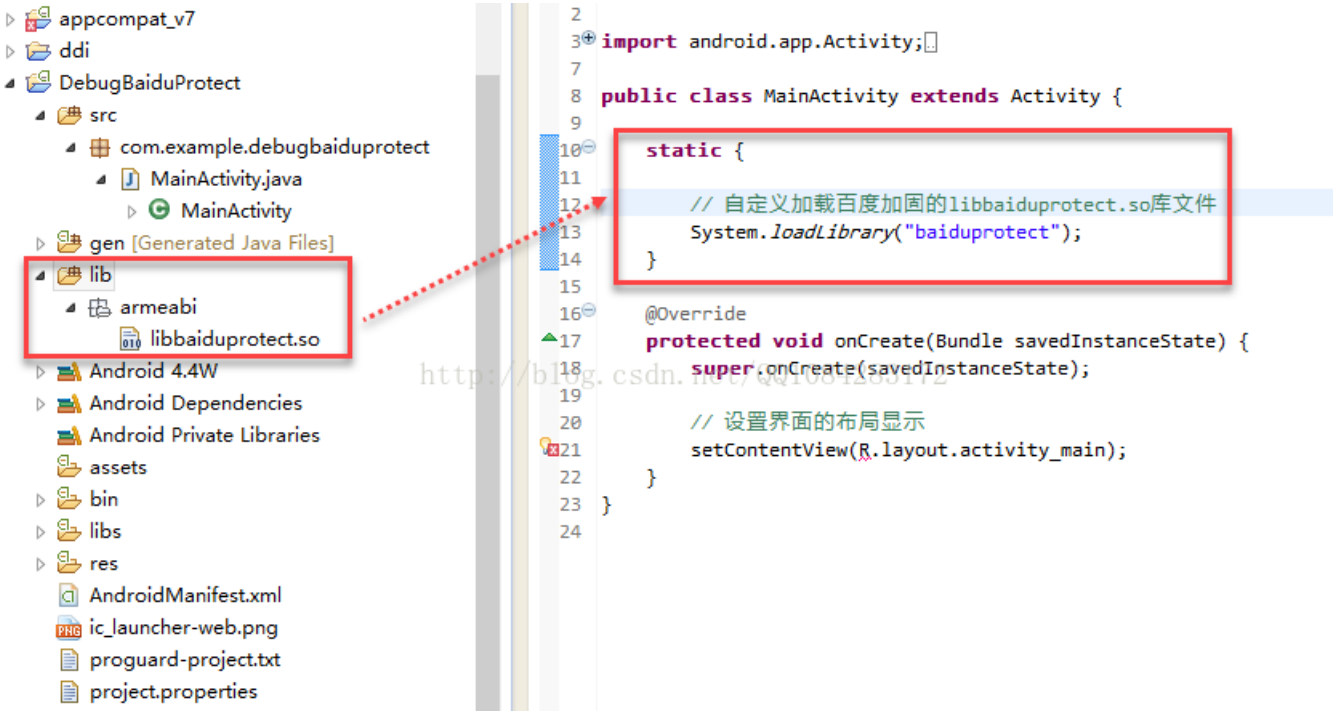
```
3418 /*
3419  * Create a new VM instance.
3420  *
3421  * The current thread becomes the main VM thread.  We return immediately,
3422  * which effectively means the caller is executing in a native method.
3423  */
3424 jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
3425     const JavaVMInitArgs* args = (JavaVMInitArgs*) vm_args;
3426     if (dvmIsBadJniVersion(args->version)) {
3427         ALOGE("Bad JNI version passed to CreateJavaVM: %d", args->version);
3428         return JNI_EVERSION;
3429     }
3430
3431     // TODO: don't allow creation of multiple VMs -- one per customer for now
3432
3433     /* zero globals; not strictly necessary the first time a VM is started */
3434     memset(&gDvm, 0, sizeof(gDvm));
3435
3436     /*
3437      * Set up structures for JNIEnv and VM.
3438      */
3439     JavaVMExt* pVM = (JavaVMExt*) calloc(1, sizeof(JavaVMExt));
3440     pVM->funcTable = &gInvokeInterface;
3441     pVM->envList = NULL;
3442     dvmInitMutex(&pVM->envListLock);
3443
3444     UniquePtr<const char*[]> argv(new const char*[args->nOptions]);
3445     memset(argv.get(), 0, sizeof(char*) * (args->nOptions));
3446
3447     /*
3448      * Convert JNI args to argv.
3449      *
3450      * We have to pull out vfprintf/exit/abort, because they use the
3451      * "extraInfo" field to pass function pointer "hooks" in.  We also
3452      * look for the -Xcheck:jni stuff here.
3453      */
3454     int argc = 0;
3455     for (int i = 0; i < args->nOptions; i++) {
3456         const char* optStr = args->options[i].optionString;
3457         if (optStr == NULL) {
3458             dvmFprintf(stderr, "ERROR: CreateJavaVM failed: argument %d was NULL\n", i);
3459             return JNI_ERR;
3460         } else if (strcmp(optStr, "vfprintf") == 0) {
3461             gDvm.vfprintfHook = (int (*)(FILE *, const char*, va_list))args->options[i].extraInfo;
3462         } else if (strcmp(optStr, "exit") == 0) {
3463             gDvm.exitHook = (void (*)(int)) args->options[i].extraInfo;
```


