Reverse Engineering 1

By 2019

About Me

CTF逆向&pwn选手

英国留学生

盘古实验室安全研究

About Reverse Engineering

什么是逆向工程? e.g. 这是Linux下的正向编译过程:

C源码 -> 汇编 -> .o -> 可执行文件 -> stripped的可执行文件

compile assemble link strip

只通过可执行文件理解程序逻辑

(e.i. 没有源代码)

Different Kinds of Reverse Engineering

1. Windows平台逆向（WinAPI, PE, 病毒）
2. Linux平台逆向（CTF, LinuxAPI, 内核驱动）
3. MacOS平台逆向（跟Linux类似）
4. Android平台逆向（Dalvik, ARM, xposed, Linux）
5. IOT固件逆向（固件提取, qemu, ARM, MIPS）
6. web前端逆向（CompressJS, Obfuscation, wasm）
7. ……

Why

为什么需要逆向?

1. 挖掘漏洞
2. 破解软件 (违法!)
3. 制作游戏外挂 (仍然违法!)
4. 分析病毒
5. 研究内部算法设计

Tools for Static Analysis

1. IDA Pro
2. Ghidra
3. Binary Ninja
4. Radare2

Tools for Dynamic Analysis (Debugger)

1. OllyDbg/OllyICE/x64Dbg/WinDbg (Windows)
2. pwndbg/gdb peda/pwngdb/gef (Linux)
3. Android Studio/IDA Pro (Android)

Advanved Tools

1. Angr/Klee (符号执行Symbolic Execution)
2. Intel Pin/DynamicRIO (动态插桩Dynamic Instrumentation)
3. Keystone (assembler)/Capstone (disassembler)/ Unicorn (emulator)
4. Binary Analysis Platform (Static Program Analysis, 世界第一侧防中国Auxy的冠名工具)

PE File Format

Windows下的可执行文件格式

.exe .dll .sys

存放代码、数据、资源、导入导出信息...

Section

在PE文件中，是分为不同区段的

e.g. 数据段，代码段，只读数据段，资源段等等

这些段，在执行时会被映射到内存

不同的段，在不同的分页(page)上

相同的段，如果size大于页的size(0x1000)，也会占多个页

DOS Header

存放了一个完整的DOS程序，为保证程序仍然可以在DOS下运行。

其中E_lfanew字段指向PE头。

指针全部是文件偏移(File Offset)。

NT Header

块数目 (Number of Sections)

基址 (Base Address)

入口点RVA (Entry Point RVA)

块对齐值 (Section Alignment) = 0x1000

文件对齐值 (File Alignment) = 0x200

输出表与输出表的RVA

RVA, VA and FO

文件内容，在执行时会被映射(map)到内存

Virtual Address (程序执行时的虚拟地址)

