Reverse

by zfn 欢迎交流 二进制交流群：OTc4OTY2NDI2 (使用了常用算法进行编码) (听说有人猜不出这是 base64?)

这篇是把某次的 slides 直接拿过来了，修复了一下格式

还欠缺很多 IDA 等工具的实际演示，因为当时讲的时候是现场演示的 x

xwh 也写了一篇 Re 从零开始的逆向生活，为了不让他白写，放个链接在这。当然肯定没我写得好喵

附件下载

题目 1-3（第 x 波实战）、壳篇、z3 篇、符号执行篇(angr 笔记)在单独的文件里，点击下载

如果某一节内容很少，可能相关内容在附件里

Table of Contents

什么是逆向工程

你说的对，但《逆向工程》是由网络空间安全自主研发的一款全新解谜类竞技游戏。游戏发生在一个被称作「操作系统」的虚拟空间，在这里，被二进制代码选中的人将被授予「反编译器」，引导逆向之力。你将扮演一位名为「逆向工程师」的神秘角色，在绕过保护措施的旅途中邂逅各种加密与混淆，和他们一起找回失散的控制流——同时，逐步发掘「flag」的真相。

什么是逆向工程

软件代码逆向主要指对软件的结构，流程，算法，代码等进行逆向拆解和分析。
一般，CTF 中的逆向工程题目形式为：程序接收用户的一个输入，并在程序中进行一系列校验算法，如通过校验则提示成功，此时的输入即 flag。这些校验算法可以是已经成熟的加解密方案，也可以是作者自创的某种算法。比如，一个小游戏将用户的输入作为游戏的操作步骤进行判断等。这类题目要求参赛者具备一定的算法能力、思维能力，甚至联想能力。

程序/可执行文件

- 一个操作系统中的对象 (文件)
- 一个字节序列
- 一个描述了状态机初始状态的数据结构
  - 状态机初始状态的描述
    - 内存中的各段的位置和权限
    - 入口点
    - 寄存器和栈由操作系统决定
  - 状态迁移的描述
    - 代码

可执行文件中的基本结构

不同操作系统对应的可执行文件的结构通常不同（如 Windows 的 PE 文件和 Linux 的 ELF 文件），但却有很多部分在本质上是相似的。这里的例子适用于常见的原生二进制程序（反例：Java, Android）。

section

section 是编译器生成的，用于组织代码和数据的逻辑部分。每个 section 具有特定的属性和用途，比如代码段、数据段、符号表等。常见的 section 包括 .text（包含可执行代码）、.data（初始化的数据）、.bss（未初始化的数据）、.rodata（只读数据）等。

segment

segment 是链接器和操作系统关注的，是程序加载时的内存映射单元。segment 是将多个 section 合并到一起，一般连续的、权限相同的节会被合并。

导入表、导出表

需使用的来自其他动态链接库中的项目（例如函数）
以及
作为动态链接库提供给其他程序的项目

相关工具

- PE 文件：studype++
- ELF 文件：readelf
- 通用：Detect it easy

特殊的可执行文件/题目类型

特殊系统

- Android，iOS，OS X
- riscv，龙芯

源代码

- 如经过混淆的 php 文件，powershell 文件，常常作为木马上传。
- web 网站中打包后的 js 文件。
- 利用了特殊机制（操作系统、并发…）JavaCPScript - deadsec ctf

字节码（VM)

- 常见的：Java, Python, lua…
- 其他：智能合约，yara 规则…

从 x86(_64)架构开始

反汇编/反编译工具

首先准备一套覆盖逆向工程全流程的工具，包括反汇编、反编译（生成伪代码）、控制流图分析、自动化处理脚本、插件支持。

- IDA（建议先从这个开始）
- Ghidra
- Binary Ninja
- radare2/rizin

调试工具

实际上上面的工具是内置调试功能的，但功能不太丰富，容易崩溃，插件支持少，卡

- Linux：gdb
  - 使用此脚本一键安装 gef 与 pwndbg：https://github.com/apogiatzis/gdb-peda-pwndbg-gef
- Windows: x64dbg，windbg
  - 注：ollydbg 停更多年，不支持 64 位，基本上已被取代

