《漏洞利用及渗透测试基础》实验报告

姓名： 艾明旭 学号：2111033 班级：信息安全一班

**实验名称：**

程序插桩及Hook实验

**实验要求：**

复现实验一，基于Windows MyPinTool或在Kali复现malloctrace这个PinTool,理解Pin插桩工具的核心步骤和相关API，关注malloc和free函数的输入输出信息

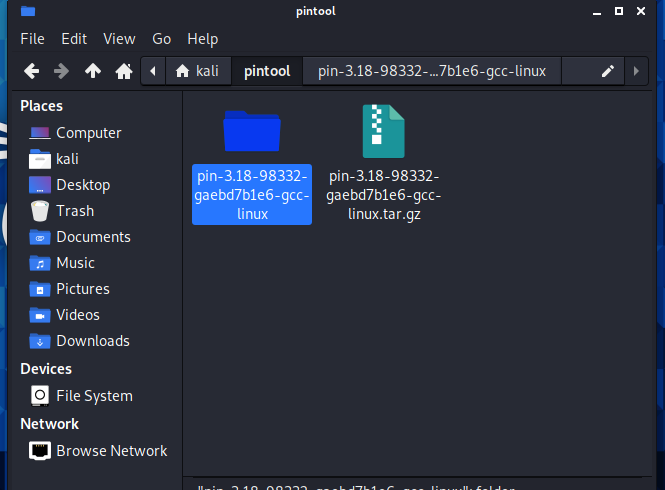
**实验过程：**

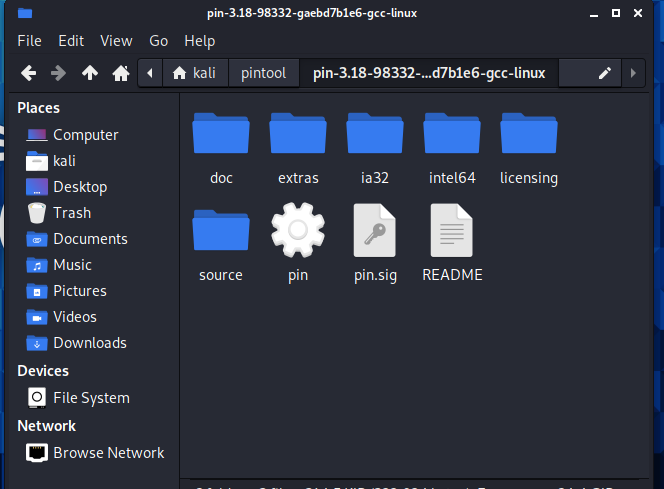
Pin 是 Intel 公司开发的动态二进制插桩框架，可以用于创建基于动态程序分析工具，

支持 IA-32 和 x86-64 指令集架构，支持 windows 和 linux。 Pin 可以监控程序的每一步执行，提供了丰富的 API，可以在二进制程序程 序运行过程中插入各种函数，比如说我们要统计一个程序执行了多少条指令，每条指令的地 址等信息。显然，这样我们对程序完全掌握了以后是可以做很多事的。

