# 基于跳转指令目标的粗粒度CFI规则

## 一、实验目的

在目前的很多内存攻击中，控制流攻击占其中的很大一部分，比如ROP，JOP等，因此有人提出了根据程序控制流图来进行严格检查控制流的想法。

CFI实现检查待运行程序的控制流图CFG或者在程序运行过程中进行动态插桩，然后在运行过程中检查控制流。具体的策略包括，当检查jmp指令时，检查跳转地址是否是当前函数内或者函数起始地址。当检查到Call指令时，检查跳转地址是否是函数起始地址。当检查到return指令时，检查返回地址是否是对应call指令的下一条指令地址。

## 实验环境

QEMU 2.7.0

Ubuntu 16.04

## 实验步骤

### 1. 标记JMP，CALL，RETURN指令

C语言代码经过编译后成为二进制代码，然后QEMU执行指令时，将其一条一条地翻译成中间代码，因此，我们首先应该在QEMU中的translate.c文件里找到的jmp，call，return指令对应的操作码。然后添加上这样的标记，（以call为例）

/\* call Ev \*/

if(coarsecfi\_enabled){

s->have\_call = 1;

}

**call**

0xfe: /\* GRP4 \*/

0xff: /\* GRP5 \*/

0xe8: /\* call im \*/

0x9a: /\* lcall im \*/

0x105: /\* syscall \*/

0xd9: /\* VMMCALL \*/

**JMP**

0xfe: /\* GRP4 \*/

0xff: /\* GRP5 \*/

0xfe: /\* GRP4 \*/

0xff: /\* GRP5 \*/

0xe9: /\* jmp im \*/

0xea: /\* ljmp im \*/

0xeb: /\* jmp Jb \*/

**ret**

0xc2: /\* ret im \*/

0xc3: /\* ret \*/

0xca: /\* lret im \*/

0xcb: /\* lret \*/

0xcf: /\* iret \*/

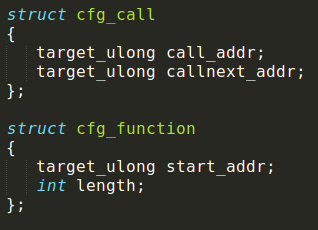
0x107: /\* sysret \*/

我们在相应的Case操作码 处插入标记代码，这样当QEMU翻译到此处时，将给相应的TB添加上指令标记，表示此处有JMP指令，CALL指令或者RETURN指令。

### 读入CFG

这里给出了两个CFG文件，vul.call记录了call指令的地址，以及它的下一条指令地址，而vul.func记录了函数头以及函数长度。 这些信息我们应该在QEMU初始化的时候就将其读入。

我们在translate.c文件中找到了void tcg\_x86\_init(void)函数，在此处读入我们的两个cfg文件。我们首先定义了全局两个结构体和两个全局变量。

cfg\_call记录了call指令的地址以及它下一个地址

cfg\_call\_cnt记录了cfg文件里call指令地址的条数

同理，cfg\_function记录了vul.func文件中函数的地址以及函数长度

cfg\_function\_cnt 记录了cfg文件中记录的函数的个数

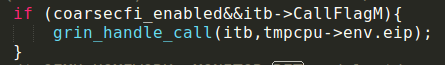
我们通过C语言的文件操作读取文件，得到类似于4008a9 4008ae这样格式的字符串。然后通过strtok函数对得到的字符串中的制表符“\t”进行分割，得到我们所需要的值4008a9和4008ae。

但是这两个值都是字符串的，我们通过strtol函数将其转换成16进制的数字。存入结构体，以供后期的检测时使用。

### 获取操作数

由于QEMU的TCG机制直接将二进制指令转换为中间语言，中间并没有计算出操作数（直接转换了），而在执行阶段，才会将操作数以及下一条pc出来计算出来。 我们的思路是JMP，CALL，以及RETURN指令必定会改变PC值，因此将它们执行完毕后，pc的值将会是它们调用，跳转或者返回的目标地址，作业给出Hint中给出了tmpcpu->env.eip的值，这正是我们所需要的操作数。

我们把当前TB以及eip作为参数传入到我们的检测函数中。

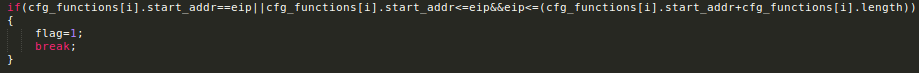


### 检测-JMP

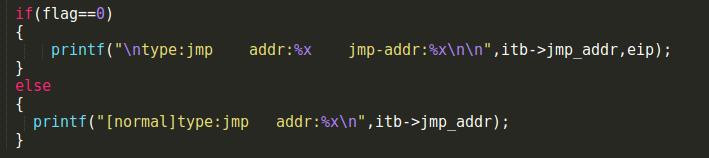
因为我们不需要检查库函数中的函数，因此我们将不是0x400开头的地址都屏蔽掉，如图所示