x86_64 汇编速成

寄存器

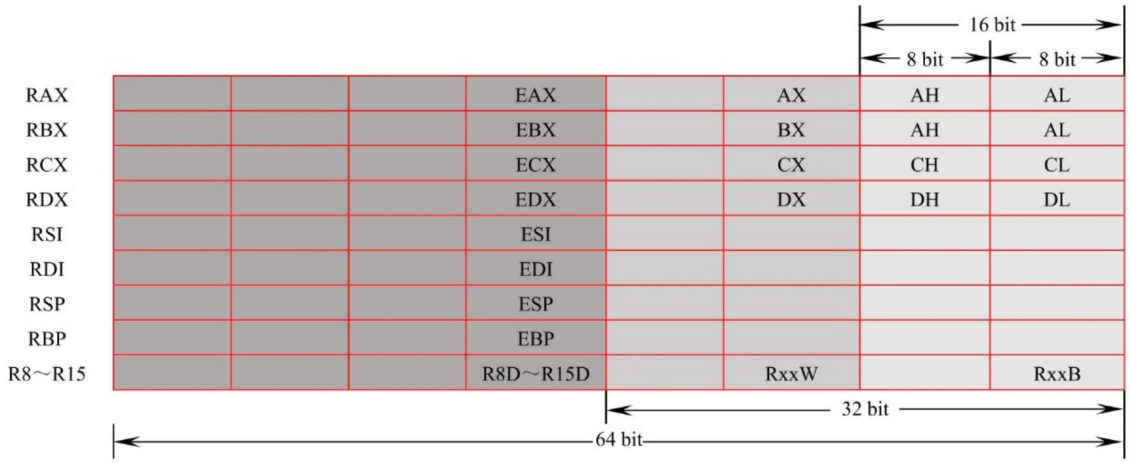
寄存器（Register）是 CPU 的组成部分，是有限存储容量的高速存储部件，用来暂存指令、数据和地址。一般的 IA-32（Intel Architecture，32-bit）即 x86 架构的处理器中包含以下在指令中显式可见的寄存器：

- 通用寄存器 EAX、EBX、ECX、EDX、ESI、EDI。
- 栈顶指针寄存器 ESP、栈底指针寄存器 EBP。
- 指令计数器 EIP（保存下一条即将执行的指令的地址）。
- 段寄存器 CS、DS、SS、ES、FS、GS。（可以忽略他们）

对于 x86-64 架构，在以上这些寄存器的基础上，将前缀的 E 改成 R，以标记 64 位，同时增加了 R8～R15 这 8 个通用寄存器。另外，对于 16 位的情况，则将前缀 E 全部去掉。

对于通用寄存器，程序可以全部使用，也可以只使用一部分。使用寄存器不同部分时对应的助记符见图 5-1-1。其中，R8～R15 进行拆分时的命名规则为 R8d（低 32 位）、R8w（低 16 位）和 R8b（低 8 位）。

可以将寄存器理解为预先定义好的变量，这变量没有类型，执行操作时只有长度的区别。



内存和寻址

寻址=解引用一个指针

寻址方式	示例	C 语言示例
直接寻址	[1000h]	int* ptr = (int*)0x1000; int val = *ptr;
寄存器间接寻址	[RAX]	int* ptr = (int*)rax; int val = *ptr;
基址寻址	[RBP+10h]	int* ptr = (int*)(rbp + 0x10); int val = *ptr;
变址寻址	[RDI+10h]	int* ptr = (int*)(rdi + 0x10); int val = *ptr;
基址加变址寻址	[RBX+RSI+10h]	int* ptr = (int*)(rbx + rsi + 0x10); int val = *ptr;

