南開大學
Nankai University

计算机学院

软件安全实验报告

实验八：程序插桩及 Hook 实验

姓名：林盛森

学号：2312631

专业：计算机科学与技术

2025 年 5 月 7 日

目录

1 实验名称

程序插桩及 Hook 实验

2 实验要求

复现实验一，基于 Windows MyPinTool 或在 Kali 中复现 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。

3 实验过程

3.1 在 kali 中安装 Pin

在官网下载 pin 压缩包至本地，再从本地 windows 机器中拖入到 kali 虚拟机中从而完成 pin 压缩包的下载。

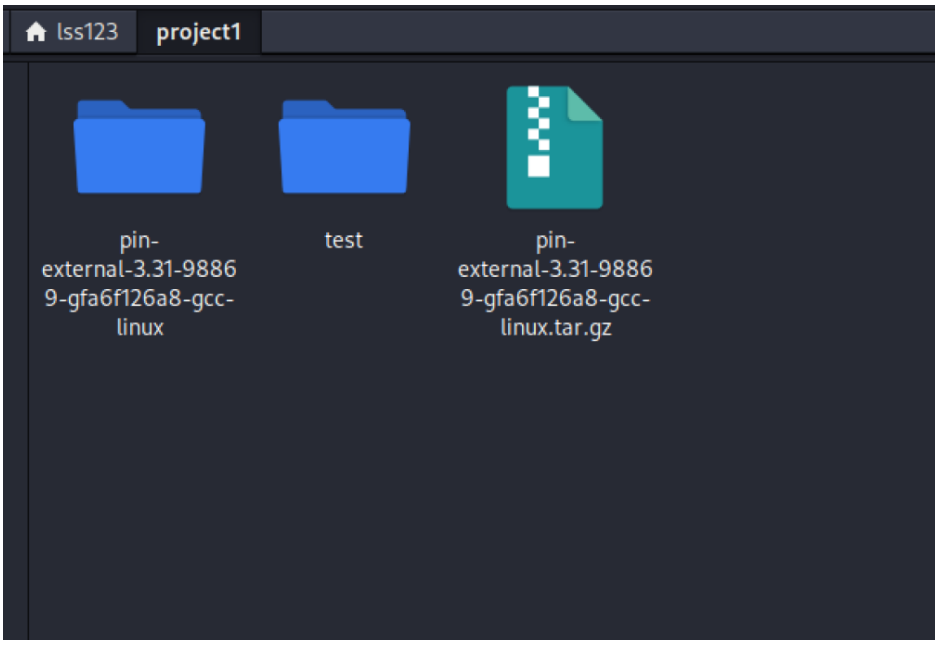

图 3.1

对其进行解压缩，进入到目录。

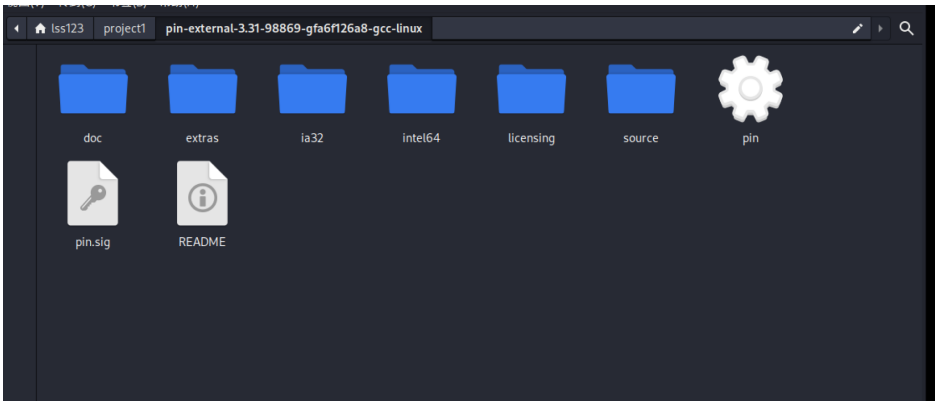

图 3.2

在子目录 source 下依次查看 tools，可以在 manualexamples 文件夹中看到很多默认已经编写好的 pintool，以.cpp 的格式存在，我们要想使用的话，得对其进行编译生成.so 文件（kali 系统下的动态链接库文件格式）。