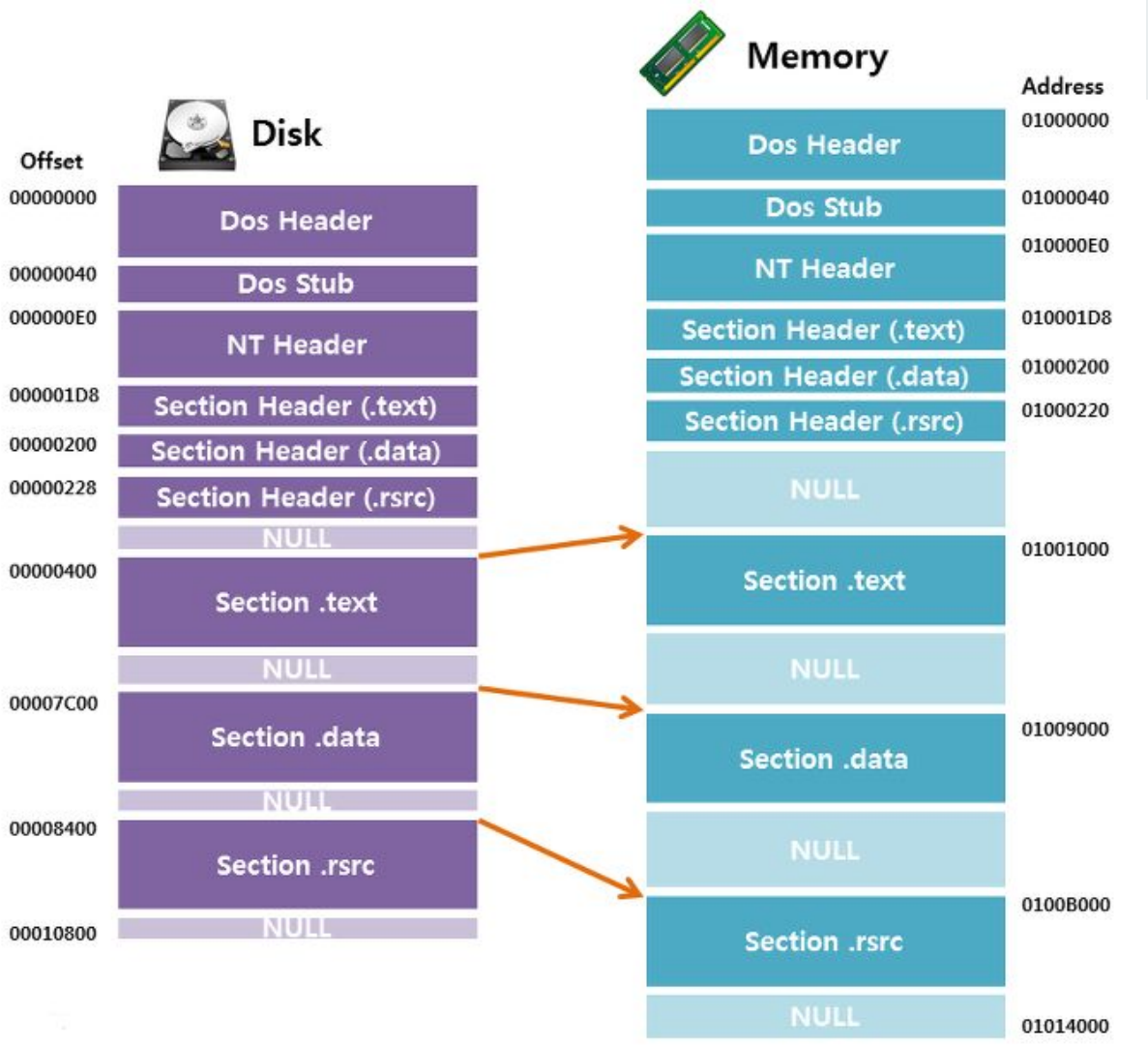
Relative Virtual Address = VA - Base Address

$$\text{File Offset} = \text{RVA} - \Delta k$$

$$\text{File Offset} = \text{VA} - \text{ImageBase} - \Delta k$$

△k 是一个区段开始RVA-FO的值

A picture is worth a thousand words



https://slimv.tistory.com/entry/PE-File-Format-0x01

Section Table

块名, e.g. “.data”, “.text”, “.rodata”, “.rsrc”

内存中的大小与RVA

文件中的大小与Offset

属性 (Characteristics), e.g. RWX属性, 是否初始化

动态链接库(Dynamic Link Library)