基本运算

数据传送指令	mov	mov rax, rbx	rax = rbx
		mov qword ptr [rdi], rax	*(rdi) = rax
取地址指令	lea	lea rax, [rsi]	rax = & *(rsi)
算术运算指令	add	add rax, rbx	rax += rbx
		add qword ptr [rdi], rax	*(rdi) += rax
	sub	sub rax, rbx	rax -= rbx
逻辑运算指令	and	and rax, rbx	rax &= rbx
	xor	xor rax, rbx	rax ^= rbx

asm_base

条件跳转

每次进行运算时，不仅会改变目标中的值，还会影响标志位

- AF：辅助进位标志（Auxiliary Carry Flag），当运算结果在第 3 位进位的时候置 1。
- PF：奇偶校验标志（Parity Flag），当运算结果的最低有效字节有偶数个 1 时置 1。
- SF：符号标志（Sign Flag），有符号整形的符号位为 1 时置 1，代表这是一个负数。
- ZF：零标志（Zero Flag），当运算结果为全零时置 1。
- OF：溢出标志（Overflow Flag），运算结果在被操作数是有符号数且溢出时置 1。
- CF：进位标志（Carry Flag），运算结果向最高位以上进位时置 1，用来判断无符号数的溢出。

但我们不用记这些

指　令	全　称	cmp a, b 条件	flag 条件
jz/je	jump if zero/equal	a = b	ZF = 1
jnz/jne	jump if not zero/equal	a != b	ZF = 0
jb/jnae/jc	jump if below/not above or equal/carry	a > b，有符号数	CF = 1
ja/jnbe	jump if above/not below or equal	a < b，有符号数	
jna/jbe	jump if not above/below or equal	a <= b，有符号数	
jnc/jnb/jae	jump if not carry/not below/above or equal	a >= b，有符号数	CF = 0
jg/jnle	jump if greater/not less or equal	a > b，无符号数	
jge/jnl	jump if greater or equal/not less	a >= b，无符号数	
jl/jnge	jump if less/not greater or equal	a < b，无符号数	
jle/jng	jump if less or equal/not greater	a <= b，无符号数	
jo	jump if overflow		OF = 1
js	jump if signed		SF = 1

asm_jmp

---

jump 里

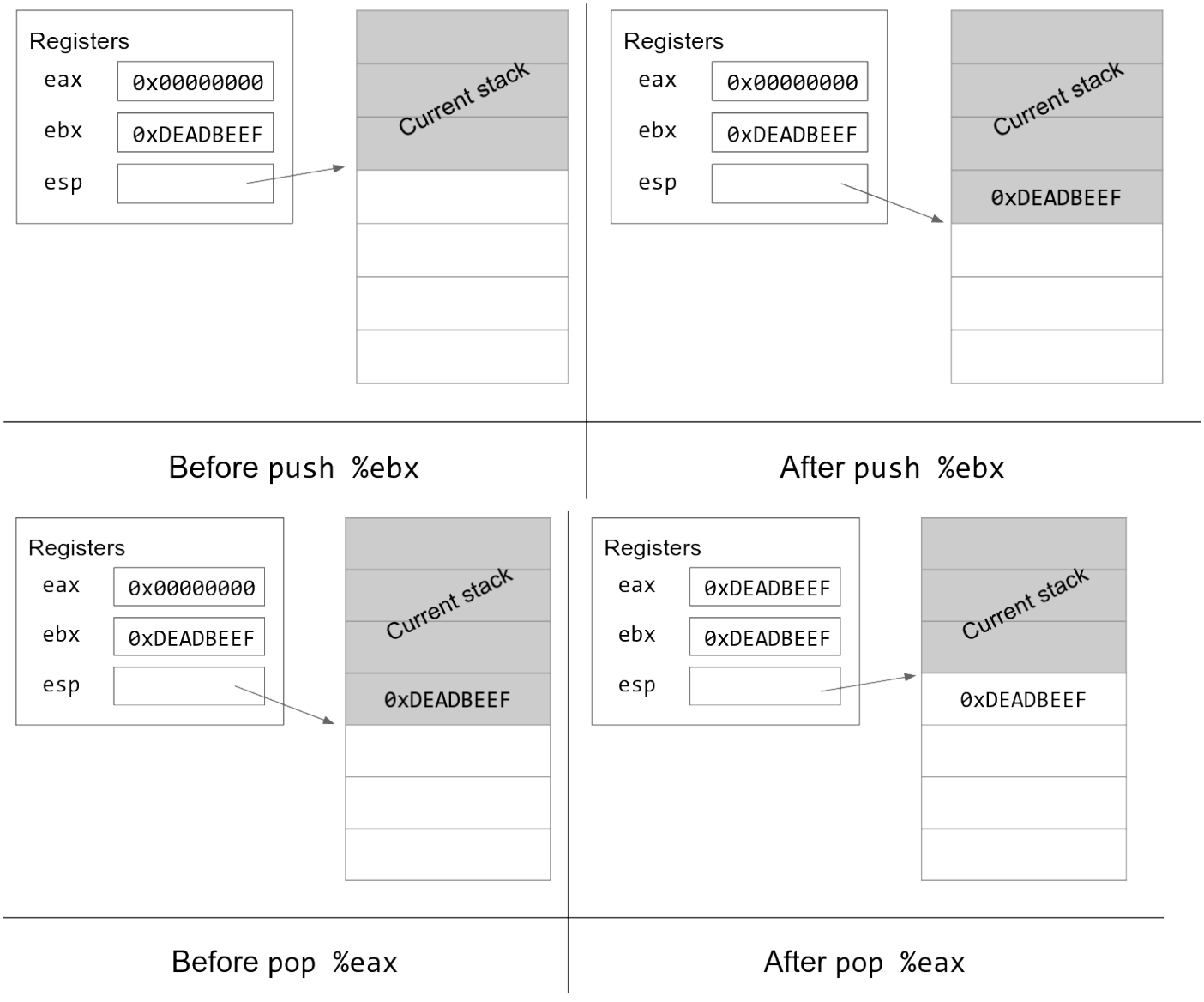
- n: not
- e: equal
- z: zero
- g: greater
- l: less
- b: blow
- a: above