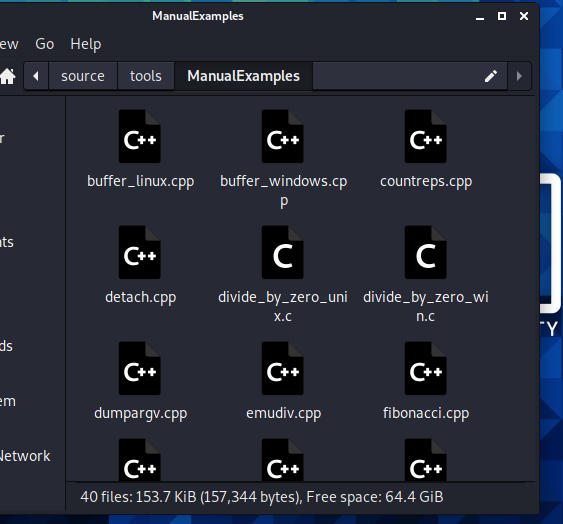
首先，安装Pin及以inscount0为例使用 PinTool

此处安装的是pin-3.18

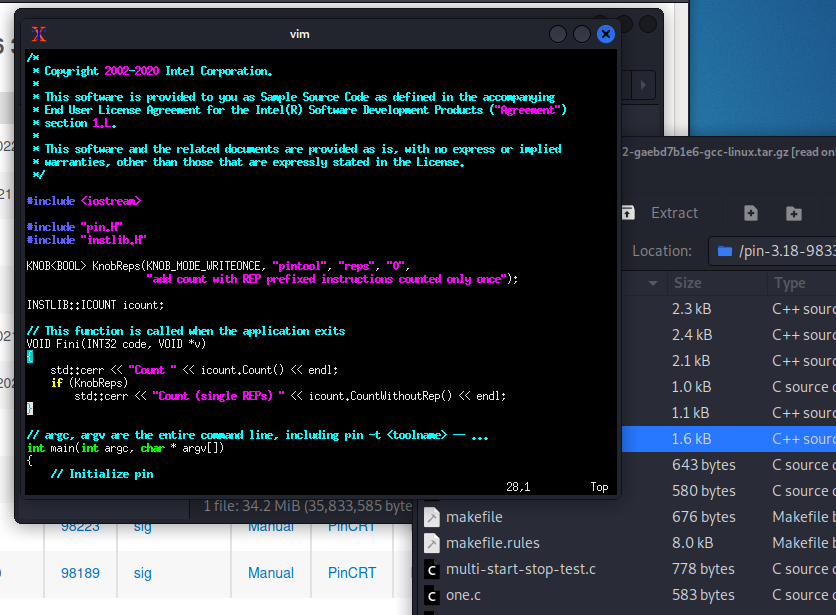
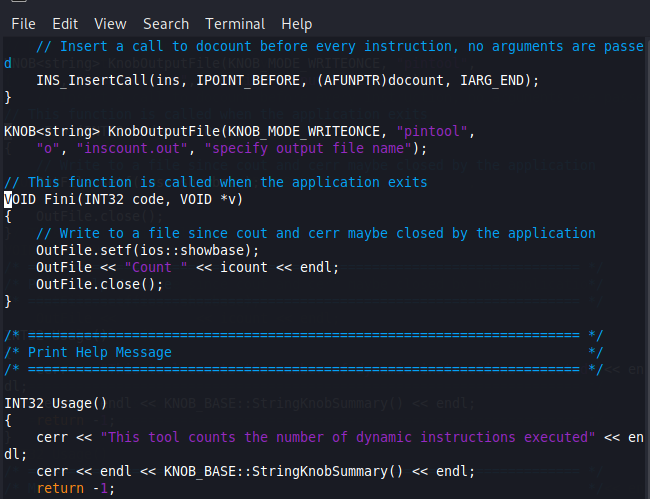
pin主要利用PinTool来完成插桩功能的包装和实现



pin中提供了一些简答的插桩函数示例

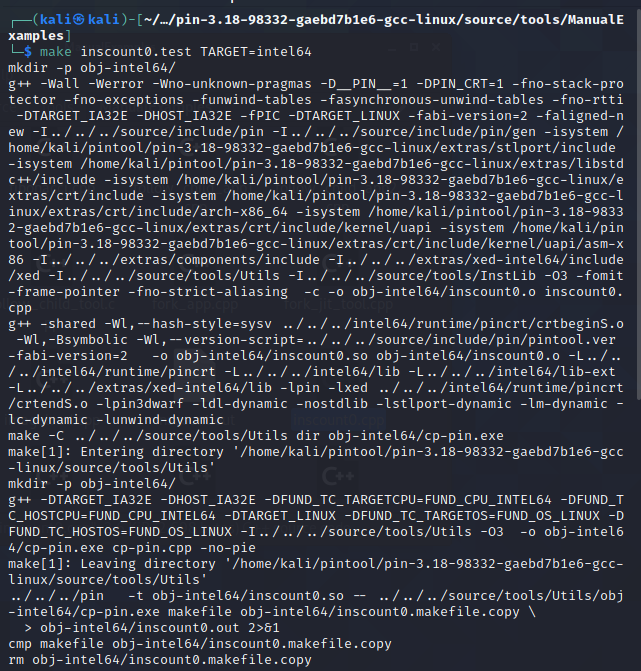


比如inscount0.cpp可以进行指令计数

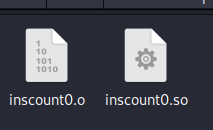
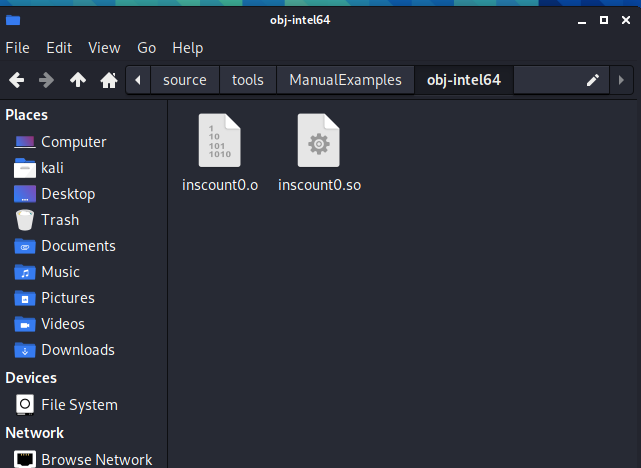


