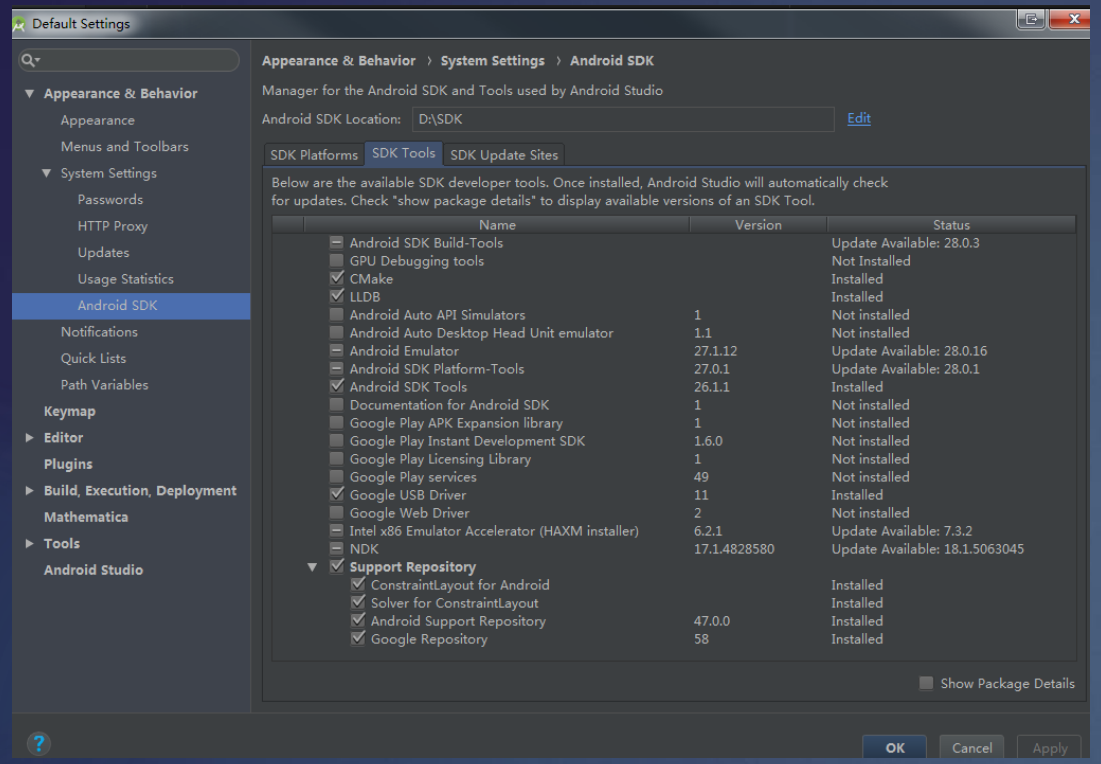
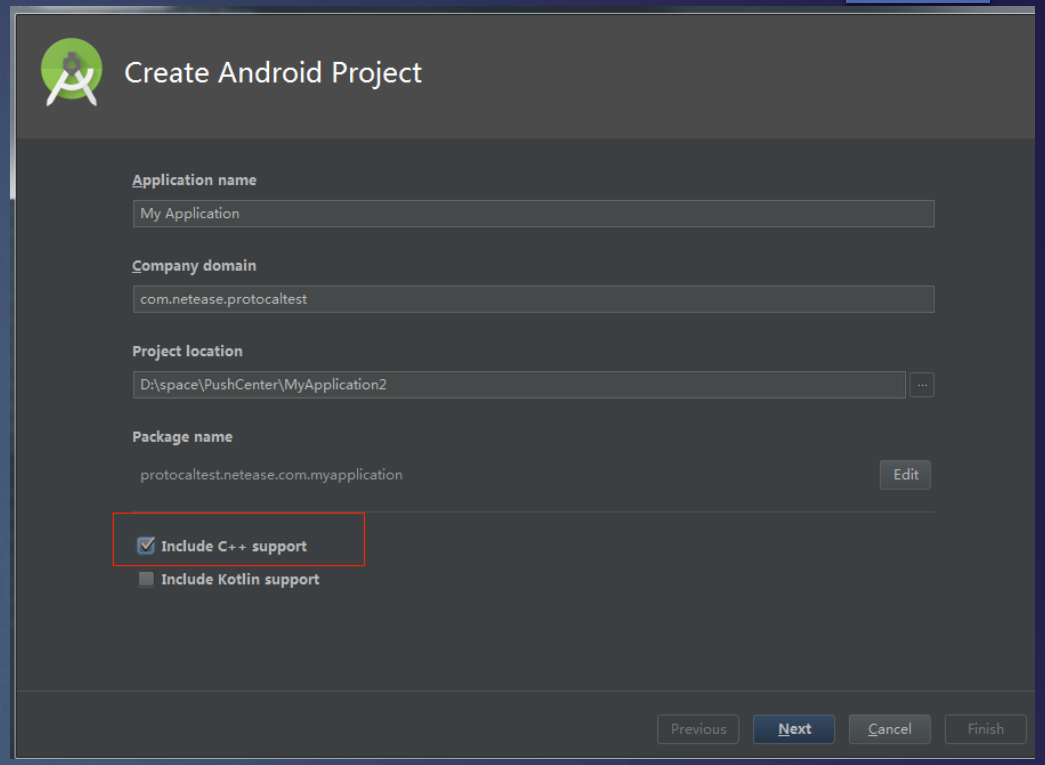
So编译与So分析

- 1、编译Gradle
- 2、编译Cmake
- 3、交叉编译工具和环境设置
- 4、SO的格式简介
- 5、使用IDA内存dump
- 6、使用IDA动态调试
- 7、加固和SO恢复

Android Studio 环境





编译Gradle

```
externalNativeBuild {
    cmake {
        abiFilters “x86”
        arguments ‘-DANDROID_PLATFORM=android-16’, ‘-DANDROID_STL=c++_static ’
        cppFlags "-std=gnu++11 -fno-exceptions -frtti -fblocks"
        cFlags "-Wno-unknown-warning-option -Wno-deprecated-register -Wno-mismatched-tags " +
                "-Wno-char-subscripts -Wno-infinite-recursion -Wno-gnu-designator " +
                "-Wno-unused-const-variable -Wno-unused-local-typedef -Wno-unused-private-field " +
                "-Wno-error=unused-variable -Wno-overloaded-virtual " +
                "-Werror -Wall" +
                "-Wsign-compare -Wtype-limits -Wuninitialized -Wclobbered -Wunused-but-set-parameter -Wempty-body" +
                "-Wno-error=conversion -Wno-error=sign-conversion -Werror=sign-compare -Wno-error=format -Wno-error=pointer-to-int-cast" +
                "-Wno-unused-parameter -Wno-missing-field-initializers "+
                "-fblocks"
    }
}
```

defaultConfig内设置
1、-DANDROID_STL=c++_static 使用的NDK提供的标准库 可以选择：库和库类型
库：libstdc++（默认系统运行时），提供较少功能。
以上库若不满足要求，使用stl库。
标准模板库（Standard Template Library，STL），是一个C++库，大量影响了c++标准库，但并非是其的一部分。
库类型：static或者share

