周哥教 IT 软件自保护与实战

视频课地址：https://ke.qq.com/course/406186?tuin=a71606

目录

为什么要保护程序？

逆向
直接分析获取源代码，C,CPP,JAVA 等程序
破解
轻易绕过正版验证
调试
获取算法逻辑
提纲
OLLVM 混淆
sub
fla
bcf
花指令混淆
加壳：UPX，VMP 等
Android apk 和 so 加壳思路
动态反调试
等等

一， OLLVM

OLLVM

OLLVM（Obfuscator-LLVM）是瑞士西北应用科技大学安全实验室于 2010 年 6 月份发起的一个项目，该项目旨在提供一套开源的针对 LLVM 的代码混淆工具，增加逆向工程的难度。
LLVM 命名最早源自于底层虚拟机(Low Level Virtual Machine)的缩写，目前 LLVM 就是该项目的全称。LLVM 核心库提供了与编译器相关的支持，可以作为多种语言编译器的后台使用。能够进行程序语言的编译期优化、链接优化、在线编译优化、代码生成。LLVM 的项目是一个模块化和可重复使用的编译器和工具技术的集合。LLVM 是伊利诺伊大学的一个研究项目，提供一个现代化的，基于 SSA（Static Single Assignment，静态单赋值，一种 IR(中间表示代码)，保证每个变量只被赋值一次，帮助简化编译器的优化算法）的编译策略，能够同时支持静态和动态的任意编程语言的编译目标。自此， LLVM 已成为主干项目，由不同的子项目组成，其中许多是正在生产中使用的各种商业和开源的项目，被广泛用于学术研究。
OLLVM 适用 LLVM 支持的所有语言（C, C++, Objective-C, Ada 和 Fortran）和目标平台（x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC, Alpha, CellSPU,MIPS, MSP430, SystemZ, 和 XCore）。

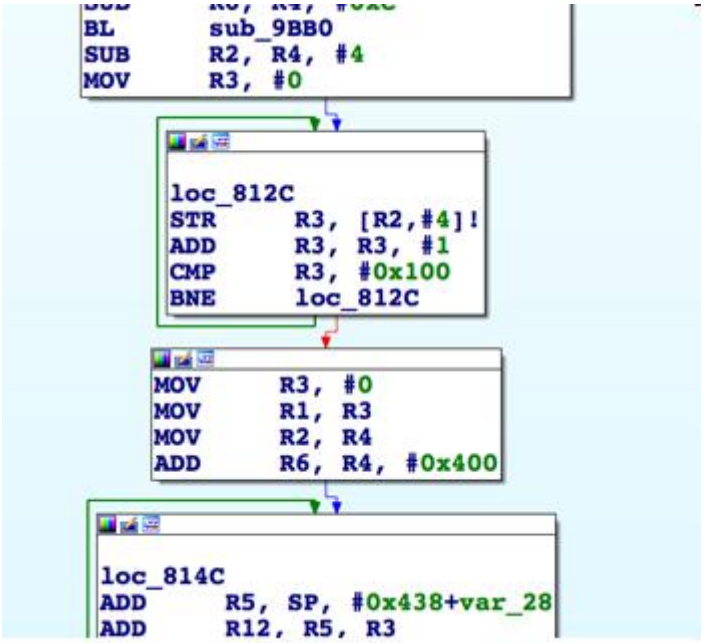
https://github.com/obfuscator-llvm/obfuscator

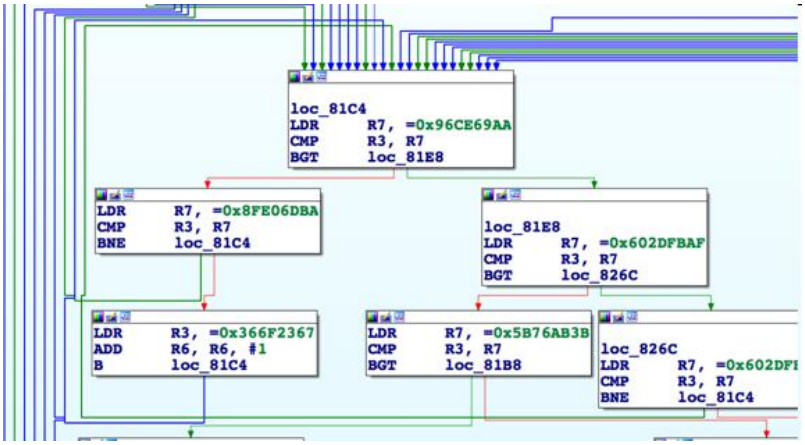
OLLVM 混淆三种模式

指令替换模式（Instructions Substitution）：主要是将正常的运算操作（+，-，*，/，&，|等）替换成功能相等但表述更复杂的形式

控制流平展模式（Control Flow Flattening）：可以完全改变程序原本的控制流图,经 FLA(CFF)模式混淆后，程序的执行流程已经被打乱，出现许多代码分支