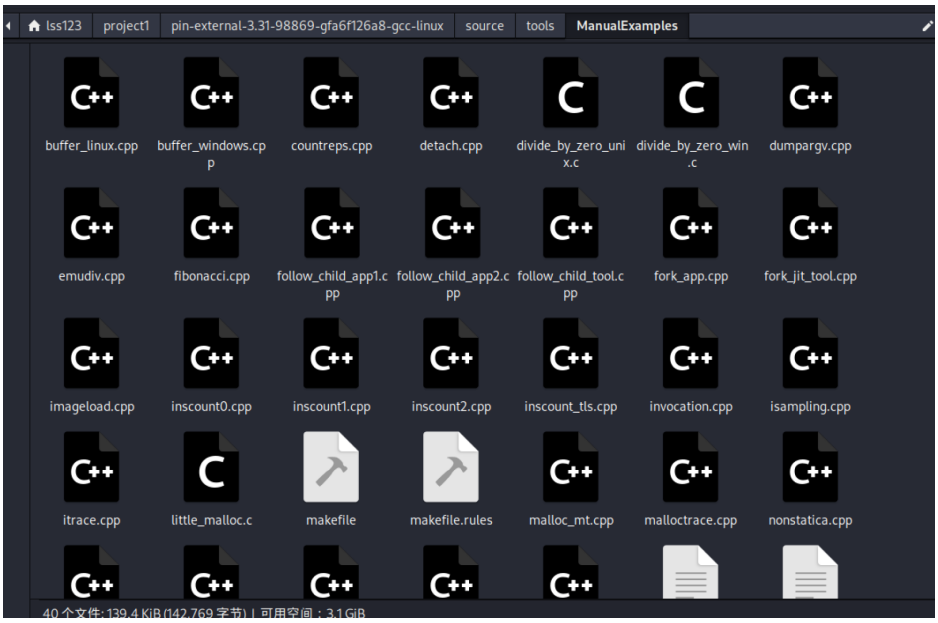

图 3.3

3.2 使用 inscount0 获取某个程序的信息

首先对 inscount0.cpp 进行编译。

命令：make inscount0.test TARGET=intel64

图 3.4

得到动态链接库 inscount0.so。

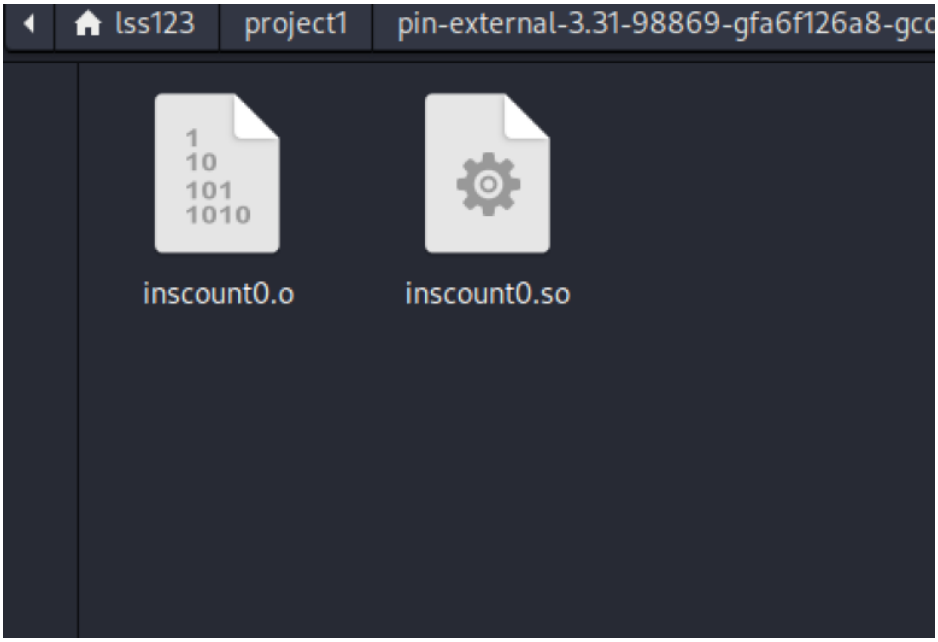

图 3.5

创建一个程序并编译。编译命令：gcc -o mytest test.c.