Art虚拟机模式下，java虚拟机的创建可以参考Android 4.4.4 r1的源码文件 **/art/runtime/jni_internal.cc** 中的代码。

http://androidxref.com/4.4.4_r1/xref/art/runtime/jni_internal.cc#2888

```
2885
2886 // JNI Invocation interface.
2887
2888 extern "C" jint JNI_CreateJavaVM(JavaVM** p_vm, JNIEnv** p_env, void* vm_args) {
2889   const JavaVMInitArgs* args = static_cast<JavaVMInitArgs*>(vm_args);
2890   if (IsBadJniVersion(args->version)) {
2891     LOG(ERROR) << "Bad JNI version passed to CreateJavaVM: " << args->version;
2892     return JNI_EVERSION;
2893   }
2894   Runtime::Options options;
2895   for (int i = 0; i < args->nOptions; ++i) {
2896     JavaVMOption* option = &args->options[i];
2897     options.push_back(std::make_pair(std::string(option->optionString), option->extraInfo));
2898   }
2899   bool ignore_unrecognized = args->ignoreUnrecognized;
2900   if (!Runtime::Create(options, ignore_unrecognized)) {
2901     return JNI_ERR;
2902   }
2903   Runtime* runtime = Runtime::Current();
2904   bool started = runtime->Start();
2905   if (!started) {
2906     delete Thread::Current()->GetJniEnv();
2907     delete runtime->GetJavaVM();
2908     LOG(WARNING) << "CreateJavaVM failed";
2909     return JNI_ERR;
2910   }
2911   *p_env = Thread::Current()->GetJniEnv();
2912   *p_vm = runtime->GetJavaVM();
2913   return JNI_OK;
2914 }
2915
```


【4】.自己编写个简单的Android程序，自定义加载百度加固的动态库文件libbaiduprotect.so，然后在这个Android应用的基础上进行百度加固动态库文件libbaiduprotect.so的动态调试。



3.这里再介绍一种 手动绕过百度加固Debug.isDebuggerConnected反调试的方法，比较实用也比较简单不需要太多的操作。在介绍这种手动过掉Debug.isDebuggerConnected函数反调试的方法之前，先了解一下Debug.isDebuggerConnected函数的执行流程,以Android 4.4.4 r1的源码为分析基础。

【1】.Debug.isDebuggerConnected函数是在Android 4.4.4 r1源码的文件 **/frameworks/base/core/java/android/os/Debug.java** 中实现的，该函数最终调用的是VMDebug.isDebuggerConnected函数。

http://androidxref.com/4.4.4_r1/xref/frameworks/base/core/java/android/os/Debug.java#isDebuggerConnected

```
/**
 * Determine if a debugger is currently attached.
 */
public static boolean isDebuggerConnected() {
    return VMDebug.isDebuggerConnected();
}
```


【2】.VMDebug.isDebuggerConnected函数是在Native层实现的，在Android 4.4.4 r1源码的文件 **/libcore/dalvik/src/main/java/dalvik/system/VMDebug.java** 中，到这里Debug.isDebuggerConnected函数的java层实现已经基本完成了，接下来是Debug.isDebuggerConnected函数在Native层的实现，由于Android系统可以运行在Dalvik虚拟机模式下或者Art虚拟机模式下，因此在Dalvik虚拟机模式下和Art虚拟机模式下，Debug.isDebuggerConnected函数底层的具体实现会有有所不同，需要分开来分析和学习。

http://androidxref.com/4.4.4_r1/xref/libcore/dalvik/src/main/java/dalvik/system/VMDebug.java#122

```
/**
 * Determines if a debugger is currently attached.
 *
 * @return true if (and only if) a debugger is connected.
 */
public static native boolean isDebuggerConnected();
```