控制流伪造模式（Bogus Control Flow）：也是对程序的控制流做操作，与 CFF 不同的是，BCF 模式会在原代码块的前后随机插入新的不确定的代码块，然后新代码块再通过条件判断跳转到原代码块中。甚至原代码块可能会被克隆并插入随机的垃圾指令。同一份代码多次做 BCF 模式的混淆时，得到的是不同的混淆效果。原本简单的 if-else 分支代码变得异常复杂，加大了逆向分析的难度





OLLVM 应用实战（1）-编译 OLLVM

0，环境
虚拟机 UBUNTU 16.04 4 核，4G 内存，磁盘 20G（编译需要 2G 磁盘空间），耗时 1 个小时左右
1，安装 git 和 cmake:
sudo apt-get install git
sudo apt-get install cmake
sudo apt-get install build-essential
2，下载和编译 ollvm
cd ~/
mkdir workspace
cd workspace
git clone -b llvm-4.0 https://github.com/obfuscator-llvm/obfuscator.git （网络不稳定，换网）
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ../obfuscator/ ，如果出错，换下面命令
cmake -DCMAKE_BUILD_TYPE=Release -DLLVM_INCLUDE_TESTS=OFF ../obfuscator/
make －j4(并行编译任务，一般为 CPU 核数的双倍，)
cd bin/
OLLVM 应用实战（1）-混淆程序
export PATH=~/workspace/build/bin/:$PATH
普通方式：
gcc hello.c －o hello
gcc －S hello.c
指令替换:
clang hello.c -mllvm -sub -o hello
clang hello.c -emit-llvm -mllvm -sub -S -o hellosub.ll
控制流平展:
clang hello.c -mllvm -fla -o hello
clang hello.c -emit-llvm -mllvm -fla -S -o hellofla.ll
控制流伪造
clang hello.c -mllvm -bcf -o hello
clang hello.c -emit-llvm -mllvm -bcf -S -o hellobcf.ll
综合应用三种混淆模式
clang hello.c -mllvm -bcf -mllvm －fla －mllvm －sub -o hello

```
#include <stdio.h>
int main() {
        int a=5;

         if(a<10){
```

```
            a++;
        }else{

        }

        printf("hello ollvm\n");
        return 0;
}
```

OLLVM 应用实战（2）-开发 ndk 程序

许多安全公司都会在保护 IOS 或安卓 APP 时用到 OLLVM 技术。如顶象加固、网易加固。利用 OLLVM 混淆 Android Native 代码：

1，安装 ndk

```
sudo apt-get unzip
cd ~/workspace
mkdir AndroidNDK
cd AndroidNDK
wget https://dl.google.com/android/repository/android-ndk-r12b-linux-x86_64.zip
unzip android-ndk-r12b-linux-x86_64.zip
android-ndk-r12b/ndk-build
```

2，编写和创建 jni 工程：

```
            mkdir jni
            cd jni
//hello-jni.c
#include <jni.h>
jint JNICALL
Java_com_test_Test(JNIEnv *env, jclass t,jint val) {
        int key = 10;
        return val^key;
}
            //Android.mk:
 LOCAL_PATH := $(call my-dir)
 include $(CLEAR_VARS)
 LOCAL_MODULE        := hello-jni
 LOCAL_SRC_FILES := hello-jni.c
 include $(BUILD_SHARED_LIBRARY)
```

3，ndk-build 编译 so，并 IDA 反汇编

```
../AndroidNDK/android-ndk-r12b/ndk-build                    NDK_PROJECT_PATH=.
```

APP_BUILD_SCRIPT=./android.mk
OLLVM 应用实战（2）-基于 ollvm 混淆
4， ollvm 和 ndk 关联
https://github.com/Fuzion24/AndroidObfuscation-NDK/tree/master/prebuilt
tar －xjvf android-ndk64-r10-linux-x86_64-obfuscator.tar.bz2
~/workspace/AndroidNDK/android-ndk-r12b/toolchains/arm-linux-androideabi-clang3.4
复制 config.mk 和 setup.mk 到
~/workspace/AndroidNDK/android-ndk-r12b/toolchains/arm-linux-androideabi-clang3.4-obfuscator/