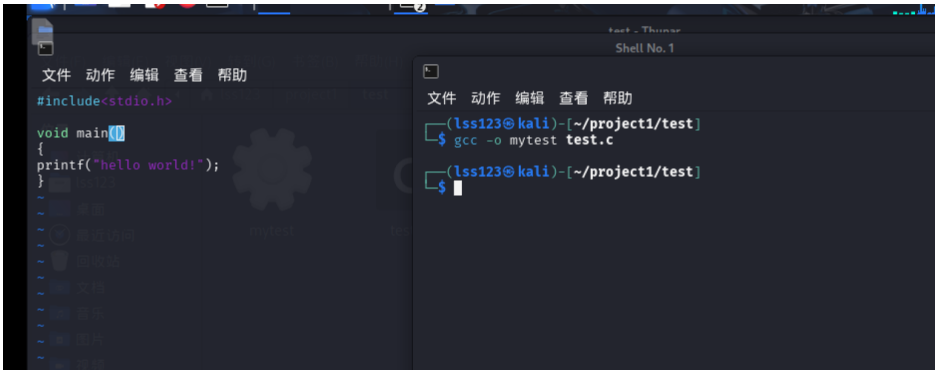

图 3.6

图 3.7

使用 pintool inscount0 对 mytest 程序进行指令数的测量。
命令：./pin -t ./source/tools/ManualExamples/obj-intel64/inscount0.so – ../test/mytest