【3】.在dalvik虚拟机模式下，VMDebug.isDebuggerConnected函数是在Android 4.4.4 r1源码的文件**/dalvik/vm/native/dalvik_system_VMDebug.cpp** 中实现的，具体就是对应Native层的函数**Dalvik_dalvik_system_VMDebug_isDebuggerConnected**。

http://androidxref.com/4.4.4_r1/xref/dalvik/vm/native/dalvik_system_VMDebug.cpp#Dalvik_dalvik_system_VMDebug_isDebuggerConnected

```
    Dalvik_dalvik_system_VMDebug_startEmulatorTracing },
{ "stopEmulatorTracing",        "()V",
    Dalvik_dalvik_system_VMDebug_stopEmulatorTracing },
{ "startInstructionCounting",   "()V",
    Dalvik_dalvik_system_VMDebug_startInstructionCounting },
{ "stopInstructionCounting",    "()V",
    Dalvik_dalvik_system_VMDebug_stopInstructionCounting },
{ "resetInstructionCount",      "()V",
    Dalvik_dalvik_system_VMDebug_resetInstructionCount },
{ "getInstructionCount",        "([I)V",
    Dalvik_dalvik_system_VMDebug_getInstructionCount },
{ "isDebuggerConnected",        "()Z",
    Dalvik_dalvik_system_VMDebug_isDebuggerConnected },
{ "isDebuggingEnabled",         "()Z",
    Dalvik_dalvik_system_VMDebug_isDebuggingEnabled },
{ "lastDebuggerActivity",       "()J",
    Dalvik_dalvik_system_VMDebug_lastDebuggerActivity },
{ "printLoadedClasses",         "(I)V",
    Dalvik_dalvik_system_VMDebug_printLoadedClasses },
{ "getLoadedClassCount",        "()I",
    Dalvik_dalvik_system_VMDebug_getLoadedClassCount },
{ "threadCpuTimeNanos",         "()J",
    Dalvik_dalvik_system_VMDebug_threadCpuTimeNanos },
```

Dalvik_dalvik_system_VMDebug_isDebuggerConnected函数的实现，Dalvik_dalvik_system_VMDebug_isDebuggerConnected最终调用的是libdvm.so库文件的导出函数**dvmDbgIsDebuggerConnected**。

```
/*
 * static boolean isDebuggerConnected()
 *
 * Returns "true" if a debugger is attached.
 */
static void Dalvik_dalvik_system_VMDebug_isDebuggerConnected(const u4* args,
    JValue* pResult)
{
    UNUSED_PARAMETER(args);

    // 最终调用的libdvm.so库文件中的导出函数dvmDbgIsDebuggerConnected
    RETURN_BOOLEAN(dvmDbgIsDebuggerConnected());
}
```


【4】.dvmDbgIsDebuggerConnected函数是在Android 4.4.4 r1源码的文件 **/dalvik/vm/Debugger.cpp** 中实现的，最终Debug.isDebuggerConnected函数的返回值是由全局对象gDvm的成员变量gDvm.debuggerActive决定的。

http://androidxref.com/4.4.4_r1/xref/dalvik/vm/Debugger.cpp#443

```
/*
 * Returns "true" if a debugger is connected.
 *
 * Does not return "true" if it's just a DDM server.
 */
bool dvmDbgIsDebuggerConnected()
{
    return gDvm.debuggerActive;
}
```