编辑 arm-linux-androideabi-clang3.4-obfuscator/setup.mk 文件，将里面的：
TARGET_CC := $(LLVM_TOOLCHAIN_PREFIX)clang$(HOST_EXEEXT)
TARGET_CXX := $(LLVM_TOOLCHAIN_PREFIX)clang++$(HOST_EXEEXT)
修改为：
LLVM_TOOLCHAIN_PATH :=你的 ollvm 的/build/bin/
TARGET_CC := $(LLVM_TOOLCHAIN_PATH)clang$(HOST_EXEEXT)
TARGET_CXX := $(LLVM_TOOLCHAIN_PATH)clang++$(HOST_EXEEXT)
5，修改 Android.mk
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := hello-jni
LOCAL_SRC_FILES := hello-jni.c
include $(BUILD_SHARED_LIBRARY)
LOCAL_CFLAGS := -mllvm -bcf -mllvm -boguscf-prob=100 -mllvm -boguscf-loop=1 -mllvm -sub -mllvm -fla -mllvm -perFLA=100
6，ndk-build NDK_PROJECT_PATH=. APP_BUILD_SCRIPT=./android.mk
sudo ../AndroidNDK/android-ndk-r14b/ndk-build NDK_PROJECT_PATH=. NDK_APPLICATION_MK=./application.mk APP_BUILD_SCRIPT=./android.mk
Linux 和 android 上。Win 上类似，举一反三
源代码分析

二，花指令

花指令（1）

花指令是程序中的无用指令或者垃圾指令，故意干扰各种反汇编静态分析工具，但是程序不受任何影响，缺少了它也能正常运行。加花指令后，IDA Pro 等分析工具对程序静态反汇编时，往往会出现错误或者遭到破坏，加大逆向静态分析的难度，从而隐藏自身的程序结构和算法，从而较好的保护自己。
开发专门的脚本来处理

花指令（2）

1，call 在 0A04B0D7 的指令不会被反汇编，因为它做了 0A04B0D6 处 5 字节指令的一部分（call 迷惑了反汇编器，从 0A04B0D6 位置开发反汇编）。在 0A04B0D6 执行 U，然后在 0A04B0D7 执行 C

jmp 到自己的指令也有问题，必须去掉 0A04B0DB 的定义（U），重新定义 0A04B0DC 处的指令（C）。最后执行的是 jmp eax（对抗静态反汇编，运行时才计算）

```
LOAD:0A04B0D1                call    near ptr loc_A04B0D6+1
LOAD:0A04B0D6
LOAD:0A04B0D6 loc_A04B0D6:                           ; CODE XREF: start+11↓p
LOAD:0A04B0D6                mov     dword ptr [eax-73h], 0FFEB0A40h
LOAD:0A04B0D6 start          endp
LOAD:0A04B0D6
LOAD:0A04B0DD
LOAD:0A04B0DD loc_A04B0DD:                           ; CODE XREF: LOAD:0A04B14C↓j
LOAD:0A04B0DD                loopne  loc_A04B06F
LOAD:0A04B0DF                mov     dword ptr [eax+56h], 5CDAB950h
LOAD:0A04B0E6                iret
LOAD:0A04B0E6 ;---------------------------------------------------------------------------
LOAD:0A04B0E7                db 47h
LOAD:0A04B0E8                db 31h, 0FFh, 66h
LOAD:0A04B0EB ;---------------------------------------------------------------------------
LOAD:0A04B0EB
LOAD:0A04B0EB loc_A04B0EB:                           ; CODE XREF: LOAD:0A04B098↑j
LOAD:0A04B0EB                mov     edi, 0C7810D98h
```

```
LOAD:0A04B0D1                call    loc_A04B0D7
LOAD:0A04B0D1 ;---------------------------------------------------------------------------
LOAD:0A04B0D6                db 0C7h ;
LOAD:0A04B0D7 ;---------------------------------------------------------------------------
LOAD:0A04B0D7
LOAD:0A04B0D7 loc_A04B0D7:                           ; CODE XREF: start+11↑p
LOAD:0A04B0D7                pop     eax
LOAD:0A04B0D8                lea     eax, [eax+0Ah]
LOAD:0A04B0DB
LOAD:0A04B0DB loc_A04B0DB:                           ; CODE XREF: start:loc_A04B0DB↑j
LOAD:0A04B0DB                jmp     short near ptr loc_A04B0DB+1
LOAD:0A04B0DB start          endp
```

```
LOAD:0A04B0D7                pop     eax
LOAD:0A04B0D8                lea     eax, [eax+0Ah]
LOAD:0A04B0D8 ; ---------------------------------------------------------------------------
LOAD:0A04B0DB                db 0EBh ; d
LOAD:0A04B0DC ; ---------------------------------------------------------------------------
LOAD:0A04B0DC                jmp     eax
LOAD:0A04B0DC start          endp
```

```
.text:00401000               xor     eax, eax
.text:00401002               jz      short near ptr loc_401009+1
.text:00401004               mov     ebx, [eax]
.text:00401006               mov     [ecx-4], ebx
.text:00401009
.text:00401009 loc_401009:                           ; CODE XREF: .text:00401002↑j
.text:00401009               call    near ptr 0ADFEFFC6h
.text:0040100E               ficom   word ptr [eax+59h]
```

```
.text:00401000               xor     eax, eax
.text:00401002               jz      short loc_40100A
.text:00401004               mov     ebx, [eax]
.text:00401006               mov     [ecx-4], ebx
.text:00401006 ; ---------------------------------------------------------------------------
.text:00401009               db 0E8h ; F
.text:0040100A ; ---------------------------------------------------------------------------
.text:0040100A
.text:0040100A loc_40100A:                           ; CODE XREF: .text:00401002↑j
.text:0040100A               mov     eax, 0DEADBEEFh
.text:0040100F               push    eax
.text:00401010               pop     ecx
```