2、flags 编译时的设置，使用gun编译，忽略一些检查和抛错。

```
externalNativeBuild {
    cmake {
        path "CMakeLists.txt"
    }
}
```

android{}内设置，指定编译脚本位置。

编译Cmake

```
1、cmake_minimum_required(VERSION 3.4.1)
2、set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${PROJECT_SOURCE_DIR}/../jniLibs/${ANDROID_ABI}) 设置编译后so输出的位置
3、include_directories() 导入头文件。
4、aux_source_directory(../../Bigger/Bigger/agent DIR_LIB_AGENT)    有时候想添加一整个文件夹， 就要使用这种方式
add_library( # Sets the name of the library.
             #native-lib
             bigger-lib

             # Sets the library as a shared library.
             SHARED
             ${DIR_LIB_AGENT}

)
5、#导入第三方依赖库
add_library(
      libcurl
      STATIC
      #SHARED
      IMPORTED
    )

#设置第三方依赖库的目录地址
set_target_properties(
      libcurl
      PROPERTIES IMPORTED_LOCATION
      #C:/Users/tianpeng/Desktop/use/9/libcurl.a
      C:/Users/tianpeng/Desktop/use/x86-9/libcurl.a
    )
6、find_library( # Sets the name of the path variable.
              log-lib

              # Specifies the name of the NDK library that
              # you want CMake to locate.
              log )
```

```
7、target_link_libraries( # Specifies the target library.
                          bigger-lib

                         # Links the target library to the log library
                          # included in the NDK.
                          ${log-lib}
                          ${z-lib}
                          #libssl
                          #libcrypto
                          libcurl
                          #${stl-lib}
                          )
```

这样配置发生了什么

GCC:由于GCC已成为GNU系统的官方编译器（包括GNU/Linux家族），它也成为编译与创建其他操作系统的主要编译器。

CLANG: Clang 是一个C、C++、Objective-C和Objective-C++编程语言的编译器前端。它采用了底层虚拟机（LLVM）作为其后端。它的目标是提供一个GNU编译器套装（GCC）的替代品。

执行buildproject 则会先把本工程的c文件编译成.o（目标文件），然后执行以下命令（只是个样例，并且不太规范，要把路径设置为相对路径）：

```
cmd.exe /C "cd . && D:\SDK\ndk-bundle\toolchains\llvm\prebuilt\windows-x86_64\bin\clang++.exe  --target=i686-none-linux-android --gcc-toolchain=D:/SDK/ndk-bundle/toolchains/x86-4.9/prebuilt/windows-x86_64 --sysroot=D:/SDK/ndk-bundle/sysroot -fPIC -isystem D:/SDK/ndk-bundle/sysroot/usr/include/i686-linux-android -D__ANDROID_API__=16 -g -DANDROID -ffunction-sections -funwind-tables -fstack-protector-strong -no-canonical-prefixes -mstackrealign -Wa,--noexecstack -Wformat -Werror=format-security -std=c++11 -std=gnu++11 -fno-exceptions -frtti -fblocks -O0 -fno-limit-debug-info  -Wl,--exclude-libs,libgcc.a -Wl,--exclude-libs,libatomic.a -nostdlib++ --sysroot D:/SDK/ndk-bundle/platforms/android-16/arch-x86 -Wl,--build-id -Wl,--warn-shared-textrel -Wl,--fatal-warnings -LD:/SDK/ndk-bundle/sources/cxx-stl/llvm-libc++/libs/x86 -Wl,--no-undefined -Wl,-z,noexecstack -Qunused-arguments -Wl,-z,relro -Wl,-z,now  -Wall -v -Wl,--no-warn-shared-textrel -shared -Wl,-soname,libbigger-lib.so -o D:\space\biggernew\jniLibs\x86\libbigger-lib.so CMakeFiles/bigger-lib.dir/Dles/bigger-lib.dir/D_/space/foundation/comm/autobuffer.cc.o CMakeFiles/bigger-lib.dir/D_/space/foundation/fileWriter/crypt/micro-ecc-master/uECC.c.o CMakeFiles/bigger-lib.dir/D_/space/thirdparty/include/json-parser/cJSON.c.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/system/src/error_code.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/filesystem/src/codecvt_error_category.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/filesystem/src/operations.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/filesystem/src/path.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/filesystem/src/path_traits.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/filesystem/src/portability.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/filesystem/src/unique_path.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/iostreams/src/file_descriptor.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/boost/libs/iostreams/src/mapped_file.cpp.o CMakeFiles/bigger-lib.dir/D_/space/foundation/comm/android/callstack.cc.o CMakeFiles/bigger-lib.dir/D_/space/foundation/comm/android/dumpcrash_stack.cc.o CMakeFiles/bigger-lib.dir/D_/space/foundation/comm/android/getprocessname.c.o CMakeFiles/bigger-lib.dir/D_/space/JNI/java2c.cpp.o  -llog -lz C:/Users/tianpeng/Desktop/use/9/libcurl.a -latomic -lm "D:/SDK/ndk-bundle/sources/cxx-stl/llvm-libc++/libs/x86/libc++_static.a" "D:/SDK/ndk-bundle/sources/cxx-stl/llvm-libc++/libs/x86/libc++abi.a" "D:/SDK/ndk-bundle/sources/cxx-stl/llvm-libc++/libs/x86/libandroid_support.a"
```

静态库 动态库 静态链接 动态链接

- 静态库： .a文件 。没有规定的格式，将c 代码编译成.o文件，打包在一起形成的。类似于将.java文件生成.class文件打成jar包。
- 动态库：.so文件。ELF文件格式之后分析。
- 静态链接：另外的一个库的代码集成进当前代码。类似于我们引入jar包到工程中。
- 动态链接：运行时链接。类似于我们运行时用classloader动态加载。

交叉编译工具和环境设置

独立工具链的典型用例是在 CC 环境变量中调用一个需要交叉编译器的开源库的 configure 脚本。如下：

编译一个成熟的第三方库的时候，使用其提供的configure文件进行编译，这是一个可执行的脚本，在linux下可以直接执行。它还会提供一些选项，用于执行脚本是作为参数输入。但是执行前需要配置好一些NDK交叉编译工具的环境变量。

独立工具链介绍和使用

- 1、编译工具位置：NDK的toolchains文件夹下，有各种平台对应的工具。
- 2、系统标头和库（system root）：位于$NDK/platforms/下，在目录下选择一个Android API，在该API下再选择一个对应的COU架构。
- 3、官方手册的使用样例：

```
export CC=“$NDK/toolchains/arm-linux-androideabi-4.8/prebuilt/ \
linux-x86/bin/arm-linux-androideabi-gcc-4.8 --sysroot=$SYSROOT”
$CC -o foo.o -c foo.c
```