接下来编译运行，产生动态链接库，以使用PinTool

使用命令：make inscount0.test TARGET=intel64 指定编译inscount0



可以看到产生了PinTool的动态链接库



接下来进行测试

编写一个简单的控制台命令程序 FirstC.c，并进行测试，代码如下。

#include <stdio.h>

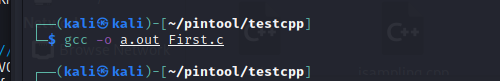
void main()

{

printf("hello world!");

}

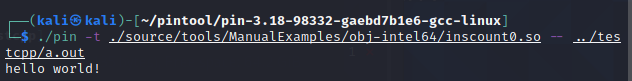
在 Linux 下编译 c 文件，命令为: gcc –o First First.c



用Pin进行插桩

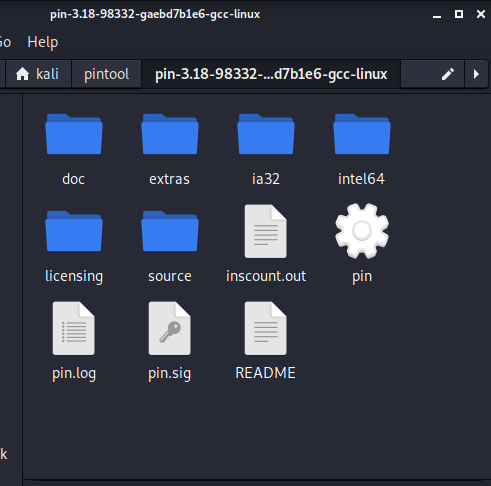
对 First 可执行程序进行程序插桩的 Pin 命令为：

./pin -t ./source/tools/ManualExamples/obj-intel64/inscount0.so -- ../testcpp/a.out

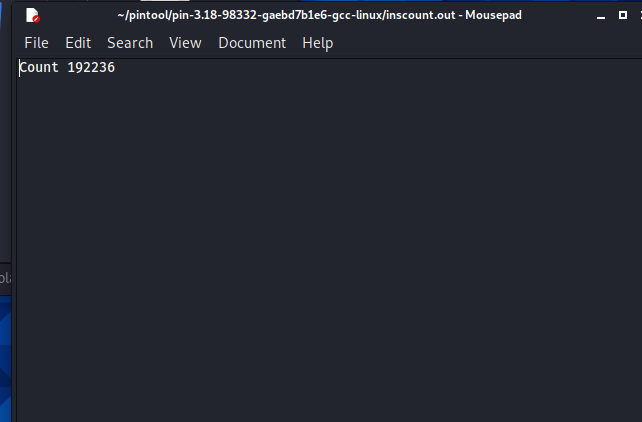


显示hello world 表明成功运行

同时产生了一个输出文件，记录输出的状态和信息



文件内容如下：“Count 192236”，即对指令数进行了插桩



接下来尝试对PinTool进行更改

原PinTool是指令级插桩，通 过调用 INS\_AddInstrumentFunction 注册了一个回调函数 Instruction。

VOID Instruction(INS ins, VOID \*v)

{

//在每个指令之前插入一个函数 docount 的调用，没有任何参数

INS\_InsertCall(ins, IPOINT\_BEFORE, (AFUNPTR)docount, IARG\_END);

}

在 Instruction 函数中，使用函数 INS\_InsertCall 注册了一个回调函数 docount，意为

在指令执行之前插入一个对 docount 函数的调用。而 docount 的作用即是将一个全局变量加 1，以达到统计执行指令条数的目的。故此处插桩的分析代码即是将指令数加 1

现在在inscount0的基础上，扩展出更加复杂的插桩分析程序

**指定插桩的位置**。最简单的情况是直接针对所有指令插桩，INS 模块中提供了很多 API

来判断当前指令的类型：

INS\_IsMemoryRead (INS ins) INS\_IsDirectCall (INS ins)