图 3.8

产生输出文件 inscount0.cout，里面存了相关信息，这里就存储了指令的数量的信息。

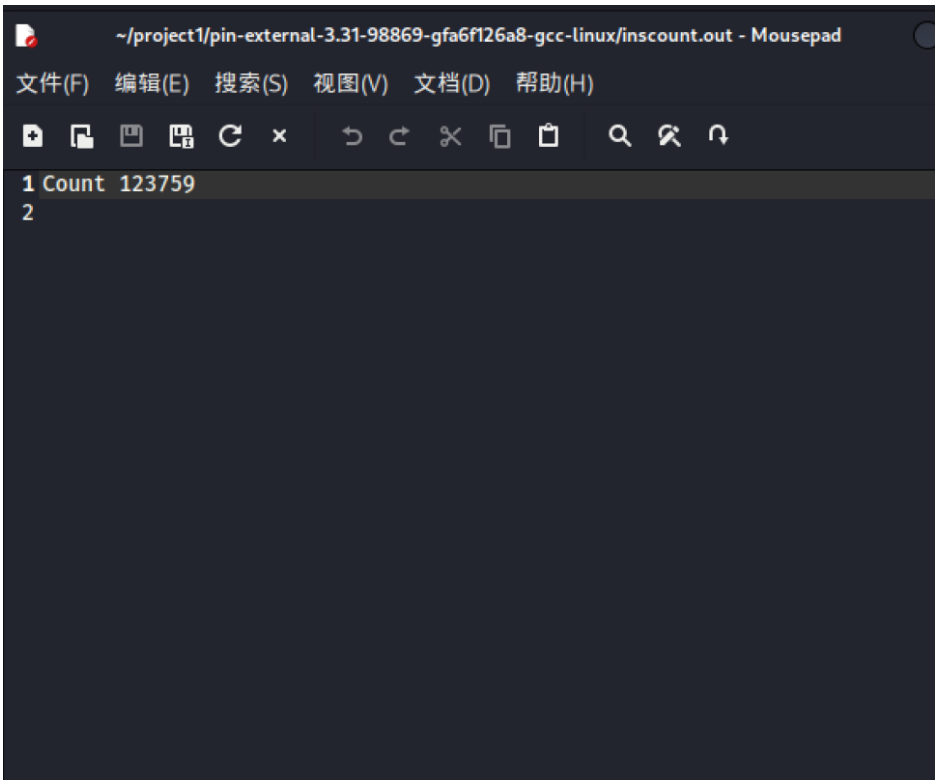

图 3.9

还可以对 pintool 进行修改编写自己的逻辑，添加相关条件判断进行筛选等。

3.3 复现 malloctrace

3.3.1 查看 malloctrace.cpp

```
#include "pin.H"
#include <iostream>
#include <fstream>
using std::cerr;
using std::endl;
using std::hex;
using std::ios;
using std::string;

/* ================================================================== */
/* Names of malloc and free */
/* ================================================================== */
#if defined(TARGET_MAC)
#define MALLOC "_malloc"
#define FREE "_free"
#else
#define MALLOC "malloc"
#define FREE "free"
#endif

/* ================================================================== */
/* Global Variables */
/* ================================================================== */

std::ofstream TraceFile;

/* ================================================================== */
/* Commandline Switches */
/* ================================================================== */

KNOB< string > KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "malloctrace.out",
    "specify trace file name");

/* ================================================================== */

/* ================================================================== */
/* Analysis routines                                                  */
/* ================================================================== */

VOID Arg1Before(CHAR* name, ADDRINT size) { TraceFile << name << "(" << size << ")"
    << endl; }

VOID MallocAfter(ADDRINT ret) { TraceFile << "  returns " << ret << endl; }
```

```
42
43 /* ===================================================================== */
44 /* Instrumentation routines                                              */
45 /* ===================================================================== */
46
47 VOID Image(IMG img, VOID* v)
48 {
49     // Instrument the malloc() and free() functions.  Print the input argument
50     // of each malloc() or free(), and the return value of malloc().
51     //
52     //  Find the malloc() function.
53     RTN mallocRtn = RTN_FindByName(img, MALLOC);
54     if (RTN_Valid(mallocRtn))
55     {
56         RTN_Open(mallocRtn);
57
58         // Instrument malloc() to print the input argument value and the return value.
59         RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before, IARG_ADDRINT,
               MALLOC, IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
60                         IARG_END);
61         RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
               IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
62
63         RTN_Close(mallocRtn);
64     }
65
66     // Find the free() function.
67     RTN freeRtn = RTN_FindByName(img, FREE);
68     if (RTN_Valid(freeRtn))
69     {
70         RTN_Open(freeRtn);
71         // Instrument free() to print the input argument value.
72         RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before, IARG_ADDRINT,
               FREE, IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
73                         IARG_END);
74         RTN_Close(freeRtn);
75     }
76 }
77
78 /* ===================================================================== */
79
80 VOID Fini(INT32 code, VOID* v) { TraceFile.close(); }
81
82 /* ===================================================================== */
83 /* Print Help Message                                                    */
84 /* ===================================================================== */
85
86 INT32 Usage()
87 {
```

```
88      cerr << "This tool produces a trace of calls to malloc." << endl;
89      cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
90      return -1;
91  }
92
93  /* ===================================================================== */
94  /* Main                                                                  */
95  /* ===================================================================== */
96
97  int main(int argc, char* argv[])
98  {
99      // Initialize pin & symbol manager
100     PIN_InitSymbols();
101     if (PIN_Init(argc, argv))
102     {
103         return Usage();
104     }
105
106     // Write to a file since cout and cerr maybe closed by the application
107     TraceFile.open(KnobOutputFile.Value().c_str());
108     TraceFile << hex;
109     TraceFile.setf(ios::showbase);
110
111     // Register Image to be called to instrument functions.
112     IMG_AddInstrumentFunction(Image, 0);
113     PIN_AddFiniFunction(Fini, 0);
114
115     // Never returns
116     PIN_StartProgram();
117
118     return 0;
119 }
```

两个分析函数：

Arg1Before：在 malloc 或 free 调用前触发，记录函数名和参数（malloc 的大小或 free 的地址）。

MallocAfter：在 malloc 调用后触发，记录返回的内存地址。

插桩函数：

Image 函数：在目标程序加载新镜像（如动态库）时被调用，用于向程序中找到 malloc 和 free 代码并进行插桩。

–插桩逻辑：查找 malloc 函数，在调用前插入 Arg1Before，调用后插入 MallocAfter。查找 free 函数，仅在调用前插入 Arg1Before。

Fini 函数：程序结束时关闭日志文件，也就是最后的输出文件。

在 main 函数中，我们可以对代码插桩的过程有更好的了解。首先，可以看到在 main 函数中使用了 Pin_Init 进行初始化，然后打开日志文件并设置十六进制输出格式，接着需要注册插桩函数，使用了 IMG_AddInstrumentFunction 这个函数，里面的参数是我们的插桩函数 Image，PIN_AddInstrumentFunction 注册退出函数，参数是 Fini 函数用于关闭输出文件，最后通过 PIN_StartProgram 启动目标程序。