使用config文件编译的话，使用之前可能还需要在电脑(linux系统)上安装 autoconf 、libtool、m4 三个软件

一个编译curl库的编译脚本例子：

```
unset IS_CROSS_COMPILE
unset IS_IOS
unset IS_ANDROID
unset IS_ARM_EMBEDDED
unset AOSP_FLAGS
unset AOSP_SYSROOT
unset AOSP_STL_INC
unset AOSP_STL_LIB
unset AOSP_BITS_INC

if [ -z "$AOSP_TOOLCHAIN_SUFFIX" ]; then
    AOSP_TOOLCHAIN_SUFFIX=4.9
Fi

if [ -z "$AOSP_API" ]; then
    AOSP_API="android-9"
Fi

export ANDROID_NDK_ROOT=/home/tp/Public/android-ndk-r13b

if [ "$#" -lt 1 ]; then
    THE_ARCH=armv7
else
    THE_ARCH=$(tr [A-Z] [a-z] <<< "$1")
Fi

case "$THE_ARCH" in
 x86)
    TOOLCHAIN_BASE="x86"
    TOOLNAME_BASE="i686-linux-android"
    AOSP_ABI="x86"
    AOSP_ARCH="arch-x86"
    AOSP_FLAGS="-march=i686 -mtune=intel -mssse3 -mfpmath=sse -funwind-tables -fexceptions -frtti"
    ;;
armv7a|armeabi-v7a)
    TOOLCHAIN_BASE="arm-linux-androideabi"
    TOOLNAME_BASE="arm-linux-androideabi"
    AOSP_ABI="armeabi-v7a"
    AOSP_ARCH="arch-arm"
    AOSP_FLAGS="-march=armv7-a -mthumb -mfpu=vfpv3-d16 -mfloat-abi=softfp -Wl,--fix-cortex-a8 -funwind-tables -
fexceptions -frtti"
    ;;
esac

AOSP_TOOLCHAIN_PATH="$ANDROID_NDK_ROOT/toolchains/$TOOLCHAIN_BASE-$AOSP_TOOLCHAIN_SUFFIX/prebuilt/$host/bin"
export AOSP_SYSROOT="$ANDROID_NDK_ROOT/platforms/$AOSP_API/$AOSP_ARCH"
export ANDROID_SYSROOT=$AOSP_SYSROOT
export CPP="$AOSP_TOOLCHAIN_PATH/$TOOLNAME_BASE-cpp --sysroot=$AOSP_SYSROOT"
export CC="$AOSP_TOOLCHAIN_PATH/$TOOLNAME_BASE-gcc --sysroot=$AOSP_SYSROOT"
export CXX="$AOSP_TOOLCHAIN_PATH/$TOOLNAME_BASE-g++ --sysroot=$AOSP_SYSROOT"
export CFLAGS="-pie -fPIE -Wno-error"
export LDFLAGS="-pie -fPIE -Wno-error"

./curl-7.59.0/configure \
                --with-ssl\
    --prefix=$PREFIX \
    --enable-static \
                --enable-shared \
    --host=$TOOLNAME_BASE
```

以上只是一种于官方手册比较一致的典型用法。

当然还有一些开源库不太一样，如 openssl。。。。

官方提供了一个脚本，在执行configure之前调用，主要还是设置之前那些文件的位置信息。但是是把路径export在了一个固定名称的变量里面，之后就是configure自己的事情了。

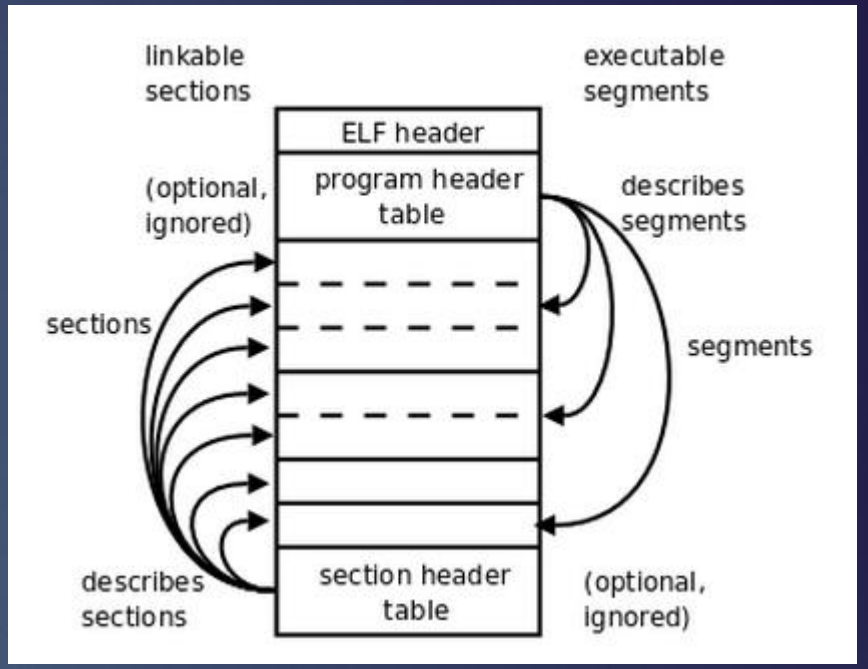
所以编译一个第三方开源库的时候，要先看执行configure需要的参数，配置独立工具的时候先看下有没有官方脚本，或者其他人的开源脚本，想找到一个通用脚本或者直接自己写可能不是一个好的选择。

可能碰到的问题及解决方案

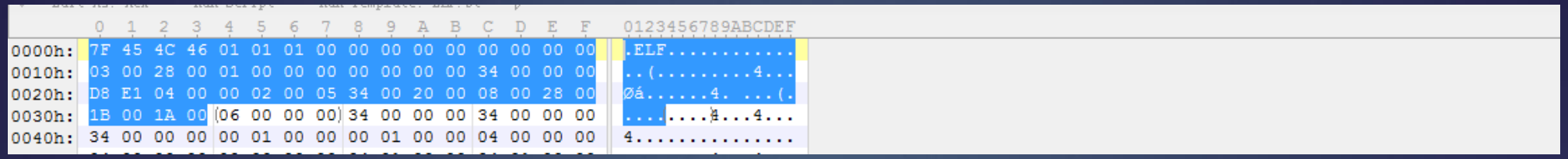
- 尝试使用较低版本的NDK ， NDK13 目前来看是一个不错的选择。
- 尝试不同android API级别。上面这两条主要是google好像看到那个库有问题，就直接删掉了没有替代品，有可能删的是NDK里的，也有可能删的的是手机系统里的。
- 使用config文件时三个软件的版本也有要求。。。。。
- 如果程序中使用了一些锁或者一些看上去比较高级的库提供的方法，gradlew中使用-DANDROID_STL=c++_static
- 静态链接和动态链接：动态链接会转换成地址调用 静态链接会转换成子程序调用，如果动态链接编译出的so使用起来有问题的话可以尝试下静态链接。
- 可以debug。
- 可以在c代码中打log。
- 直接运行studio会直接把so打到apk里面，在工程目录下可能找不到，需要再cmake里设置。
- 没遇到的问题，翻下官方手册，目前碰到的问题解决后再去看官方手册其实都有提到（主要是一开始看不太懂。。。）https://developer.android.com/ndk/guides/

一个正常的so（elf文件）格式

- 头 elf_header
- 程序执行所需 program_header_table
- 程序链接所需section_header_table
- https://blog.csdn.net/jiangwei0910410003/article/details/49336613



Struct elf_header



```
          0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F    0123456789ABCDEF
0000h: 7F 45 4C 46 01 01 01 00 00 00 00 00 00 00 00 00  .ELF............
0010h: 03 00 28 00 01 00 00 00 00 00 00 00 34 00 00 00  ..(.........4...
0020h: D8 E1 04 00 00 02 00 05 34 00 20 00 08 00 28 00  Øá......4. ...(.
0030h: 1B 00 1A 00 06 00 00 00 34 00 00 00 34 00 00 00  ........4...4...
0040h: 34 00 00 00 00 01 00 00 00 01 00 00 04 00 00 00  4...............
```

1、第一行打开了几个so内容都是一样的。 7F 45 4C 46 魔数 对应了ELF字符。后面01 01 01 00 00 是class（值为01 和02代表32位还是64位程序） 、data （编码是高位在前还是低位在前。基本都一样） 、version（固定01）、osabi、abi version（这里的abi似乎不对应so的abi， 打开x86和v7的so这一行也是一样的）。这一行主要起elf文件的标识作用。

2、第二行。开始出现so之间的差异。03 00标识type，代表这是一个so文件（还有其他的如exe等标识）。28 00 代表这是一个arm架构的ABI(X86对应的值是03)。4-7： version 01 固定。8-B :entry 入口地址，我们使用的so没有入口，是0不奇怪。C-F： 重要，是program_header_table的位置。

3、第三行。0-3： 重要，是section_header_table的位置。4-7 （保存与文件相关的，特定于处理器的标志。）不知道有什么用，相同abi架构看到的都一样。8-9头部大小。A-B,重要，程序头部表的单个表项大小，C-D,代表程序头部表数量。E-F 和第四行的0-1 对应section即节区表相应的值。