1. 共享只读页，节约物理内存空间
2. 封装：程序只需要知道API，不需要知道实现

    e.g. 同一个API，可能不同DLL版本的实现不同

输入表

有时候可执行文件需要从DLL调用外部的函数

比如说在User32.dll中的MessageBoxW

PE文件需要告诉OS，我需要某些DLL的某些函数

这个信息就由输入表来表示

_IMAGE_IMPORT_DESCRIPTOR，表示一个导入DLL

DLL文件名字符串的RVA指针

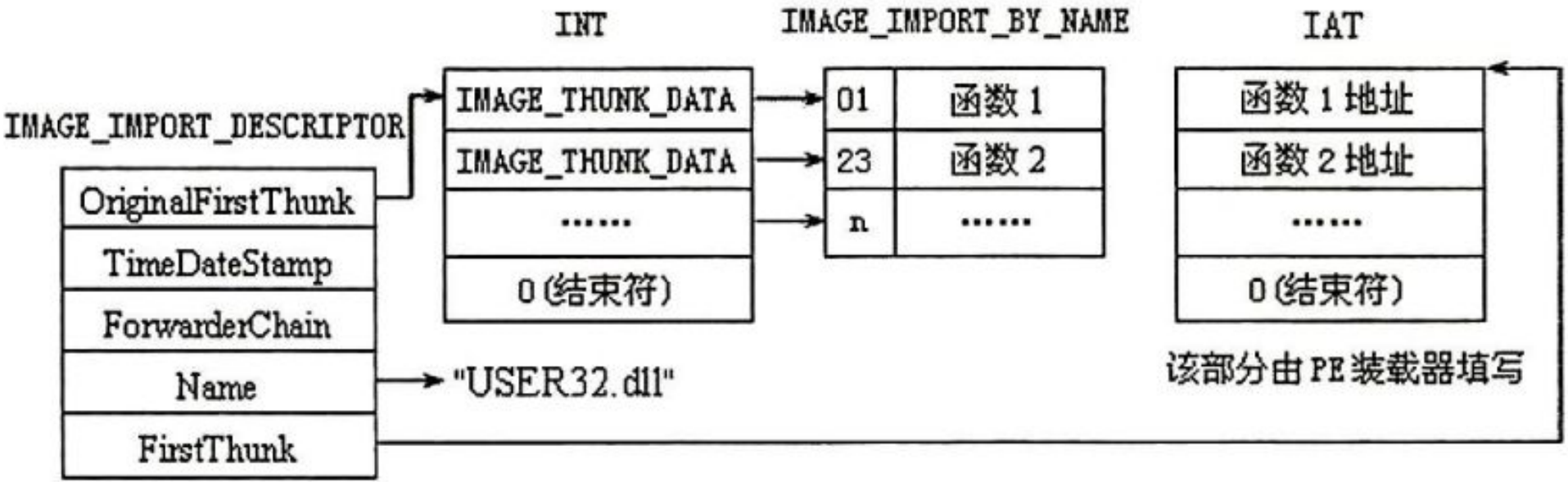
指向INT和IAT的RVA指针

这是两个RVA指针数组，

指向_IMAGE_IMPORT_BY_NAME结构，

这里通常存放函数名

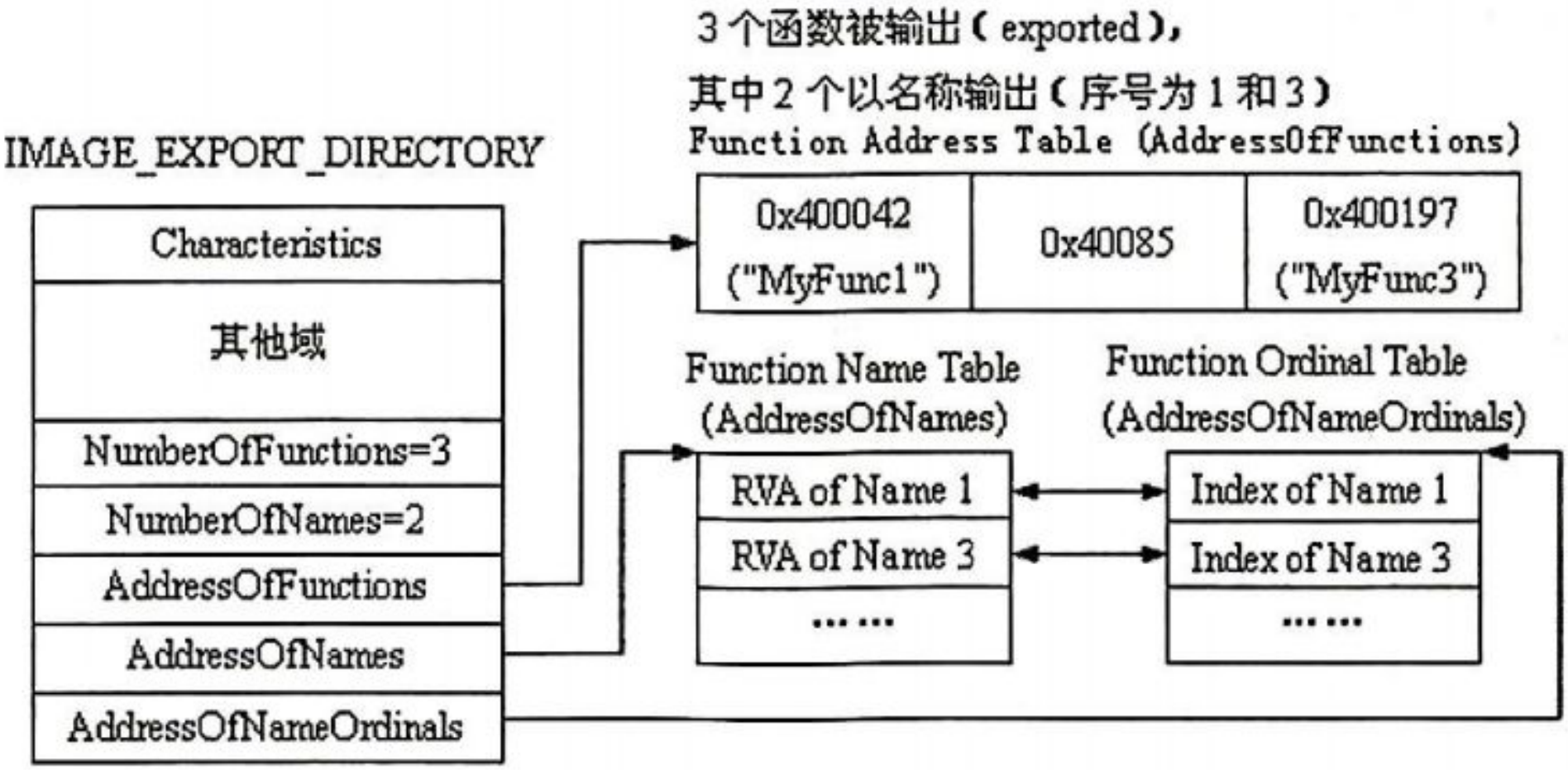
IAT在程序实际执行时，会被替换成实际DLL中的函数地址


INT
IMAGE_IMPORT_BY_NAME
IAT
IMAGE_IMPORT_DESCRIPTOR
OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name
FirstThunk
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
.....
0(结束符)
"USER32.dll"
01
函数 1
23
函数 2
n
.....
函数 1 地址
函数 2 地址
.....
0(结束符)
该部分由 PE 装载器填写

输出表

一个DLL需要告诉OS自己哪些函数需要导出

输出表就是用来存放这些信息的


3个函数被输出（exported），
其中2个以名称输出（序号为1和3）
IMAGE_EXPORT_DIRECTORY
Function Address Table (AddressOfFunctions)
Characteristics
其他域
NumberOfFunctions=3