a,b 有符号
g,l 无符号

函数调用

栈

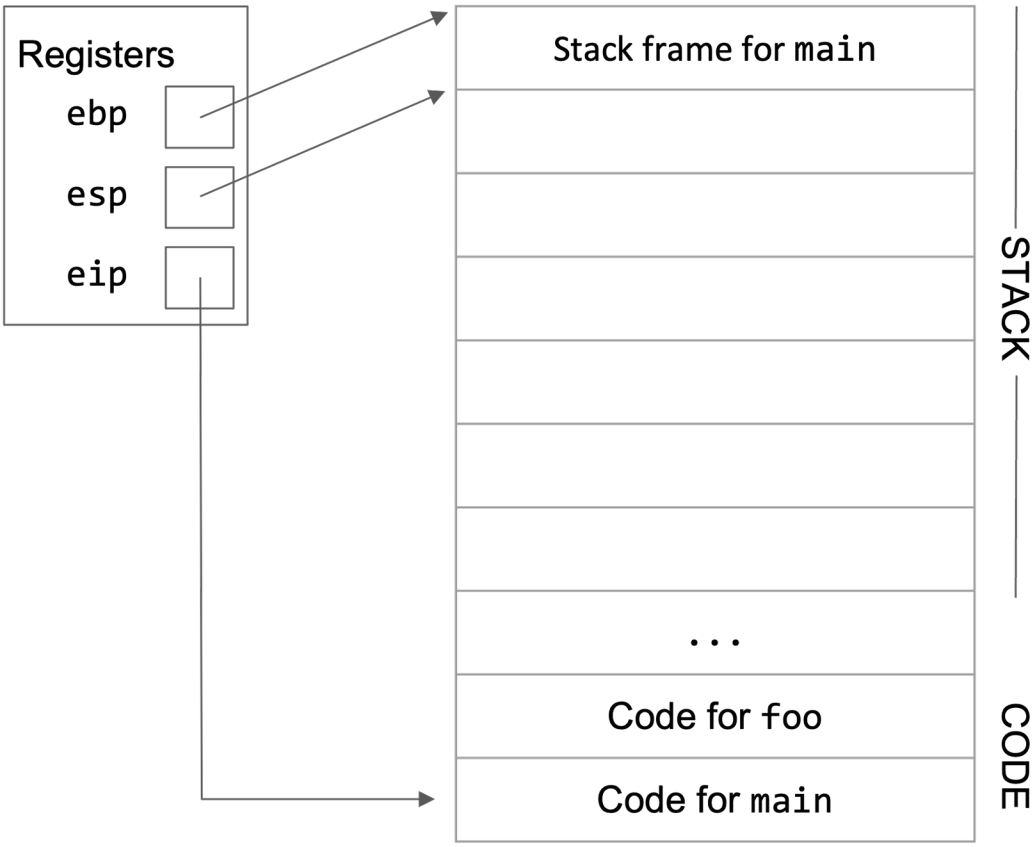
x86 的栈是从顶往下塞东西的。
ebp 是基指针，它存储当前栈顶地址。（什么叫当前栈？之后解释）
esp 是栈指针，它存储当前堆底地址。