Program_header_table

1、根据头部信息，这里有8个个header，根据头部记录的header大小可以计算出header_table的大小。

2、table中，一个header的索引，有八个字段。

p_type

p_offset： 段相对于文件的索引地址

p_vaddr： 段在内存中的虚拟地址

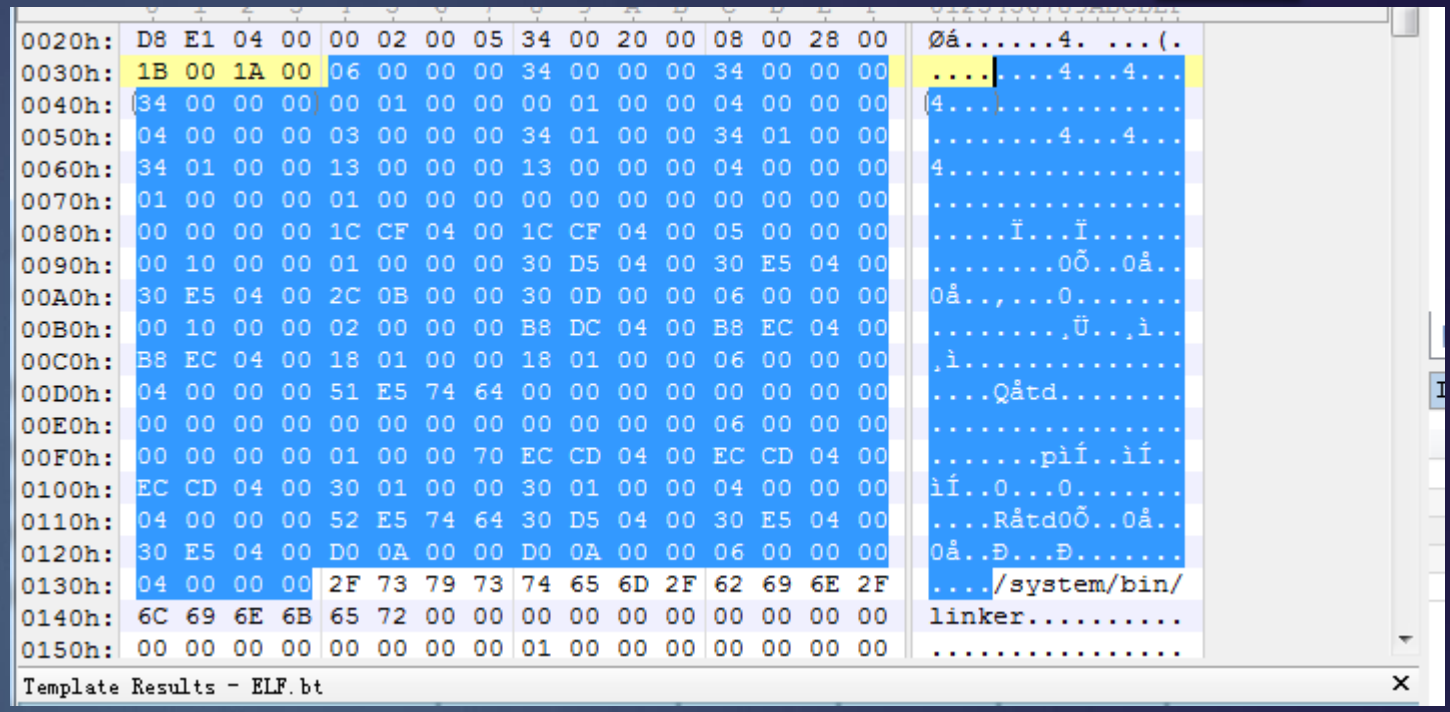
p_paddr： 段的物理地址

p_filesz： 段在文件中所占的长度

p_memsz： 段在内存中所占的长度

p_flage： 段相关标志(read、write、exec)

p_align： 字节对其,p_vaddr 和 p_offset 对 p_align 取模后应该相等



section_header_table

- **sh_name** 节区名称,此处是一个在名称节区的索引。
- **sh_flags** 同Program Header的p_flags
- **sh_addr** 节区索引地址
- **sh_offset** 节区相对于文件的偏移地址
- **sh_size** 节区的大小
- **sh_link** 此成员给出节区头部表索引链接。
- **sh_addralign**某些节区带有地址对齐约束。
- **sh_entsize**某些节区中包含固定大小的项目,如符号表。对于这类节区,此成员给出每个表项的长度字节数。 如果节区中并不包含固定长度表项的表格,此成员取值为0。

重要的节区

- **字符串表：**在一个ELF文件中通常拥有一个或以上的字符串表,即类型为 SHT_STRTAB 的节区,如: ELF Header 中 e_shstrndx 索引的**节区名称表(.shstrtab)、符号名称表(.dynstr)**等。**对于字符串的定义,是以NULL(\0)开头,以NULL结尾。**
- **符号表**: 指函数或者数据对象等。

  既然叫做表,那么也分为一个一个表项,其表项也有自己的结构定义:

  st_name 符号名称,给出的是一个在符号名称表(.dynstr)中的索引

  st_value 一般都是函数地址,或者是一个常量值

  st_size 从 st_value 地址开始,共占的长度大小

  st_info 用于标示此符号的属性,占一个字节(2个字),两个标示位,第一个标示位(低四位)标志作用域,第二个标示位(高四位)标示符号类型
- 代码段：**存放指令的节区(.text)**
- **过程链接表** .plt节区,其每个表项都是一段代码,作用是跳转至真实的函数地址
- **数据段**