【5】.在art虚拟机模式下，VMDebug.isDebuggerConnected函数是在Android 4.4.4 r1源码的文件 **/art/runtime/native/dalvik_system_VMDebug.cc** 中实现的，具体就是实现函数**VMDebug_isDebuggerConnected**。

```
static JNINativeMethod gMethods[] = {
  NATIVE_METHOD(VMDebug, countInstancesOfClass, "(Ljava/lang/Class;Z)J"),
  NATIVE_METHOD(VMDebug, crash, "()V"),
  NATIVE_METHOD(VMDebug, dumpHprofData, "(Ljava/lang/String;Ljava/io/FileDescriptor;)V"),
  NATIVE_METHOD(VMDebug, dumpHprofDataDdms, "()V"),
  NATIVE_METHOD(VMDebug, dumpReferenceTables, "()V"),
  NATIVE_METHOD(VMDebug, getAllocCount, "(I)I"),
  NATIVE_METHOD(VMDebug, getHeapSpaceStats, "([J)V"),
  NATIVE_METHOD(VMDebug, getInstructionCount, "([I)V"),
  NATIVE_METHOD(VMDebug, getLoadedClassCount, "()I"),
  NATIVE_METHOD(VMDebug, getVmFeatureList, "()[Ljava/lang/String;"),
  NATIVE_METHOD(VMDebug, infopoint, "(I)V"),
  NATIVE_METHOD(VMDebug, isDebuggerConnected, "()Z"),
  NATIVE_METHOD(VMDebug, isDebuggingEnabled, "()Z"),
  NATIVE_METHOD(VMDebug, getMethodTracingMode, "()I"),
  NATIVE_METHOD(VMDebug, lastDebuggerActivity, "()J"),
  NATIVE_METHOD(VMDebug, printLoadedClasses, "(I)V"),
  NATIVE_METHOD(VMDebug, resetAllocCount, "(I)V"),
  NATIVE_METHOD(VMDebug, resetInstructionCount, "()V"),
  NATIVE_METHOD(VMDebug, startAllocCounting, "()V"),
  NATIVE_METHOD(VMDebug, startEmulatorTracing, "()V"),
  NATIVE_METHOD(VMDebug, startInstructionCounting, "()V"),
```

【6】.VMDebug_isDebuggerConnected函数最终是调用的 **Dbg::IsDebuggerActive**函数。

http://androidxref.com/4.4.4_r1/xref/art/runtime/native/dalvik_system_VMDebug.cc#117

```
static jboolean VMDebug_isDebuggerConnected(JNIEnv*, jclass) {
  return Dbg::IsDebuggerActive();
}
```


【7】.Dbg::IsDebuggerActive 函数是在Android 4.4.4 r1源码文件 **/art/runtime/debugger.cc** 中实现的，具体就是获取全局变量gDebuggerActive的状态值。

http://androidxref.com/4.4.4_r1/xref/art/runtime/debugger.cc#578

```
// Runtime JDWP state.
static JDWP::JdwpState* gJdwpState = NULL;
static bool gDebuggerConnected;  // debugger or DDMS is connected.
static bool gDebuggerActive;     // debugger is making requests.
static bool gDisposed;           // debugger called VirtualMachine.Dispose, so we should drop the connection.

bool Dbg::IsDebuggerActive() {
  return gDebuggerActive;
}
```


4.Debug.isDebuggerConnected函数的底层实现已经分析差不多啦，下面就说下 Dalvik虚拟机模式下 手动Debug.isDebuggerConnected反调试的方法。关于Android应用so库文件的动态调试就不详细介绍了，网上的教程很多自己去看。

【A】.由于Dalvik虚拟机模式下，函数**dvmDbgIsDebuggerConnected**是**libdvm.so库文件** 中的**导出函数**，因此在进行Dalvik虚拟机模式下的so库文件的