INS\_IsDirectBranchOrCall (INS ins) INS\_IsMemoryWrite (INS ins)

INS\_IsBranchOrCall (INS ins) INS\_IsLea (INS ins)

INS\_IsCall (INS ins) INS\_IsNop (INS ins)

INS\_IsRet (INS ins) INS\_IsBranch (INS ins)

INS\_IsDirectBranch (INS ins)

回调函数 Instruction 修改如下

VOID Instruction(INS ins, VOID \*v)

{

if (INS\_Opcode(ins) == XED\_ICLASS\_MOV &&

INS\_IsMemoryRead(ins) &&

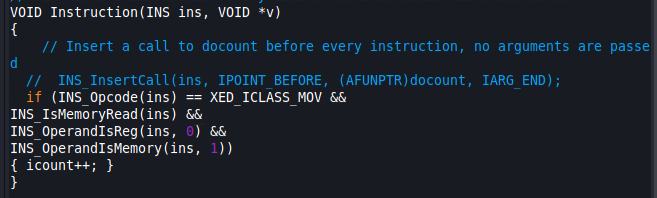
INS\_OperandIsReg(ins, 0) &&

INS\_OperandIsMemory(ins, 1))

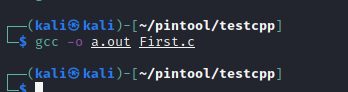
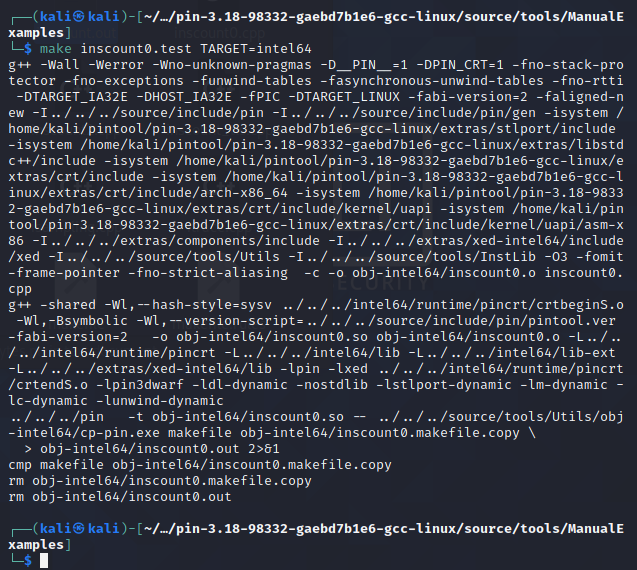
{ icount++; } }

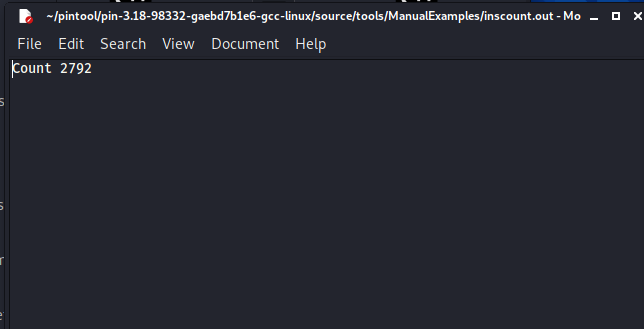
在现在的函数中，设定了复杂的指令插桩条件，只有当下述条件满足的时候才会计数：

命令是 mov 指令、是一条内存读指令、指令的第一个操作数是寄存器、指令的第二个操作数 是内存。实际上，通过组合这些 API 就可以非常精确地筛选出想要插桩的指令了。