我们遍历了cfg函数数组，如果函数的起始地址与跳转的目标地址相同，或者跳转的目标地址在当前函数范围内，则我们认为它是正常的跳转指令。



如果跳转指令为正常，则我们将其输出，格式为 类型+指令的地址，如果是异常，我们则将其输出为 类型 + 指令地址 + 跳转地址。



### 检测-CALL

同理我们将库函数中的call地址排除，不进行检测。

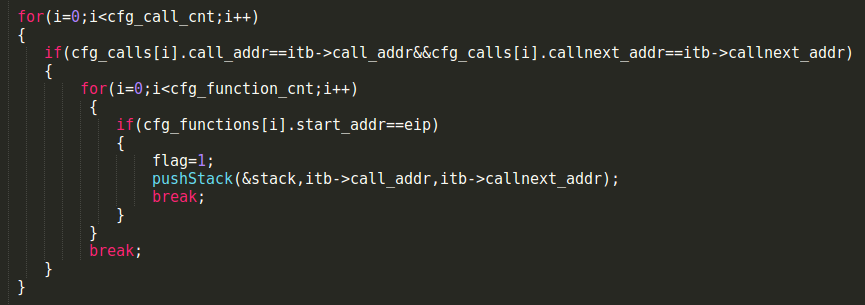
我们的算法是

先检测我们的调用函数的地址是否在vul.call中出现

如果出现了，则遍历 vul.func中的函数地址

如果跳转的地址是函数的首地址，则符合我们的要求，并且将其push到

我们设计的一个栈中，为了匹配后面的return指令做准备。

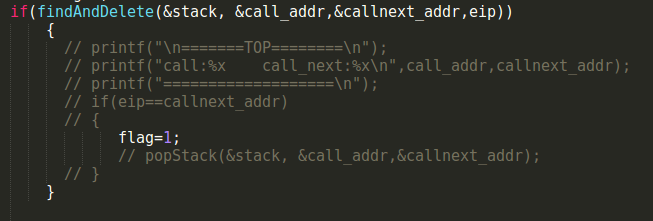


输出的格式与JMP类似，不再赘述！

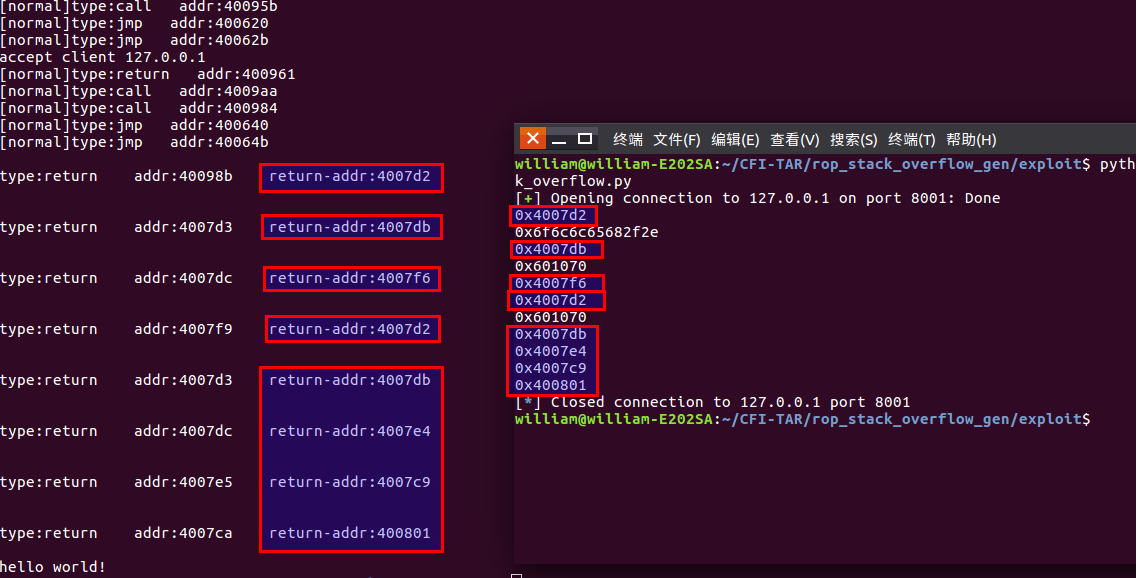
1. 检测-RETURN

RETURN的返回地址比较复杂，因为它要匹配到我们对应的call指令上，我们在此处认为，如果我们栈中保存的未结束的call指令的下一条地址与我们当前的return指令的返回地址相同，则它们是对应的call-return对。

我们反向遍历（从TOP处开始遍历）我们的栈，如果遇到与当前RETURN地址的返回地址对应的call指令，则我们将call从栈中移除。并认为我们的return是符合要求的。



### 演示



ROP 攻击中构造了很多ret配件，这些ret没有对应的call调用，因此会被判为异常。

### 存在的问题

一般在进行左右括号匹配“（）”的时候，会使用栈这一数据结构，当遇到左括号时，进栈，遇到右括号则匹配出栈。但是在进行匹配call和return的时候，这一算法将失效。因为所有的call和return不一样（每个左括号和右括号都是一样的），因此我们最后把它改成了遍历寻找对应的call，这在递归时可能找到的call并不是自己对应的那个call。