  .data、.bss、.rodata都属于数据段。其中，

  .data 存放已初始化的全局变量、常量。

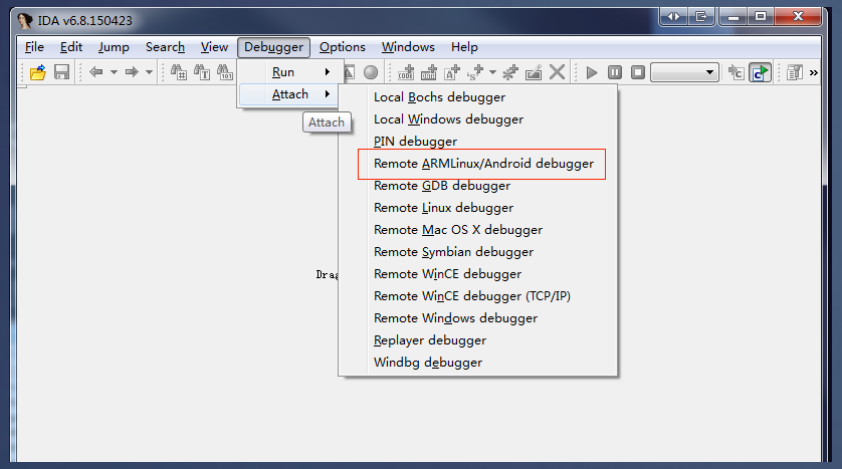
  .bss 存放未初始化的全局变量,所以此段数据均为0，仅作占位。

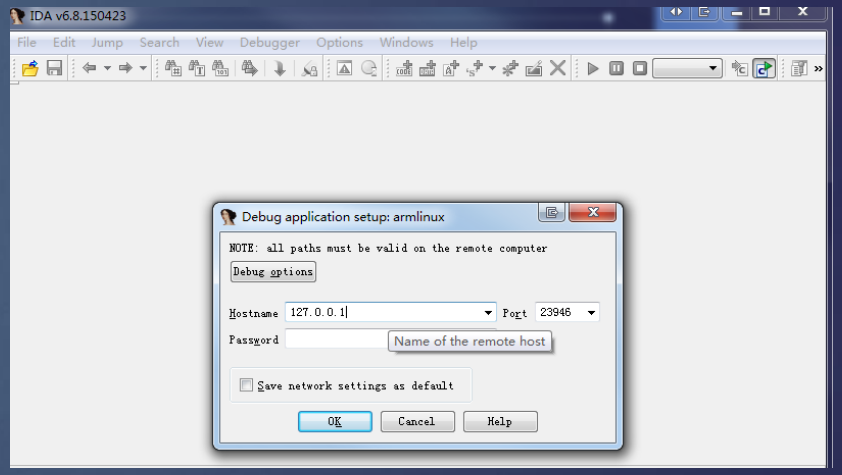
  .rodata 是只读数据段,此段的数据不可修改,存放常量。

- section的权限是没有规律的，比如第一个section的权限是只读，第二个是读写、第三个又是只读。如果在内存当中直接以这种形式存在，那么文件在执行的时候会造成权限控制难度加大，导致不必要的消耗。所以当我们将so文件链接到内存中时，存在的不是section，而是segment，每个segment可以看作是相同权限的section的集合。也就是说在内存当中一个segment对应N个section（N>=0），而这些section和segment的信息都会被保存在文件中。
- System.load的过程，是对so文件各个部分逐次mmap映射到内存的过程：
  1、读头 校验。
  2、读program header
  3、找到类型为 PT_LOAD 的 segment，load
  4、load so文件完毕，分配soinfo结构体空间。
  5、解析dynamic section，加载依赖的so，重定向。
  6、将so信息保存在soinfo中。
  7、调用初始化函数。可以通过链接选项-init或是给函数添加属性__attribute__((constructor))来指定 SO 的初始化函数。会先调用依赖so的初始化函数。之后还会调用jni_onload函数。

使用IDA内存dump

- 步骤一：IDA目录下找到android_server文件，push到手机某个目录下，然后切换到android_server所在目录下./android_server。
- 步骤二：执行完成android_server后会看到一个端口号。这是android_server监听的端口。执行adb forward tcp:端口号1 tcp:端口号2，意思是： PC上所有端口号1的通信数据将被重定向到手机端端口2的server上。这里IDA和server的端口号一样都是23946。
- 步骤三：





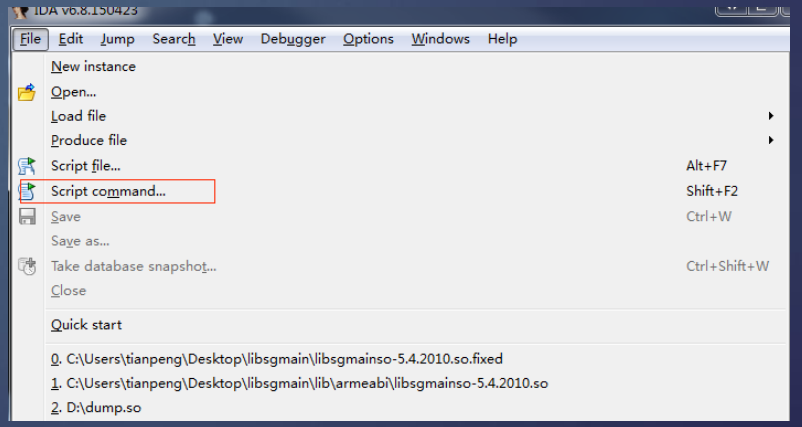
选择相应的程序attach

- 步骤四：cat 手机 proc/pid/maps 找到该应用占用内存的划分信息

```
b7402000-b7403000 rw-p 00020000 08:02 914        /system/lib/libm.so
b7403000-b740c000 r-xp 00000000 08:02 912        /system/lib/liblog.so
b740c000-b740d000 r--p 00008000 08:02 912        /system/lib/liblog.so
b740d000-b740e000 rw-p 00009000 08:02 912        /system/lib/liblog.so
b740e000-b7427000 r-xp 00000000 08:02 864        /system/lib/libcutils.so
b7427000-b7428000 r--p 00018000 08:02 864        /system/lib/libcutils.so
b7428000-b7429000 rw-p 00019000 08:02 864        /system/lib/libcutils.so
b7429000-b750f000 r-xp 00000000 08:02 853        /system/lib/libc.so
b7510000-b7513000 r--p 000e6000 08:02 853        /system/lib/libc.so
b7513000-b7516000 rw-p 000e9000 08:02 853        /system/lib/libc.so
b7520000-b755d000 r-xp 00000000 08:02 851        /system/lib/libbinder.so
b755e000-b7564000 r--p 0003d000 08:02 851        /system/lib/libbinder.so
b7564000-b7565000 rw-p 00043000 08:02 851        /system/lib/libbinder.so
b7565000-b76e9000 r-xp 00000000 08:02 829        /system/lib/libandroid_runtime.
so
b76e9000-b76f0000 r--p 00183000 08:02 829        /system/lib/libandroid_runtime.
so
b76f0000-b76f8000 rw-p 0018a000 08:02 829        /system/lib/libandroid_runtime.
so
b76ff000-b7700000 r-xp 00000000 00:00 0          [vdso]
root@x86:/proc/2147 # _
```

- 步骤五：执行如下IDA脚本dump出起始终止地址的内存：

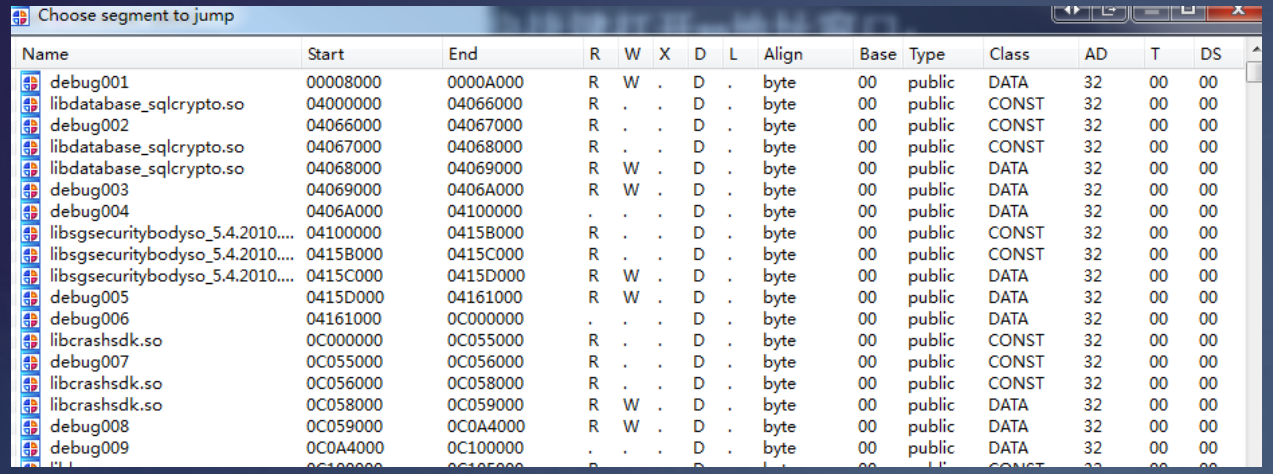
```
auto fp,begin,end,dexbyte
fp = fopen("D:\\dump.dex","wb");
begin = 0x0ccc2000;
end = 0x0ccc4000;
for(dexbyte = begin;dexbyte<end;dexbyte++)
    fputc(Byte(dexbyte),fp)
```