NumberOfNames=2
AddressOfFunctions
AddressOfNames
AddressOfNameOrdinals
0x400042
("MyFunc1")
0x40085
0x400197
("MyFunc3")
Function Name Table
(AddressOfNames)
Function Ordinal Table
(AddressOfNameOrdinals)
RVA of Name 1
RVA of Name 3
……
Index of Name 1
Index of Name 3
……

重定位表

ASLR：地址随机化技术，pwn里应该讲过。。。

但是代码中是以PE头中的基址把相关指针hardcode

比如说 mov eax, [0x402010]

如果ASLR开启，实际地址应该是base+0x2010

重定位表就是记录所有这种需要修改的数据的位置的

PE32+

64位的可执行文件的格式

跟PE32基本没什么区别，道理都是一样的

个别字段的size不一样

https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

Practical

使用010Editor研究PE文件

找到各种结构的位置，结构中重要字段的值

Hint：使用010Editor的template，使世界更加美好

Windows Program

Windows是基于**消息机制**的

事件触发代码

点击一下窗口，会生成WM_LBUTTONDOWN消息

然后一些相应的代码会被执行，如果有的话

WindowProc (Callback Function)

窗口过程

每个窗口（包括子窗口），都有一个WindowProc

当产生消息时，WindowProc会被调用

这个callback有一些参数，包括消息种类，以及相关参数

How to find WindowProc?