xor eax，eax

jz short near ptr loc_40100A
构成了一个绝对跳转（在 40100A 前的都不会执行）。
但反汇编编译器没有识别，继续对 jz 后面的指令比如 0401009 处的进行反汇编，出错。

添加花指令实战 1-jz+jnz

```
int main(void)
{
	_asm
	{
		jz label;
		jnz label;
		_emit 0e8h;call 机器码，永远不会执行，一个字节，不完整，0e8h B8 08 00 00
label:
		mov eax, 8;影响到这条指令识别，B8 08 00 00 00
		xor eax, 7; 83 F0 07
	}
	  return 0; //33 C0, xor		eax,eax
}
```

emit 伪指令，在当前位置直接插入数据（指令），一般用来插入汇编里面没有的特殊指令，和 db,dw 效果相同。目的：
编译器不认识的指令，拆成机器码来写。
插入垃圾字节来反逆向，又称花指令。
加花指令和不加花指令汇编对比

添加花指令实战 2-call+ret

call 指令：首先把自身所在位置的下一条指令的地址压栈，然后 jmp 到 call 的地址
ret 指令：jmp 到 call 指令压栈的地址
1.call 一个地址，在 call 下面随便写入花指令并记住花指令字节长度
2.在 call 里面，也就是函数里面，首先 pop 出压入的地址，然后把这个地址加上花指令占用的字节数，再重新 push 进栈，然后就 ret（跳到了花指令的下一条指令）

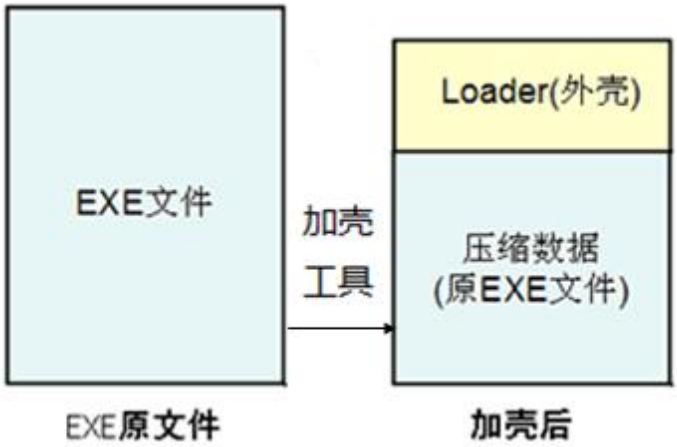
```
int main(){
	_asm{
		call L1
		_EMIT 0xEA//jmp 机器码，没有地址，不完整，永远不会执行，1 个字节
		jmp L2//call 结束以后执行到这里
L1:
		pop ebx;压栈地址弹出
		inc ebx;压栈地址+1，绕过花指令
```

```
            push ebx//返回地址完成加 1，绕过了花指令
            mov eax,0x1;设置返回值
            ret
L2:
        }
        printf("hello world\n");
}
```