IDA脚本的编写可以参考《IDA+PRO权威指南》

使用IDA动态调试so

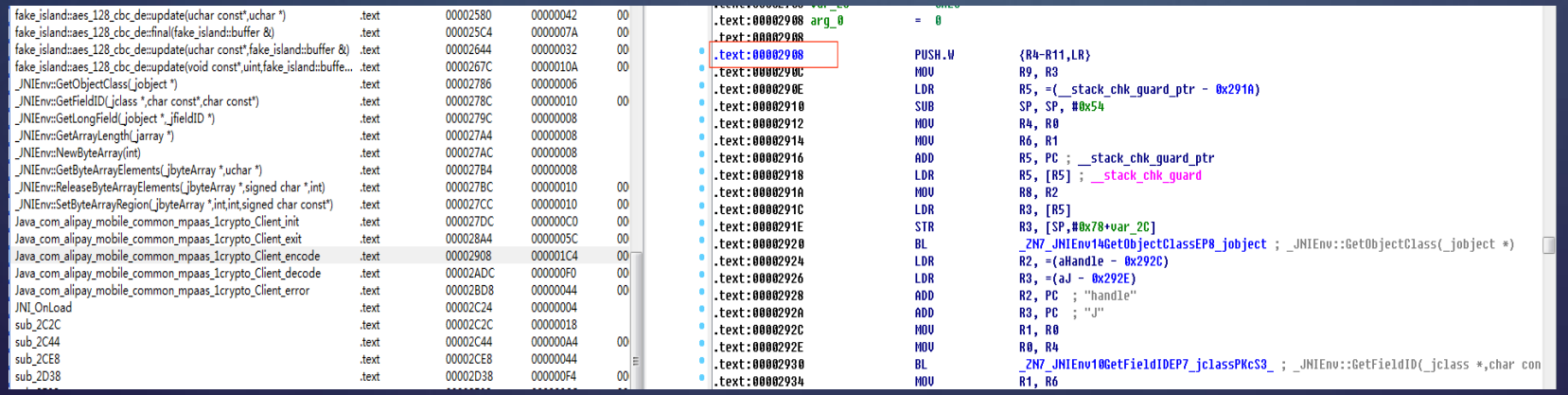
- 步骤一：Attach到目标进程的方法同上。使用ctrl+s快捷键打开so地址窗口：



Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	T	DS
debug001	00008000	0000A000	R	W	.	D	.	byte	00	public	DATA	32	00	00
libdatabase_sqlcrypto.so	04000000	04066000	R	.	.	D	.	byte	00	public	CONST	32	00	00
debug002	04066000	04067000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libdatabase_sqlcrypto.so	04067000	04068000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libdatabase_sqlcrypto.so	04068000	04069000	R	W	.	D	.	byte	00	public	DATA	32	00	00
debug003	04069000	0406A000	R	W	.	D	.	byte	00	public	DATA	32	00	00
debug004	0406A000	04100000	.	.	.	D	.	byte	00	public	DATA	32	00	00
libsgsecuritybodyso_5.4.2010....	04100000	0415B000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libsgsecuritybodyso_5.4.2010....	0415B000	0415C000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libsgsecuritybodyso_5.4.2010....	0415C000	0415D000	R	W	.	D	.	byte	00	public	DATA	32	00	00
debug005	0415D000	04161000	R	W	.	D	.	byte	00	public	DATA	32	00	00
debug006	04161000	0C000000	.	.	.	D	.	byte	00	public	DATA	32	00	00
libcrashsdk.so	0C000000	0C055000	R	.	.	D	.	byte	00	public	CONST	32	00	00
debug007	0C055000	0C056000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libcrashsdk.so	0C056000	0C058000	R	.	.	D	.	byte	00	public	CONST	32	00	00
libcrashsdk.so	0C058000	0C059000	R	W	.	D	.	byte	00	public	DATA	32	00	00
debug008	0C059000	0C0A4000	R	W	.	D	.	byte	00	public	DATA	32	00	00
debug009	0C0A4000	0C100000	.	.	.	D	.	byte	00	public	DATA	32	00	00

效果与proc/pid/maps一致。找到要调试的so start地址。

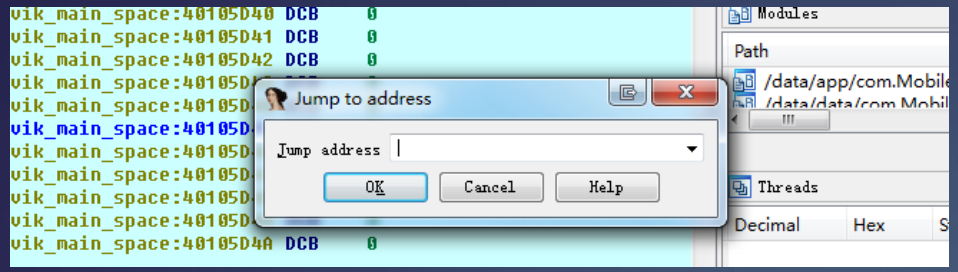
- 步骤二：新起一个窗口打开要调试的so，查看下断点的位置：



fake_island::aes_128_cbc_de::update(uchar const*,uchar *)	.text	00002580	00000042
fake_island::aes_128_cbc_de::final(fake_island::buffer &)	.text	000025C4	0000007A
fake_island::aes_128_cbc_de::update(uchar const*,fake_island::buffer &)	.text	00002644	00000032
fake_island::aes_128_cbc_de::update(void const*,uint,fake_island::buffe...	.text	0000267C	0000010A
_JNIEnv::GetObjectClass(_jobject *)	.text	00002786	00000006
_JNIEnv::GetFieldID(_jclass *,char const*,char const*)	.text	0000278C	00000010
_JNIEnv::GetLongField(_jobject *,_jfieldID *)	.text	0000279C	00000008
_JNIEnv::GetArrayLength(_jarray *)	.text	000027A4	00000008
_JNIEnv::NewByteArray(int)	.text	000027AC	00000008
_JNIEnv::GetByteArrayElements(_jbyteArray *,uchar *)	.text	000027B4	00000008
_JNIEnv::ReleaseByteArrayElements(_jbyteArray *,signed char *,int)	.text	000027BC	00000010
_JNIEnv::SetByteArrayRegion(_jbyteArray *,int,int,signed char const*)	.text	000027CC	00000010
Java_com_alipay_mobile_common_mpaas_1crypto_Client_init	.text	000027DC	000000C0
Java_com_alipay_mobile_common_mpaas_1crypto_Client_exit	.text	000028A4	0000005C
Java_com_alipay_mobile_common_mpaas_1crypto_Client_encode	.text	00002908	000001C4
Java_com_alipay_mobile_common_mpaas_1crypto_Client_decode	.text	00002ADC	000000F0
Java_com_alipay_mobile_common_mpaas_1crypto_Client_error	.text	00002BD8	00000044
JNI_OnLoad	.text	00002C24	00000004
sub_2C2C	.text	00002C2C	00000018
sub_2C44	.text	00002C44	000000A4
sub_2CE8	.text	00002CE8	00000044
sub_2D38	.text	00002D38	000000F4

```
.text:00002908 arg_0           =  0
.text:00002908
.text:00002908                 PUSH.W          {R4-R11,LR}
.text:0000290C                 MOV             R9, R3
.text:0000290E                 LDR             R5, =(__stack_chk_guard_ptr - 0x291A)
.text:00002910                 SUB             SP, SP, #0x54
.text:00002912                 MOV             R4, R0
.text:00002914                 MOV             R6, R1
.text:00002916                 ADD             R5, PC ; __stack_chk_guard_ptr
.text:00002918                 LDR             R5, [R5] ; __stack_chk_guard
.text:0000291A                 MOV             R8, R2
.text:0000291C                 LDR             R3, [R5]
.text:0000291E                 STR             R3, [SP,#0x78+var_2C]
.text:00002920                 BL              _ZN7_JNIEnv14GetObjectClassEP8_jobject ; _JNIEnv::GetObjectClass(_jobject *)
.text:00002924                 LDR             R2, =(aHandle - 0x292C)
.text:00002926                 LDR             R3, =(aJ - 0x292E)
.text:00002928                 ADD             R2, PC  ; "handle"
.text:0000292A                 ADD             R3, PC  ; "J"
.text:0000292C                 MOV             R1, R0
.text:0000292E                 MOV             R0, R4
.text:00002930                 BL              _ZN7_JNIEnv10GetFieldIDEP7_jclassPKcS3_ ; _JNIEnv::GetFieldID(_jclass *,char con
.text:00002934                 MOV             R1, R6
```