1. OD  窗口可以直接看到
2. RegisterClassEx/RegisterClass的参数

```
typedef struct tagWNDCLASSEX {

  //...

  WNDPROC   lpfnWndProc;

  // ...

} WNDCLASSEX, *PWNDCLASSEX;
```

MFC Introduction

一个C++库，把Windows的API封装了一层

使得用C++开发Windows窗口程序更简单

其实在9102年，这玩意已经没什么人用了。。。

Reverse Engineering MFC

找到点击按钮所对应的事件

常见API：

GetWindowTextA/GetWindowTextW

MessageBoxA/MessageBoxW

断下之后找backtrace，或者在程序模块的代码页下断

Reverse Engineering MFC

找到点击按钮所对应的事件

寻找关键字符串 : View -> Open subviews -> Strings

消息断点 :  > 刷新 -> 右键按钮 -> 消息断点 -> WM_LBUTTONUP -> 在程序代码页F2 -> F9 -> 如果不是就回到DLL模块重复这个步骤

Example

看雪CTF 2019 Q1 第一题 流浪者

Reverse Engineering Data Structure

在IDA中创建struct

struct是如何被初始化的？(nullptr？值从哪来？malloc？)

struct是如何被使用和更新的？（loop bound, switch case）

Reverse Engineering Data Structure

常见数据结构：链表，AVL，红黑树，堆，字典树。。。

链表：CMU bomb lab phase_5

字典树：看雪CTF 2018 数据结构

Encrypt Algorithm

Xor　Base64　TEA　RC4

大整数运算　RSA

AES　DES

Complex Number　Fourier Transformation

目的:快速识别+快速找到相关参数(如key)

XOR

(a ^ b) ^ c = a ^ (b ^ c)　　（结合律）

a ^ 0 = 0 ^ a = 0　　（单位元）

a ^ a = 0　　（逆元）

a ^ b = b ^ a　　（交换律）

阿贝尔群

XOR

```
    a ^ b ^ a
=   b ^ a ^ a
=   b ^ (a ^ a)
=   b ^ 0
=   b
```

Example

picoCTF 2018 quackme

Base64

Source character	V								m								0							
ASCII number	86								109								48							
Bit pattern	0	1	0	1	0	1	1	0	0	1	1	0	1	1	0	1	0	0	1	1	0	0	0	0
Base64 number	21								38								52							
Base64 character	V								m								0							

https://www.imperva.com/blog/wp-content/uploads/sites/9/2018/04/B64-6.png

Base64

如何快速识别base64算法？

如何快速找到自定义base64表？

custom_b64encode(b64decode(std_tab)) == custom_tab

强网杯 2019 强网先锋AD

TEA

https://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm#Reference_code

*CTF 2018 wasm (modified)

RC4

https://github.com/freebsd/freebsd/blob/master/sys/crypto/rc4/rc4.c

GoogleCTF 2019 Qual MicroServiceDaemonOS

Big Integer & RSA

一般涉及到大整数的逆向，会有一个表示大整数的类

可能需要C++逆向的相关知识

HCTF2017 babyre

Complex Number & Fourier Transformation

一分看，六分猜，三分验证

GoogleCTF 2019 dialtone

Block Encryption and Hash

一些经验：

扒常数Google

猜+调试

搜特征字符串Google（识别开源库）

Example: Teaser Dragon 2018 Brutal Oldskull

1. CreateProcessA/GetExitCodeProcess
2. 哈希函数是什么？
3. 该如何去爆破？

Obfuscation

花指令：用于阻碍静态分析，使逆向更困难

阻碍反汇编器(disassembler)与反编译器(decompiler)

破坏指令对齐(instruction alignment)

破坏栈帧分析(stack frame analysis)