三，加壳

壳的概念

壳是指在一个程序的外面再包裹上另外一段代码，保护里面的代码不被非法修改或反编译的程序。它们一般都是先于程序运行，拿到控制权，然后完成它们保护软件的任务。
壳的加载过程：
保存现场(pushad/popad,pushfd/popfd)
获取壳自己需要的 API 地址（LoadLibrary+GetProcAddress）
解密原程序各个区块
IAT 的初始化
重定位
Hook-API
跳到 OEP



壳的分类

压缩壳：
UPX

UPX 是一个以命令行方式操作的可执行文件经典免费压缩程序，压缩算法自己实现，速度极快。（开源）主页：http://upx.sourceforge.net/

ASPack。

ASPack 是 Win32 可执行文件压缩软件，可压缩 Windows 32 位可执行文件（.exe）以及库文件（.dll、.ocx），文件压缩比率 40%～70%。主页：http://www.aspack.com/

加密壳：

ASProtect

ASProtect 是一款非常强大的 Windows 32 位保护工具，开发者是俄国人 Alexey Solodovnikov，拥有压缩、加密、反跟踪代码、反-反汇编代码、CRC 校验和花指令等保护措施，使用 Blowfish、Twofish、TEA 等强劲的加密算法，还用 RSA1024 作为注册密钥生成器。它还通过 API 钩子（API hooks，包括 Import hooks（GPA hook）和 Export hooks）与加壳的程序进行通信。甚至用到了多态变形引擎（Polymorphic Engine）。反 Apihook 代码（Anti-Apihook Code）和 BPE32 的多态变形引擎（BPE32 的 Polymorphic Engine）。ASProtect 为软件开发人员提供 SDK，实现加密程序内外结合。主页：http://www.aspack.com/

Armadillo

Armadillo（穿山甲），是一款应用面较广的壳。可以运用各种手段来保护你的软件，同时也可以为软件加上种种限制，包括时间、次数，启动画面等等。很多商用软件采用其加壳。Armadillo 中比较强大的保护选项是 Nanomites 保护（即 CC 保护），用的好能提高强度，其他选项没什么强度。主页：http://www.siliconrealms.com

Themida

Themida 是 Oreans（西班牙著名的软件系统保护公司）的一款商业壳。Themida 1.1 以前版本带驱动，稳定性有些影响。Themida 最大特点就是虚拟机保护技术，在程序中用 SDK 将关键的代码让 Themida 用虚拟机保护起来。Themida 的缺点是生成的软件有些大。WinLicense 壳和 Themida 是同一公司的一个系列产品，主要多了一个协议，可以设定使用时间，运行次数等功能，两者核心保护是一样的。主页：www.oreans.com

VMProtect

VMProtect 是一款纯虚拟机保护软件，是当前最强的虚拟机保护软件，经 VMProtect 处理过的代码，还原难度极大。流行的做法，先用 VMProtect 将核心代码处理一下，再选用一款兼容性好的壳保护，比如继续用 Asprotect, Themida 等加壳软件进一步保护。主页：www.VMProtect.ru

VMP 与传统的壳（压缩，加密）相比，它会修改目标，让目标的部分指令（形成虚拟机下的字节码）在它创建的虚拟机环境下运行，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