- 步骤三：在调试窗口中，按G ：



输入调试要下断点的位置，位置为：调试窗口中的so的start地址（内存起始地址） + so分析窗口中断点的位置（代码相对起始地址的位置）

- 步骤四：点击ok跳转到位置后，按F2就下好了断点，可以调试了。

加固

加固技术经过几代的发展由混淆→加壳→指令抽离→指令转换（vmp）

1、加壳。

2、指令抽离：对某个函数进行加密，破坏其在dex文件中的结构，使用之前再修复该方法或者类。

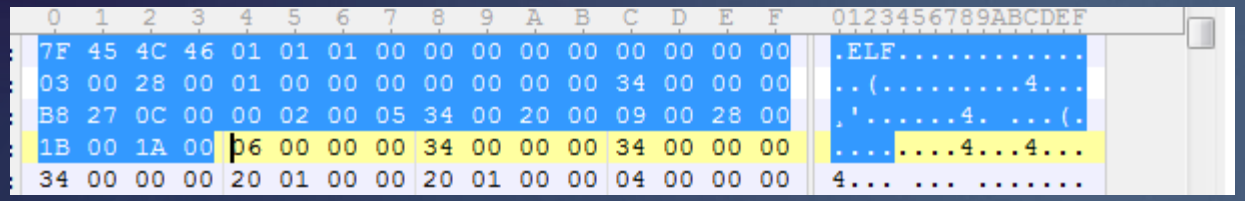
3、vmp。VMP（虚拟软件保护技术）大概思路就是自定义一套虚拟机指令和对应的解释器，并将标准的指令转换成自己的指令，然后由解释器将自己的指令给对应的解释器。由于兼容性和效率等问题，所以VMP一般只用于关键函数。
https://www.zhihu.com/question/51585199