IDA脚本自动化去花

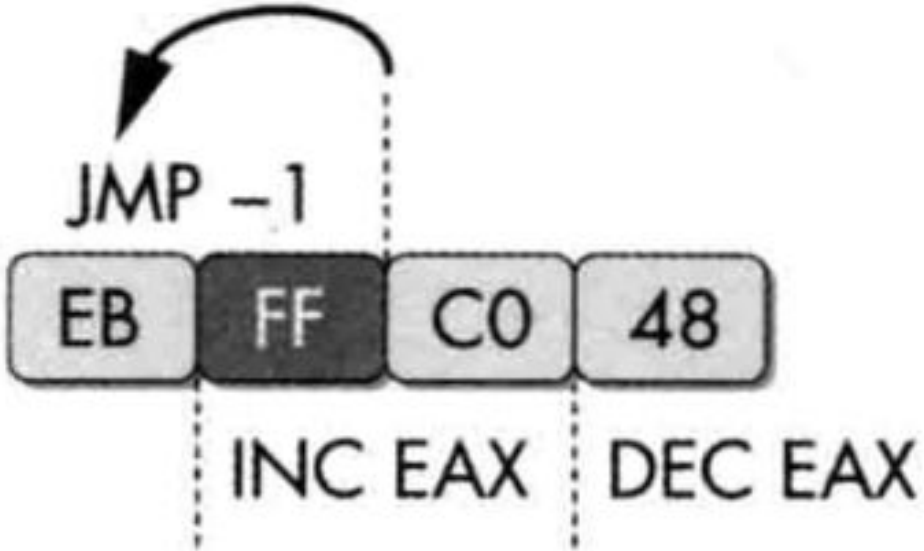
Break Instruction Alignment

```
    xor ecx,ecx
    jz label+1
label:
    mov al, ds:76E8616Ah ; wtf???
```

```
74 03                   jz      short near ptr loc_4011C4+1
75 01                   jnz     short near ptr loc_4011C4+1
                        loc_4011C4:                         ; CODE XREF: sub_4011C0
                                                            ; ❷sub_4011C0+2j
E8 58 C3 90 90          ❶call    near ptr 90D0D521h
```

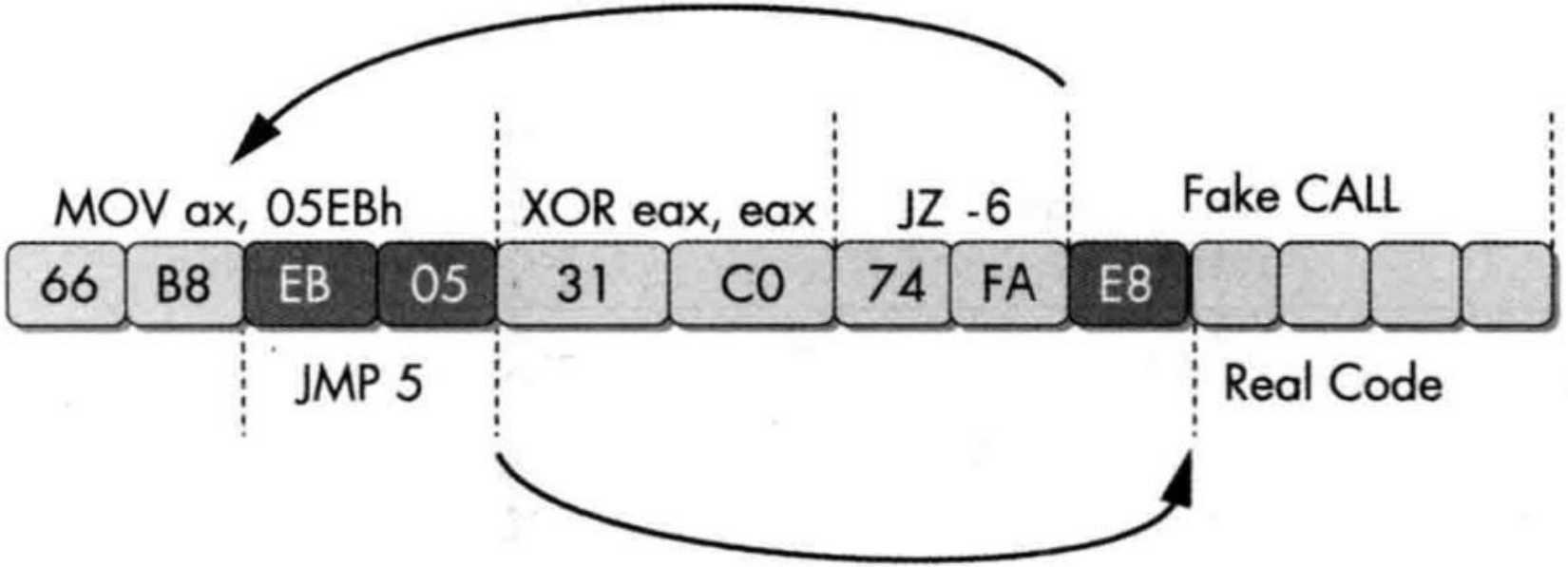
```
74 03                   jz      short near ptr loc_4011C5
75 01                   jnz     short near ptr loc_4011C5
        ; ---------------------------------------------------------------------
E8                      db 0E8h
         ; ---------------------------------------------------------------------
                        loc_4011C5:                         ; CODE XREF: sub_4011C0
                                                            ; sub_4011C0+2j
  58                    pop     eax
  C3                    retn
```