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

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

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

Upx 加壳实战

Demo：

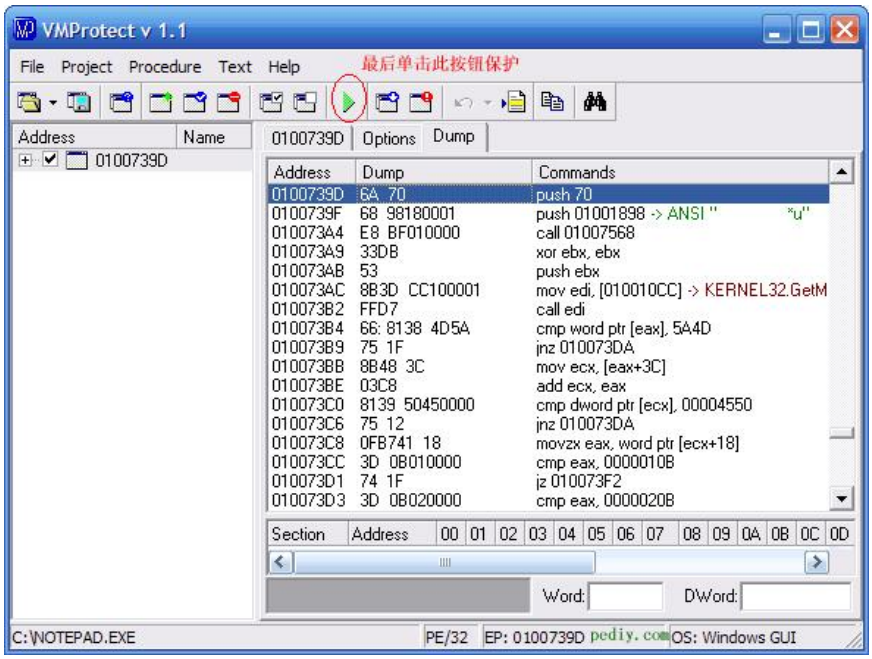
http://upx.sourceforge.net/

upx xxx.exe
导入表模糊化了
经过 UPX 压缩的 win32/pe 文件，包含区段：UPX0, UPX1。
UPX0：在文件中没有内容，它的"Virtual size"加上 UPX1 的构成了原文件全部区段需要的内存空间，相当于区段合并。
UPX1：起始位置为需解压缩的源数据，目标地址为 UPX0 基址。紧接着源数据块是“UPX stub”，即壳代码。
实战

VMP 加壳

加壳时，须告诉 VMProtect 要加密的代码地址（可以用调试器，如 OllyDbg 跟踪）然后将该地址添加到 VMProtect。

同时 VMP 支持 SDK，可以编程时插入一个标记，然后在加密时，VMProtect 会认出这些标记，并在有标记的地方进行保护。
VMProtect 的 SDK 标志
VMPBEGIN
//要加密的核心代码片断
VMPEND
观察 VMP 壳



VMP 加壳实战-1

开发一个简单的 MFC 程序
逆向分析

VMP 加壳实战-2

安装 VMProtect，安装目录获取库文件：
vmprotect_gr\Include\C
VMProtectSDK.h VMProtectDDK.h
vmprotect_gr\Lib
VMProtectSDK32.dll
VMProtectSDK32.lib
库：

```
#include "VMProtectSDK.h"
#pragma comment(lib, "VMProtectSDK32.lib")
```

代码保护：

```
VMProtectBegin("tagname");
VMProtectEnd();
```

字符串保护：

```
VMProtectDecryptStringW("mf123456"), //此时密文密钥 key 被解密
```

更多方法

```
//开始保护处标记
        VMProtectBegin(const char *);

//开始虚拟化代码处标记（包括保护设置）
VMProtectBeginVirtualization(const char *);
//开始变异代码处标记（包括保护设置）
VMProtectBeginMutation(const char *);
//开始虚拟+代码变异标记处
VMProtectBeginUltra(const char *);
VMProtectBeginVirtualizationLockByKey(const char *);
VMProtectBeginUltraLockByKey(const char *);

 //保护结束处标记
VMProtectEnd(void);

//检测调试
BOOL VMProtectIsDebuggerPresent(BOOL);

        //检测虚拟机
BOOL VMProtectIsVirtualMachinePresent(void);

//映像文件 CRC 校验
BOOL VMProtectIsValidImageCRC(void);
```

```
 //解密被保护的名为字符串 A
char * VMProtectDecryptStringA(const char *value);

 //解密被保护的名为字符串 W
wchar_t * VMProtectDecryptStringW(const wchar_t *value);
char* Decrypt( char* key, char* buff, long len ){}
Decrypt(
VMProtectDecryptStringA("mf123456"), //此时密文密钥 key 被解密
buff,
256
);
```

移动 APP 的加固（加壳）

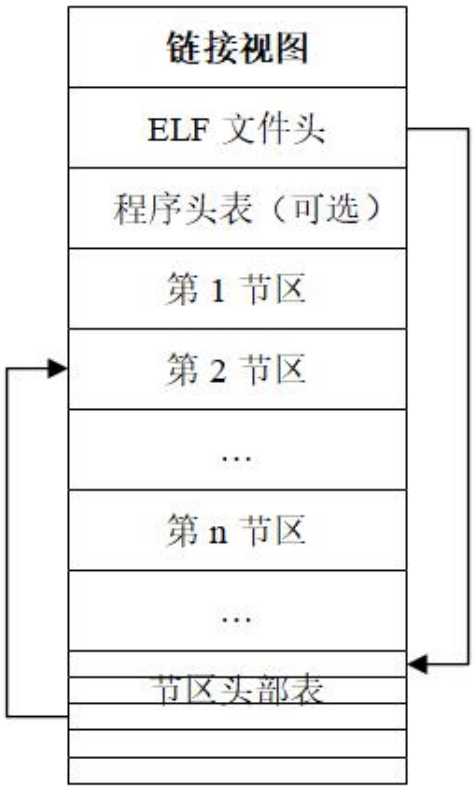
Android APK 加固思路简析

源 APK：准备一个源 APK（待加壳的 APK），普通 Android 工程开发，实现 Application 类
解壳 APK：将源 APK 加密后放在解壳 APK 的 DEX 文件中，由解壳 APK 运行时候，负责从解壳 APK 的 DEX 中解密和加载执行源 APK，普通 Android 工程开发
加壳程序：将源 APK 加密合并放入解壳 APK 的 DEX 文件中（修改 dex 文件头）生成一个新的 DEX，并替换解壳 APK 中的 DEX 文件，普通的 java/C 工程皆可