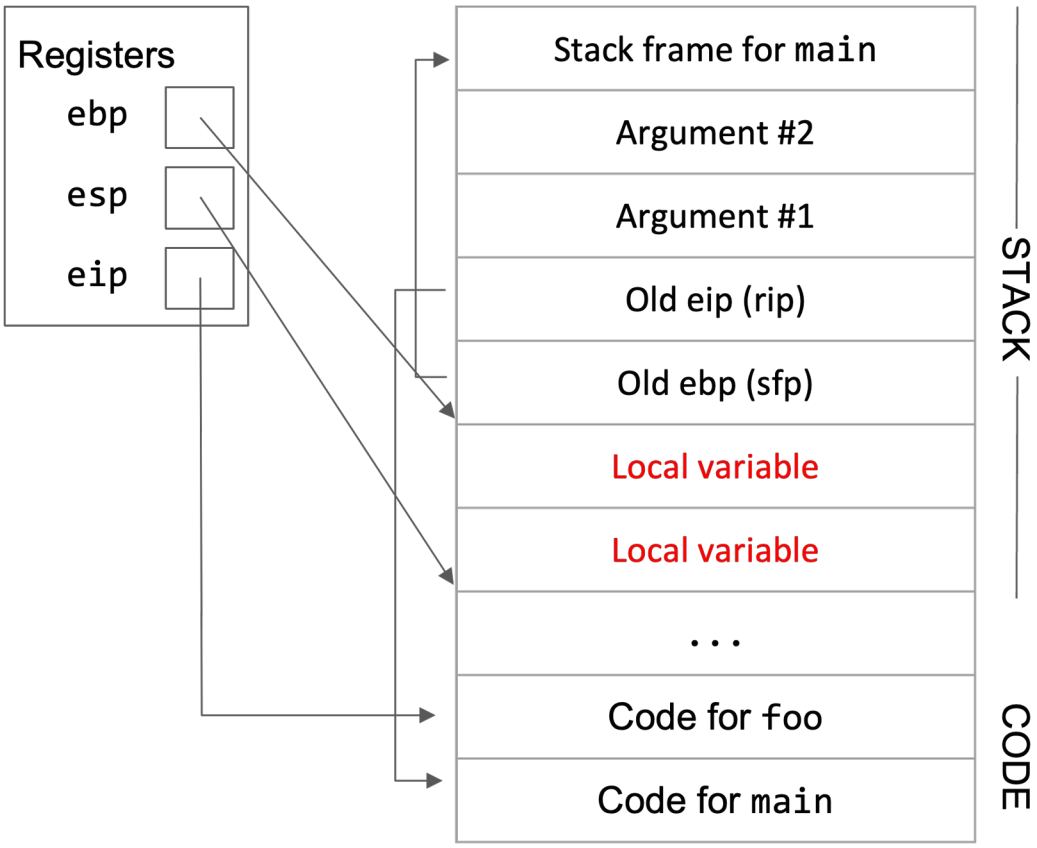


调用过程

在 x86 架构中，ESP（中保存的地址）和 EBP（中保存的地址）之间的区域通常称为栈帧。栈帧是每个函数调用时在栈上分配的一块内存，用于保存函数的局部变量、返回地址、传递的参数以及保存调用者的寄存器状态。