重新编译生成 Pintool 动态链接库，然后重新对 First 进行插桩





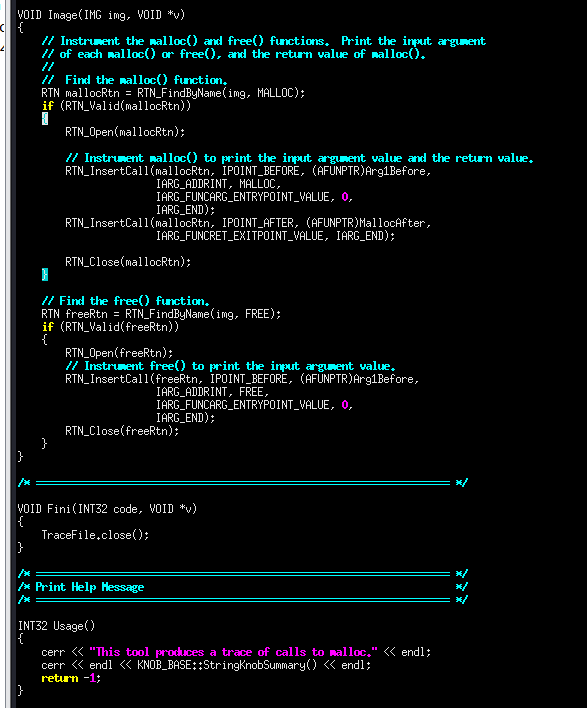
执行结果为：Count

2792

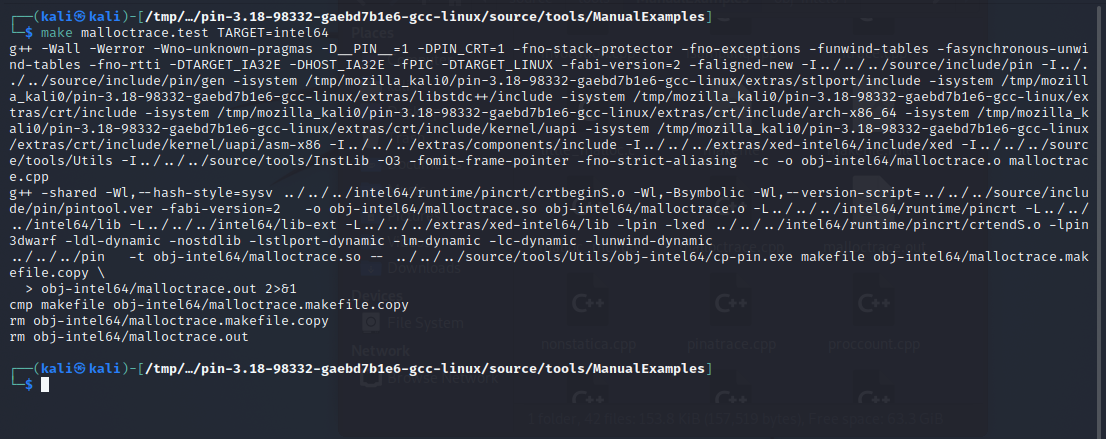
类似的，接下来复现malloctracePinTool

首先编译运行，产生动态链接库，以使用PinTool

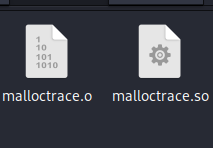
其代码如下



使用命令：make malloctrace.test TARGET=intel64 指定编译malloctrace

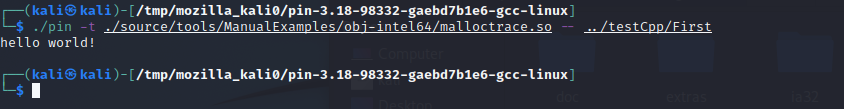


产生动态链接库

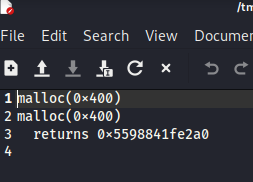
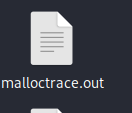


同样对 First 可执行程序进行程序插桩, Pin 命令为：

./pin -t ./source/tools/ManualExamples/obj-intel64/malloctrace.so -- ../testCpp/First



查看输出文件



**心得体会：**

在进行程序插桩及hook实验的过程中，我深刻认识到了程序的运行原理和系统的安全机制。通过插桩和hook技术，我们可以实现对程序的监控和控制，达到对系统的保护和防范攻击的目的。

在实验过程中，我首先学习了程序插桩技术。通过在程序中添加额外的代码，我们可以实现对程序的监控和调试。程序插桩技术可以在程序运行时动态地修改代码，并将修改后的代码加载到程序中。通过插桩工具，我可以实现对程序中的函数进行监控和修改，修改后的代码可以直接影响程序的运行。

另一方面，我还学习了hook技术。hook技术是一种通过修改函数调用地址的方式，实现对程序行为的控制和监控。通过hook技术，我们可以截获程序的函数调用，并对调用的参数和返回值进行修改和监控。通过hook工具，我可以实现对程序中的函数进行截获和修改，实现对程序的控制和监控。

通过这些实验，我深刻认识到了程序的运行机制和系统的安全机制。在实际的应用中，我们可以利用这些技术，提高系统的安全性和稳定性。同时，我也认识到了程序插桩和hook技术的潜在风险和安全隐患，需要在使用时谨慎处理，避免对系统造成不必要的损失和风险。