## Wp--控制流平坦化+虚假控制流idapy脚本处理

## 1.处理文件

## 1.1 控制流平坦化

## 将文件直接拖入ida64中，查看main函数的控制流程图(CFG)

## IMG_256

## 得到的代码是一个多重循环,我们需要对代码进行控制流平坦化。

## 可以参考：<https://security.tencent.com/index.php/blog/msg/112>

## deflat.py脚本

## <https://pan.baidu.com/s/1Tuw2ITns1bI05nWfTpwNa>g

## 提取码：u2g0

## 或<https://github.com/cq674350529/deflat>

## 

## 使用时需要注意哦！！

## 使用deflat.py执行控制流平坦化脚本命令：

## python3 deflat.py attachment 0x400620

## #python版本+脚本名+文件名+起始地址

## 

## （ps：使用脚本需要安装angr库。不会安装的小伙伴可以看看我的这篇文章：Kali angr环境配置）

## <https://blog.csdn.net/liuxiaohuai_/article/details/114291565>

## 需要等待几分钟！！！。

## 1.2 处理虚假控制流程

## 得到的文件使用拖入ida64中，查看main函数，得到反汇编代码：

## （注意：我自己写的wp里ida数据库文件是经过虚假控制流idapy脚本处理过的，故看不到下列控制流

## IMG_256观察代码，代码使用了虚假控制流程，代码中的条件都是永真条件，因此可以在观察时忽略or删除while语句和do…while语句。

## 也可以使用脚本。提供一个官方WP中给出的脚本，在IDA中的script中执行即可。

## st = 0x0000000000401117

## end = 0x0000000000402144

## def patch\_nop(start,end):

## for i in range(start,end):

## PatchByte(i, 0x90) #修改指定地址处的指令 0x90是最简单的1字节nop

## 

## def next\_instr(addr):

## return addr+ItemSize(addr) #ItemSize获取指令或数据长度，这个函数的作用就是去往下一条指令

## 

## 

## addr = st

## while(addr<end):

## next = next\_instr(addr)

## if "ds:dword\_603054" in GetDisasm(addr): #GetDisasm(addr)得到addr的反汇编语句

## while(True):

## addr = next

## next = next\_instr(addr)

## if "jnz" in GetDisasm(addr):

## dest = GetOperandValue(addr, 0) #得到操作数，就是指令后的数

## PatchByte(addr, 0xe9)

## PatchByte(addr+5, 0x90)

## offset = dest - (addr + 5)

## PatchDword(addr + 1, offset)

## print("patch bcf: 0x%x"%addr)

## addr = next

## break

## else:

## addr = next

## 最终我们的到了控制流平坦化后的代码

## 

## 2 分析代码

## 通过观察可以，实际上这部分代码使用的是CRC32的查表法，对数据进行加密。

## 2.1 加密原理

## 加密原理实际上就是CRC32算法—输入一组长度48的字符串，每8个字节分为1组，共6组。对每一组取首位，判断正负。正值，左移一位；负值，左移一位，再异或0xB0004B7679FA26B3。重复判断操作64次，得到查表法所用的表。

## 因此我们只需要将整个加密过程逆向操作得到查表法的表，再进行CRC64计算，就能得到flag

## 3 通过脚本得到flag

## secret = [0xBC8FF26D43536296, 0x520100780530EE16, 0x4DC0B5EA935F08EC,

## 0x342B90AFD853F450, 0x8B250EBCAA2C3681, 0x55759F81A2C68AE4]

## key = 0xB0004B7679FA26B3

## flag = ""

## # 产生CRC32查表法所用的表

## for s in secret:

## for i in range(64):

## sign = s & 1

## # 判断是否为负

## if sign == 1:

## s ^= key

## s //= 2

## # 防止负值除2，溢出为正值

## if sign == 1:

## s |= 0x8000000000000000

## # 输出表

## print(hex(s))

## # 计算CRC64

## j = 0

## while j < 8:

## flag += chr(s&0xFF)

## s >>= 8

## j += 1

## print(flag)

## 查表法的表：

## 0x6666367b67616c66

## 0x63362d3039333932

## 0x2d363563342d3032

## 0x3539612d30376162

## 0x6631643365383537

## 0x7d38

## flag:

## flag{6ff29390-6c20-4c56-ba70-a95758e3d1f8}

## 2.JMP 目标地址计算方式

目标地址 = 当前指令地址 + 指令长度 + RVA

### JMP 有 3 种类型：

1. 短跳转（Short Jmp，只能跳转到256字节的范围内），对应机器码：EB
2. 近跳转（Near Jmp，可跳至同一段范围内的地址），对应机器码：E9
3. 远跳转（Far Jmp，可跳至任意地址），对应机器码： EA

短跳转和近跳转指令中包含的操作数都是相对于(E)IP的偏移。  
远跳转指令中包含的是目标的绝对地址。  
所以短/近跳转会出现跳至同一目标的指令机器码不同，不仅会不同，而且应该不同。  
而远跳转中包含的是绝对地址，因此转移到同一地址的指令机器码相同 。  
至此，我们知道了跳转指令所对应的机器码根据E(IP)计算出来的，那到底是怎么计算的呢？

如图： 想要从 001392DB 处跳转到 001392E1 这个地址，使用近跳转来计算则是如下公式：

RVA = 001392E1 - 001392DB - 5

在计算机中使用上面的公式计算结果就是 1 所以后面应该是跟 E9 01 00 00 00  
其中 5 是指令长度 E9 后面需要跟 4 字节的地址，所以指令(1)+地址(4) = 5