更详细的：https://textbook.cs161.org/memory-safety/x86.html#27-x86-calling-convention





参数

- x86 32 位架构的调用约定
  - __cdecl：参数从右向左依次压入栈中，调用完毕，由调用者负责将这些压入的参数清理掉，返回值置于 EAX 中。绝大多数 x86 平台的 C 语言程序都在使用这种约定。
  - __stdcall：参数同样从右向左依次压入栈中，调用完毕，由被调用者负责清理压入的参数，返回值同样置于 EAX 中。Windows 的很多 API 都是用这种方式提供的。
  - __thiscall：为类方法专门优化的调用约定，将类方法的 this 指针放在 ECX 寄存器中，然后将其余参数压入栈中。
  - __fastcall：为加速调用而生的调用约定，将第 1 个参数放在 ECX 中，将第 2 个参数放在 EDX 中，然后将后续的参数从右至左压入栈中。
- x86 64 位架构的调用约定
  - Microsoft x64 位（x86-64）调用约定：在 Windows 上使用，依次将前 4 个参数

放入 RDI、RSI、RDX、RCX 这 4 个寄存器，然后将剩下的参数从右至左压入栈中。

- SystemV x64 调用约定：在 Linux、MacOS 上使用，比 Microsoft 的版本多了两个寄存器，使用 RDI、RSI、RDX、RCX、R8、R9 这 6 个寄存器传递前 6 个参数，剩下的从右至左压栈

第一批实战

- Reversing-x64Elf-100: 简单

- re5-packed-movement：了解 ida 脚本
  IDA python: https://www.yunyawu.com/2020/06/28/ida-python%E5%AD%A6%E4%B9%A0/
- encrypt：搜索加密算法

算　法	特征值（如无特殊说明为十六进制）	备　注
TEA 系列	9e3779b9	Delta 值
AES	63 7c 77 7b f2 6b 6f c5 …	S 盒
	52 09 6a d5 30 36 a5 38 …	逆 S 盒
DES	3a 32 2a 22 1a 12 0a 02 …	置换表
	39 31 29 21 19 11 09 01 …	密钥变换数组 PC-1
	0e 11 0b 18 01 05 03 1c …	密钥变换数组 PC-2
	0e 04 0d 01 02 0f 0b 08 …	S 函数表格 1
BlowFish	243f6a88 85a308d3 13198a2e 03707344	P 数组
MD5	67452301 efcdab89 98badcfe 10325476	寄存器初始值
	d76aa478 e8c7b756 242070db c1bdceee …	Ti 数组常量
SHA1	67452301 efcdab89 98badcfe 10325476 c3d2e1f0	寄存器初始值
CRC32	00000000 77073096 ee0e612c 990951ba	CRC 表
Base64	字符串"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"	字符集

代码保护

自修改

在运行时修改自身代码，从而使得程序实际行为与反汇编结果不符，同时修改前的代码段数据也可能非合法指令，从而无法被反汇编器识别。

花指令

花指令（junk code）是一种专门用来迷惑反编译器的指令片段，这些指令片段不会影响程序的原有功能，但会使得反汇编器的结果出现偏差，从而使破解者分析失败。比较经典的花指令技巧有利用 jmp 、call、ret 指令改变执行流，从而使得反汇编器解析出与运行时不相符的错误代码。

壳

附件中有手动脱壳教程 [附件]](#附件下载)

把代码藏在数据里，运行时再解密/解压到代码段（这个 segment 也是现场创建的）

做题时建议用工具，不要手动脱，这边演示一下 win 和 linux 的 upx。

但研究原理的时候建议手动试一下

windows: scylla
linux: coredump(`echo 0xff > /proc/self/coredump_filter` 才能 dump 完整内存)