3.3.2 编译

通过命令 make malloctrace.test TARGET=intel64 对 malloctrace.cpp 进行编译。

图 3.10

可以看到成功生成了动态链接库 malloctrace.so。

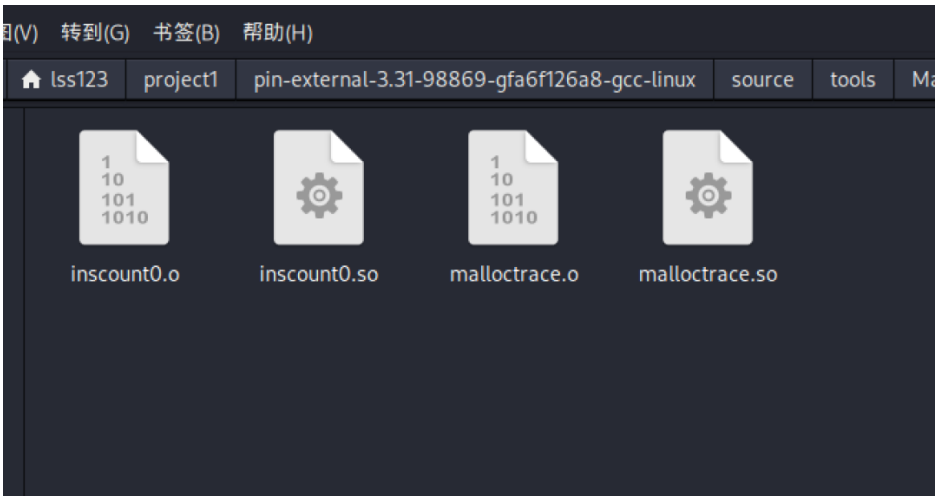

图 3.11

3.3.3 进行插桩实验

在编译成功的同时，生成了一个输出文件 malloctrace.out，我们可以进行查看，输出内容如下所示：

```
malloc(0x12000)
malloc(0x12000)
  returns 0x3afbe2a0
malloc(0x208)
malloc(0x208)
  returns 0x3afd02b0
malloc(0x1d8)
  returns 0x3afd04c0
malloc(0x2000)
malloc(0x2000)
  returns 0x3afd06a0
malloc(0x200)
```

```
malloc(0x200)
  returns 0x3afd26b0
malloc(0x1d8)
  returns 0x3afd28c0
malloc(0x2000)
malloc(0x2000)
  returns 0x3afd2aa0
free(0x3afd06a0)
free(0x3afd06a0)
free(0x3afd04c0)
free(0x3afd2aa0)
free(0x3afd2aa0)
free(0x3afd28c0)
```

在程序刚加载的时候，会调用 malloc 函数进行相关堆内存空间的开辟，malloc 函数括号里的是分配的堆内存空间的大小，return 返回的是堆空间的起始地址；在程序结束时，会调用 free 函数对开辟的内存进行释放，free 函数括号里是要释放的空间的起始地址。通过输出信息，我们可以发现，malloc 函数每次调用时，通过插桩，我们可以知道生成的堆空间的大小，堆空间的起始地址，释放堆空间的起始地址等。通过插桩操作 malloctrace，我们获取了堆内存分配释放时的相关信息。

4 心得体会

1. 对程序插桩的流程有了更好的理解,其核心框架就是首先要初始化,接着调用 IMG_AddInstrumentFunction 注册相关的插桩函数,调用 PIN_AddInstrumentFunction 注册相关的退出函数,调用 PIN_StartProgram 启动目标程序，学会了这种核心逻辑之后我们可以编写自己的 pintool。

2. 对 linux 命令行的操作更加熟练，例如利用命令行编译程序，利用./pin -t 去使用 pintool，后面跟着 pintool 和目标程序。

3. 学会了如何进行程序插桩，首先对 pintool 进行编译生成动态链接库（windows 下为.dll，linux 下为.so），对目标程序进行编译（如果需要的话），再使用./pin -t 进行程序插桩。