DEX Header
checksum
signature
file_size
解壳DEX Body
加密Source DEX数据
加密Source DEX数据长度

so 基于特定 section 的加解密原理

```
typedef struct {
    unsigned char    e_ident[EI_NIDENT];  /* File identification. */
    Elf32_Half    e_type;        /* File type. */
    Elf32_Half    e_machine;  /* Machine architecture. */
    Elf32_Word  e_version;    /* ELF format version. */
    Elf32_Addr   e_entry;/* Entry point. */
    Elf32_Off     e_phoff;/* Program header file offset. */
    Elf32_Off     e_shoff; /* Section header file offset. */
    Elf32_Word  e_flags;  /* Architecture-specific flags. */
    Elf32_Half    e_ehsize;      /* Size of ELF header in bytes. */
    Elf32_Half    e_phentsize;/* Size of program header entry. */
    Elf32_Half    e_phnum;    /* Number of program header entries. */
    Elf32_Half    e_shentsize; /* Size of section header entry. */
    Elf32_Half    e_shnum;    /* Number of section header entries. */
    Elf32_Half    e_shstrndx;  /* Section name strings section. */
} Elf32_Ehdr;
```

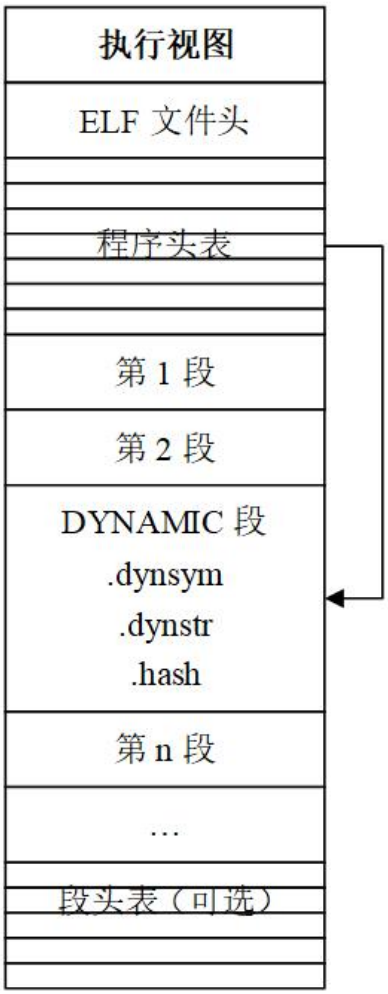


节区头部表里记录了节区的地址 shdr ->sh_offset ，大小 shdr ->sh_size 和名字在字符串节区的偏移 shdr ->sh_name

根据偏移 shdr ->sh_name 在字符串节区里找到该节区的名字，与要加密的节区比对

so 基于特定函数的加解密原理

```
typedef struct {
    unsigned char    e_ident[EI_NIDENT];  /* File identification. */
    Elf32_Half    e_type;        /* File type. */
    Elf32_Half    e_machine;  /* Machine architecture. */
    Elf32_Word  e_version;    /* ELF format version. */
    Elf32_Addr   e_entry;/* Entry point. */
    Elf32_Off     e_phoff;/* Program header file offset. */
    Elf32_Off     e_shoff; /* Section header file offset. */
    Elf32_Word  e_flags;  /* Architecture-specific flags. */
    Elf32_Half    e_ehsize;      /* Size of ELF header in bytes. */
    Elf32_Half    e_phentsize;/* Size of program header entry. */
    Elf32_Half    e_phnum;    /* Number of program header entries. */
    Elf32_Half    e_shentsize; /* Size of section header entry. */
    Elf32_Half    e_shnum;    /* Number of section header entries. */
    Elf32_Half    e_shstrndx;  /* Section name strings section. */
} Elf32_Ehdr;
```