工具脱：

```
upx -d [filename]
```

虚拟机

虚拟机就是要去模仿一个机器，让机器去执行一个程序
一般包括指令序列、存储（堆栈、寄存器）

https://bbs.kanxue.com/thread-267670.htm

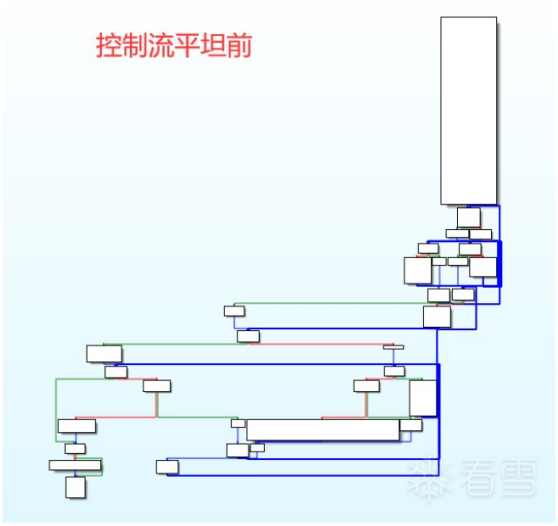
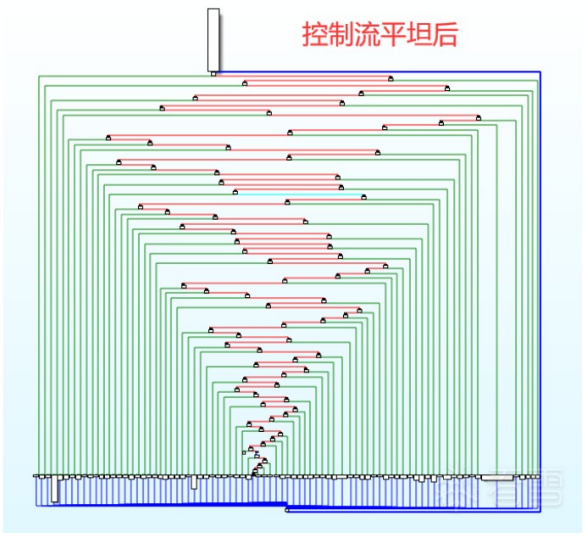
混淆

把代码变成狗看了都摇头的样子。

---

ollvm

最直观的变化是控制流平坦化，但其实还有别的功能（虚假控制流，指令替换）。
https://github.com/cq674350529/deflat
https://oacia.dev/ollvm-study/

控制流平坦前	控制流平坦后
	

---

movfuscator

工具：https://github.com/leetonidas/demovfuscator

```
<is_prime>:
  push   ebp
  mov    ebp,esp
  sub    esp,0x10
  cmp    DWORD PTR [ebp+0x8],0x1
  jne    8048490 <is_prime+0x13>
  mov    eax,0x0
  jmp    80484cf <is_prime+0x52>
  cmp    DWORD PTR [ebp+0x8],0x2
  jne    804849d <is_prime+0x20>
  mov    eax,0x1
  jmp    80484cf <is_prime+0x52>
  mov    DWORD PTR [ebp-0x4],0x2
  jmp    80484be <is_prime+0x41>
  mov    eax,DWORD PTR [ebp+0x8]
  cdq
  idiv   DWORD PTR [ebp-0x4]
  mov    eax,edx
  test   eax,eax
  jne    80484ba <is_prime+0x3d>
  mov    eax,0x0
  jmp    80484cf <is_prime+0x52>
  add    DWORD PTR [ebp-0x4],0x1
  mov    eax,DWORD PTR [ebp-0x4]
  imul   eax,DWORD PTR [ebp-0x4]
  cmp    eax,DWORD PTR [ebp+0x8]
  jle    80484a6 <is_prime+0x29>
  mov    eax,0x1
  leave
  ret
```

反调试

- Windows: 直接在 x64dbg 上装 scylla hide
- Linux: 请看文章：https://xz.aliyun.com/t/6882

第二波实战

- EASYHOOK 自修改
- oflo 花指令+自修改 https://ctf-wiki.org/reverse/obfuscate/junk-code/#n1ctf2020-oflo
- pack.exe：windows upx 壳
- a.out：linux upx 壳
- EzMachine: vm

高级技巧

Hook

Hook 是在指令的关键位置插入特定代码，以干预程序原有的执行流程，实现拦截目标进程运行过程的关键信息改变目标进程原本执行流程等目的。

题目 3 中的 TraceMe 和 hooking.js 是使用 frida 框架的例子

约束求解

Z3 是一个微软出品的开源约束求解器，能够解决很多种情况下的给定部分约束条件寻求一组满足条件的解的问题。

```
from z3 import *

x = Int('x')
y = Int('y')
solve(x > 2, y < 10, x + 2*y == 7)
```

z3.ipynb 和题目 3：useZ3

符号执行

符号执行就是在运行程序时，用符号来替代真实值，目的（可以）是探究通过各种分支最终抵达某个程序状态的条件。符号的概念更接近于（对某个寄存器/某段内存的）一组约束，而不是具体的值。

笔记(angr)和题目 3：angr_ctf