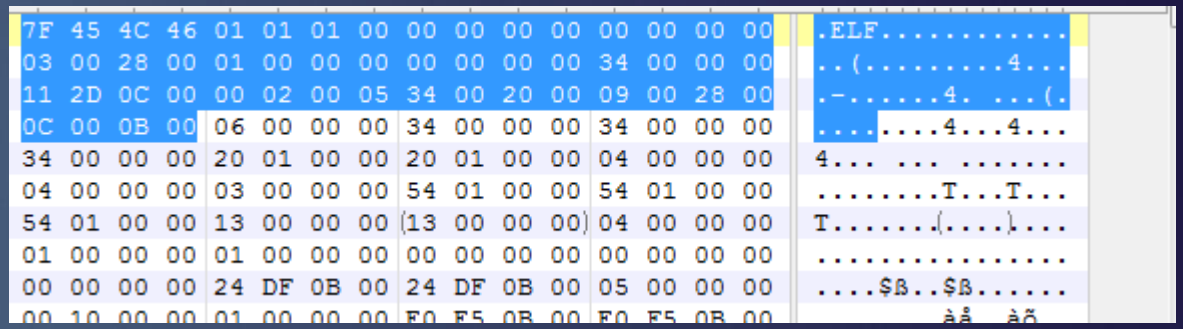
可以使用内存dump，Hook虚拟机加载流程。

Vmp的入门案例：https://bbs.pediy.com/thread-221270.htm

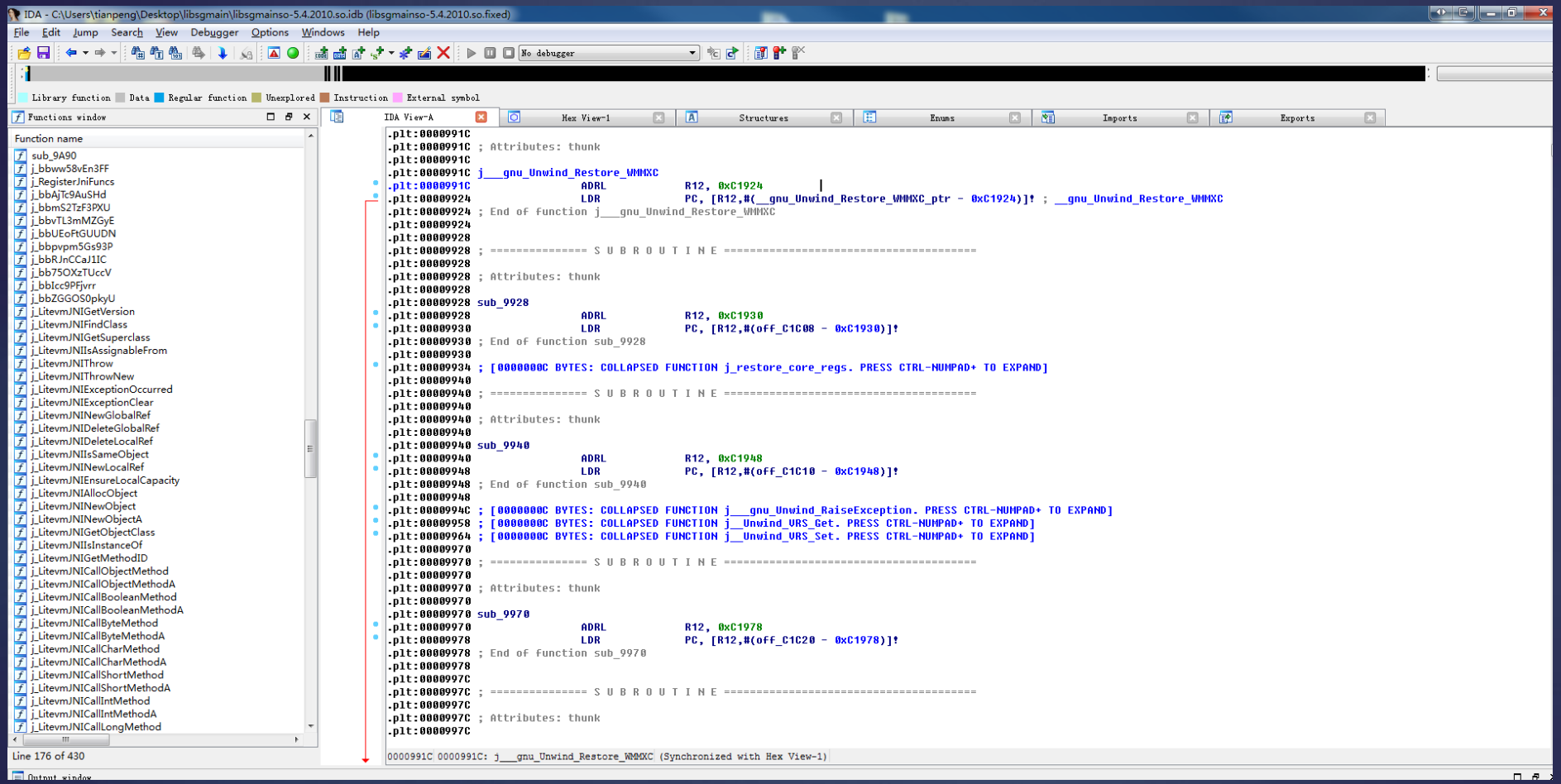
So更改

- 使用0l0editor
- 看雪论坛提供的so修复工具
- 修复前后so对比





加固方案Vmp


IDA - C:\Users\tianpeng\Desktop\libsgmain\libsgmainso-5.4.2010.so.idb (libsgmainso-5.4.2010.so.fixed)
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Data Regular function Unexplored Instruction External symbol
Functions window
Function name
sub_9A90
j_bbww58vEn3FF
j_RegisterJniFuncs
j_bbAjTc9AuSHd
j_bbmS2TzF3PXU
j_bbvTL3mMZGyE
j_bbUEoFtGUUDN
j_bbpvpm5Gs93P
j_bbRJnCCaJ1IC
j_bb75OXzTUccV
j_bbIcc9PFjvrr
j_bbZGGOS0pkyU
j_LitevmJNIGetVersion
j_LitevmJNIFindClass
j_LitevmJNIGetSuperclass
j_LitevmJNIIsAssignableFrom
j_LitevmJNIThrow
j_LitevmJNIThrowNew
j_LitevmJNIExceptionOccurred
j_LitevmJNIExceptionClear
j_LitevmJNINewGlobalRef
j_LitevmJNIDeleteGlobalRef
j_LitevmJNIDeleteLocalRef
j_LitevmJNIIsSameObject
j_LitevmJNINewLocalRef
j_LitevmJNIEnsureLocalCapacity
j_LitevmJNIAllocObject
j_LitevmJNINewObject
j_LitevmJNINewObjectA
j_LitevmJNIGetObjectClass
j_LitevmJNIIsInstanceOf
j_LitevmJNIGetMethodID
j_LitevmJNICallObjectMethod
j_LitevmJNICallObjectMethodA
j_LitevmJNICallBooleanMethod
j_LitevmJNICallBooleanMethodA
j_LitevmJNICallByteMethod
j_LitevmJNICallByteMethodA
j_LitevmJNICallCharMethod
j_LitevmJNICallCharMethodA
j_LitevmJNICallShortMethod
j_LitevmJNICallShortMethodA
j_LitevmJNICallIntMethod
j_LitevmJNICallIntMethodA
j_LitevmJNICallLongMethod
Line 176 of 430
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
.plt:0000991C
.plt:0000991C ; Attributes: thunk
.plt:0000991C
.plt:0000991C j___gnu_Unwind_Restore_WMMXC
.plt:0000991C ADRL R12, 0xC1924
.plt:00009924 LDR PC, [R12,#(__gnu_Unwind_Restore_WMMXC_ptr - 0xC1924)]! ; __gnu_Unwind_Restore_WMMXC
.plt:00009924 ; End of function j___gnu_Unwind_Restore_WMMXC
.plt:00009924
.plt:00009928
.plt:00009928 ; =============== S U B R O U T I N E =======================================
.plt:00009928
.plt:00009928 ; Attributes: thunk
.plt:00009928
.plt:00009928 sub_9928
.plt:00009928 ADRL R12, 0xC1930
.plt:00009930 LDR PC, [R12,#(off_C1C08 - 0xC1930)]!
.plt:00009930 ; End of function sub_9928
.plt:00009930
.plt:00009934 ; [0000000C BYTES: COLLAPSED FUNCTION j_restore_core_regs. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:00009940
.plt:00009940 ; =============== S U B R O U T I N E =======================================
.plt:00009940
.plt:00009940 ; Attributes: thunk
.plt:00009940
.plt:00009940 sub_9940
.plt:00009940 ADRL R12, 0xC1948
.plt:00009948 LDR PC, [R12,#(off_C1C10 - 0xC1948)]!
.plt:00009948 ; End of function sub_9940
.plt:00009948
.plt:0000994C ; [0000000C BYTES: COLLAPSED FUNCTION j___gnu_Unwind_RaiseException. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:00009958 ; [0000000C BYTES: COLLAPSED FUNCTION j__Unwind_VRS_Get. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:00009964 ; [0000000C BYTES: COLLAPSED FUNCTION j__Unwind_VRS_Set. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:00009970
.plt:00009970 ; =============== S U B R O U T I N E =======================================
.plt:00009970
.plt:00009970 ; Attributes: thunk
.plt:00009970
.plt:00009970 sub_9970
.plt:00009970 ADRL R12, 0xC1978
.plt:00009978 LDR PC, [R12,#(off_C1C20 - 0xC1978)]!
.plt:00009978 ; End of function sub_9970
.plt:00009978
.plt:0000997C
.plt:0000997C ; =============== S U B R O U T I N E =======================================
.plt:0000997C
.plt:0000997C ; Attributes: thunk
.plt:0000997C
0000991C 0000991C: j___gnu_Unwind_Restore_WMMXC (Synchronized with Hex View-1)
Output window

1、与传统的加壳工具不同，不是简单的把目标进行压缩、内存解压运行，而是修改目标源码，让目标的部分指令在vmp创建的虚拟环境下运行，虚拟环境中无操作数比较指令、条件跳转和无条件跳转指令。

2.被修改替换的目标指令最终形成的字节码有前后相关性，即你改变其他任意一个字节会影响到所有被vmp虚拟化的指令

3.vmp的虚拟机其实是一个字节码解释器，循环的读取指令并执行，并且只有一个入口和一个出口

4.虚假跳转和垃圾指令，vmp会使用大量的虚拟跳转和垃圾指令将原有简单的代码变得复杂

5.vmp是基于堆栈的虚拟机，虚拟机指令不是显示的读取参数，而是把要使用的参数压入堆栈，而后直接从堆栈中读取

6、vmp指令

1)算数运算和移位运算

2)堆栈操作

3)系统相关

4)逻辑运算，这个最复杂，vmp中只有一个逻辑运算指令nor,它可以模拟not and or xor 四个逻辑运算指令

7、vmp寄存器轮转

mvp将所有的寄存器都放在一个堆栈的结构vm_context中,结构中的每一项代码一个寄存器或临时变量

在程序运行过程中,vm_context结构中保存的寄存器不是固定的，每当执行完一个操作或一个指令结构中的项与真实寄存器之间的映射关系会发生变化，就像一个齿轮随机的转动了一下,转动过后原有的映射关系全部改变了

8、字节码加密和随机效验

随机效验比较牛B,vmp会在编译好的字节码中加入自己的一些指令(专属于vmp自动的指令),每一次执行都会对一段代码随机生成hash值,然后与另一个随机数相加,vmp要求相加的结果必须为0,否则会出错.

JNI 和 函数重命名

```
#define TARGET_CLASS "com/common/push/util/AESCryptor “


#define TARGET_CRYPT "crypt"
#define TARGET_CRYPT_SIG "([BJI)[B"
#define TARGET_READ "read"
#define TARGET_READ_SIG "(Ljava/lang/String;J)[B"


static const JNINativeMethod gMethods[] = { { TARGET_CRYPT, TARGET_CRYPT_SIG,
      (void*) android_native_aes }, { TARGET_READ, TARGET_READ_SIG,
      (void*) android_native_read } };

JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
   JNIEnv* env = NULL;
   if ((*vm)->GetEnv(vm, (void**) &env, JNI_VERSION_1_4) != JNI_OK) {
      return -1;
   }

   jclass clazz = (*env)->FindClass(env, TARGET_CLASS);
   if (!clazz) {
      return -1;
   }
   //这里就是关键了，把本地函数和一个java类方法关联起来。不管之前是否关联过，一律把之前的替换掉！
   if ((*env)->RegisterNatives(env, clazz, gMethods,
         sizeof(gMethods) / sizeof(gMethods[0])) != JNI_OK) {
      return -1;
   }

   return JNI_VERSION_1_4;
}
```

反调试

自定义init方法或在jni_onload 里调用ptrace(PTRACE_TRACEME, 0, 0, 0);

使用Cydia Substrate hook c层函数
修改本地so跳过执行

O-LLVM混淆native代码

- https://blog.csdn.net/skylin19840101/article/details/79992022