Break Instruction Alignment



```
>>> print disasm("\xff\xc0\x48")
   0:   ff c0                inc eax
   2:   48                   dec     eax
>>> print disasm("\xeb\xff\xc0\x48")
   0:   eb ff                jmp     0x1
   2:   c0                   .byte 0xc0
   3:   48                   dec     eax
```

Break Instruction Alignment



```
66 B8 EB 05             mov     ax, 5EBh
31 C0                   xor     eax, eax
74 FA                   jz      short near ptr sub_4011C0+2
                loc_4011C8:
E8 58 C3 90 90          call    near ptr 98A8D525h
```

```
66                      byte_4011C0     db 66h
B8                                      db 0B8h
EB                                      db 0EBh
05                                      db    5
                        ; ---------------------------------------
31 C0                                   xor     eax, eax
                        ; ---------------------------------------
74                                      db 74h
FA                                      db 0FAh
E8                                      db 0E8h
                        ; ---------------------------------------
58                                      pop     eax
C3                                      retn
```

Break Stack Frame Analysis

破坏IDA栈帧分析

让F5和CFG失效

```
    call next

next:

    pop eax

    add eax, xxx

    push eax

    ret
```

Break Stack Frame Analysis

本质上就是一个jmp

因为jmp也是一个相对跳转

（E9 + DWORD OFFSET）

但是IDA栈帧分析无法识别出来

```
00401543 000                    sub     esp, 8
00401546 008                    sub     esp, 4
00401549 00C                    cmp     esp, 1000h
0040154F 00C                    jl      short loc_401556
00401551 00C                    add     esp, 4
00401554 008                    jmp     short loc_40155C
00401556        ; ---------------------------------------------------------
00401556
00401556        loc_401556:                             ; CODE XREF: sub_401543+Cj
00401556 00C                    add     esp, 104h
0040155C
0040155C        loc_40155C:                             ; CODE XREF: sub_401543+11j
0040155C -F8❶                   mov     [esp-0F8h+arg_F8], 1E61h
00401564 -F8                    lea     eax, [esp-0F8h+arg_F8]
```

使用ALT+K调整栈帧

Example

Enigma 2017 syzygy

Patching

通过改变可执行文件来改变程序逻辑

常用于软件破解 jnz->nop

还能用来去混淆 花指令->nop

工具：IDA Pro, OD, 010Editor

Example

picoctf 2018 be-quick-or-be-dead-1

reversing.kr replace

IDA Python

Byte/Word/Dword/Qword

PatchByte/PatchWord/PatchDword/PatchQword

GetOpnd/GetOperandValue/GetMnem/GetManyBytes

NextHead/PrevHead

Other Obfuscation

OLLVM

movfuscator

...

The End

祝大家七夕快乐，有情人终成兄妹

晚上注意身体，不要玩的太嗨，明天还要继续

Reference

《加密与解密》

《恶意代码分析实战》