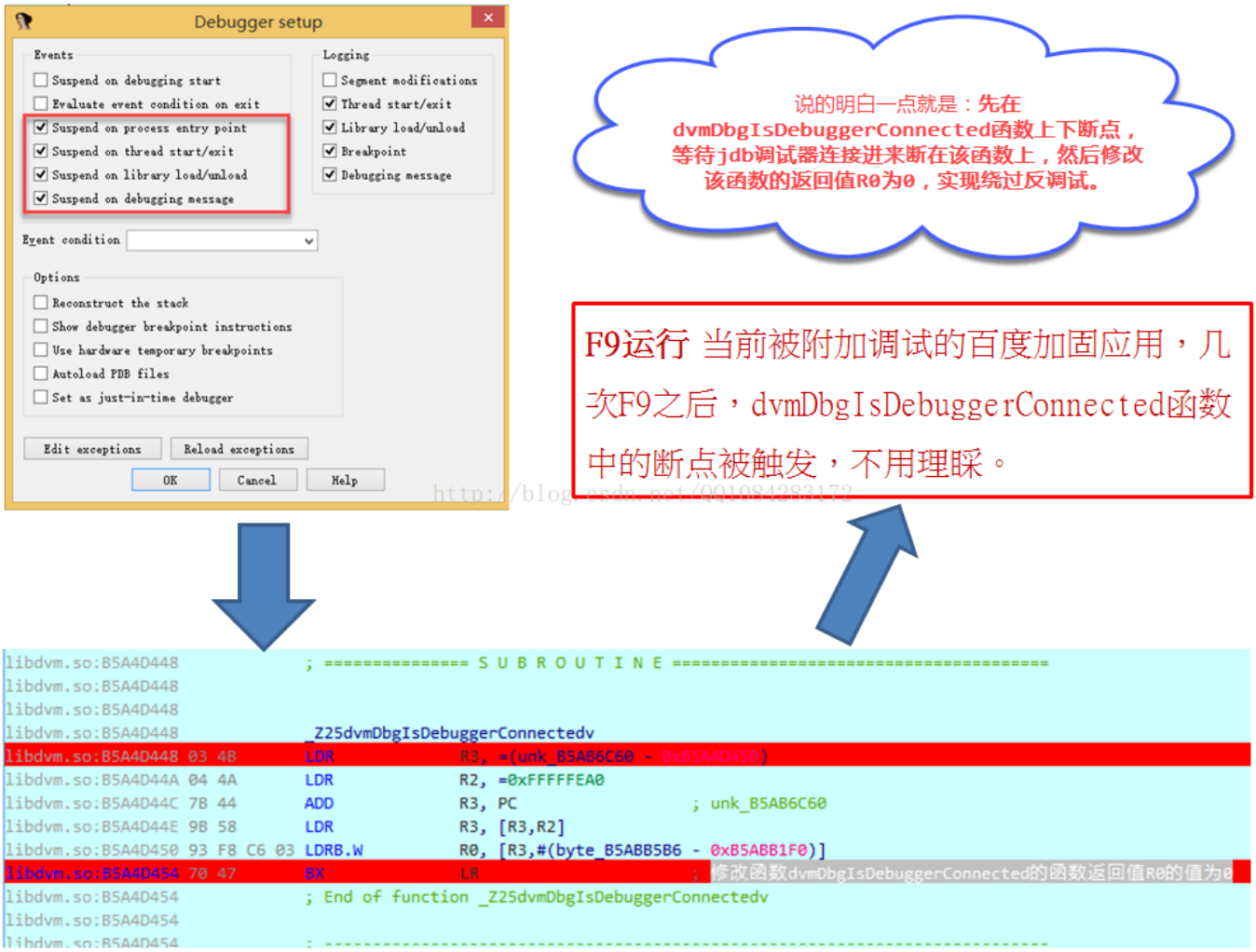
动态调试时，想要 **过掉Debug.isDebuggerConnected函数的反调试** 需要在 **libdvm.so库文件** 的dvmDbgIsDebuggerConnected函数开头和结尾的位置下断点进行拦截，然后修改dvmDbgIsDebuggerConnected函数的返回值为0即可。

【B】.由于Art虚拟机模式下，函数**art::Dbg::IsDebuggerActive**是**libart.so库文件** 中的**导出函数**，因此在进行Art虚拟机模式下的so库文件的

动态调试时，想要 **过掉Debug.isDebuggerConnected函数的反调试** 需要在**libart.so库文件** 的**art::Dbg::IsDebuggerActive**函数开头和结尾的位置下断点进行拦截，然后修改**art::Dbg::IsDebuggerActive**函数的返回值为0即可。

下面就以dalvik虚拟机模式下的百度加固的 libbaiduprotect.so 库文件动态调试为例，进行手动绕过百度加固Debug.isDebuggerConnected反调试的方法步骤说明。

【1】.百度加固的Android应用以调试模式启动等待调试以后，IDA Pro附加调试该百度加固的Android应用成功以后如下图设置IDA Pro的调试运行选项，并在**libdvm.so库文件** 的dvmDbgIsDebuggerConnected函数开头和结尾的位置下断点进行拦截，然后**F9** 运行当前被附加的Android应用程序几次，不过F9运行当前被附加的程序几次以后，该应用程序会断在dvmDbgIsDebuggerConnected函数的开头或者结尾的位置即当前函数断点被触发啦，没事，不用理会，继续做下面的操作即可。



【2】.另开启一个命令行终端Terminate，使用 **jdb调试器** 连接到被调试附加百度加固的Android应用，jdb调试器连接成功以后，dvmDbgIsDebuggerConnected函数开头的断点会被触发，断在dvmDbgIsDebuggerConnected函数开头的位置，再**F9运行1次** 断在dvmDbgIsDebuggerConnected函数结尾的位置，此时dvmDbgIsDebuggerConnected函数的返回值R0的值为1（不做任何操作继续F9运行），**将第2次dvmDbgIsDebuggerConnected函数的返回值R0的值1修改为0** 即可手动绕过百度加固Debug.isDebuggerConnected的反调试（千万记得：**jdb附加之后，是修改dvmDbgIsDebuggerConnected函数第2次返回值1为0**）。