Elf 头中的：ehdr.e_phoff，ehdr.e_phnum 指定了程序头表，程序头表遍历找到 DYNAMIC 段。
DYNAMIC 段里：
hash 节中存放在函数 hash 的 index
通过 index 可以在 dynsym 节中找到函数符号的名字在 dynstr 中的位置 funSym.st_name，起始地址 funSym.st_value 和长度 funSym.st_size
通过函数名字位置可以在 dynstr 节中找到函数具体名字和要加密的比对
简单粗暴的 so 加解密实现
https://bbs.pediy.com/thread-191649.htm
基于对 so 中的 section 加密技术实现 so 加固
http://blog.csdn.net/jiangwei0910410003/article/details/49962173
更多的加固方法
OLLVM：
混淆 native 代码：so 文件
Proguard
DashO
Dexguard
DexProtector
ApkProtect
Shield4j
Stringer
Allitori
VMP：
对二进制函数指令翻译为自定义指令集并加密存储，在运行过程中交由虚拟机解释执行。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
android 平台目前不存在纯粹的 android VMP，实现的原理为 dalvik 解释器的机制
常见的移动 APP 加固产品
1.360 加固保 链接：http://jiagu.360.cn/
2.梆梆安全 链接：http://www.bangcle.com/
3.爱加密 http://www.ijiami.cn/android
4.顶象 https://www.dingxiang-inc.com/business/stee
5.网易易盾：http://dun.163.com/product/android-reinforce

四，动态反调试

1-DebugPort 清零
2-KdDisableDebugger，禁用内核调试
KdEnableDebugger
3-IsDebuggerPresent 和 CheckRemoteDebuggerPresent
4-hook
DebugPort 是进程 EPROCESS 结构里的一个成员，指向了一个用于进程调试的对象，如果一

个进程不在被调试的时候那么就是 NULL，否则他是一个指针。该对象负责在调试器与被调进程之间进行调试事件传递，因此被称为调试端口。被调试程序的事件由这个端口发送到调试器进程的。

```
lkd> dt _EPROCESS
nt!_EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x06c ProcessLock      : _EX_PUSH_LOCK
   +0x070 CreateTime       : _LARGE_INTEGER
   +0x078 ExitTime         : _LARGE_INTEGER
   +0x080 RundownProtect   : _EX_RUNDOWN_REF
   +0x084 UniqueProcessId  : Ptr32 Void
   +0x088 ActiveProcessLinks : _LIST_ENTRY
   +0x090 QuotaUsage       : [3] Uint4B
   +0x09c QuotaPeak        : [3] Uint4B
   +0x0a8 CommitCharge     : Uint4B
   +0x0ac PeakVirtualSize  : Uint4B
   +0x0b0 VirtualSize      : Uint4B
   +0x0b4 SessionProcessLinks : _LIST_ENTRY
   +0x0bc DebugPort        : Ptr32 Void
```

Hook

HOOK 系统中一些与调试相关的函数，也可以防止被各种调试器调试。比如某款程序在内核中就 HOOK 了下面这些函数：
NtOpenThread（）：防止调试器在程序内部创建线程
NtOpenProcess（）：防止 OD（OllyDbg）等调试工具在进程列表中看到
KiAttachProcess（）：防止被附加上
NtReadVirtualMemory（）：防止被读内存
NtWriteVirtualMemory（）：防止内存被写
KdReceivePacket（）：KDCOM.dll 中 Com 串口接收数据函数
KdSendPacket（）：KDCOM.dll 中 Com 串口发送数据函数，可以 HOOK 这 2 个函数用来防止双机调试。

Android 反调试

反调试：
JNI_OnLoad 方法中，读取本进程的 status 文件，查看 TracerPid 字段是否为 0，如果不为 0，那么就表示自己的进程被别人跟踪了，也就是 attach 了，那么这时候立马退出程序
端口号
ida 常用远程调试 23946 端口号

获取调试器进程名
/proc/$pid/cmdline
反模拟器
常见检测:
特有设备：/dev/qemu_pipe
特有 socket：/dev/socket/qemud
/system/build.prop 特征信息
基于 arm 处理器的 cache 检测
…

反调试实战

```
UINT AntiDebug(PVOID param)
{
	while (g_bWillExit == FALSE)
	{
		HANDLE hProcess = GetCurrentProcess();
		BOOL bDebuggerPresent=FALSE;
		CheckRemoteDebuggerPresent(hProcess, &bDebuggerPresent);

		if (IsDebuggerPresent() || bDebuggerPresent)
		{
			::ExitProcess(0);
		}
		Sleep(5000);

	}
	return 0;

}

AfxBeginThread((AFX_THREADPROC)AntiDebug, NULL);
```