



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	2
2 实验要求	2
3 实验过程	2
3.1 在 kali 中安装 Pin	2
3.2 使用 inscount0 获取某个程序的信息	3
3.3 复现 malloctrace	6
3.3.1 查看 malloctrace.cpp	6
3.3.2 编译	9
3.3.3 进行插桩实验	9
4 心得体会	10

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。

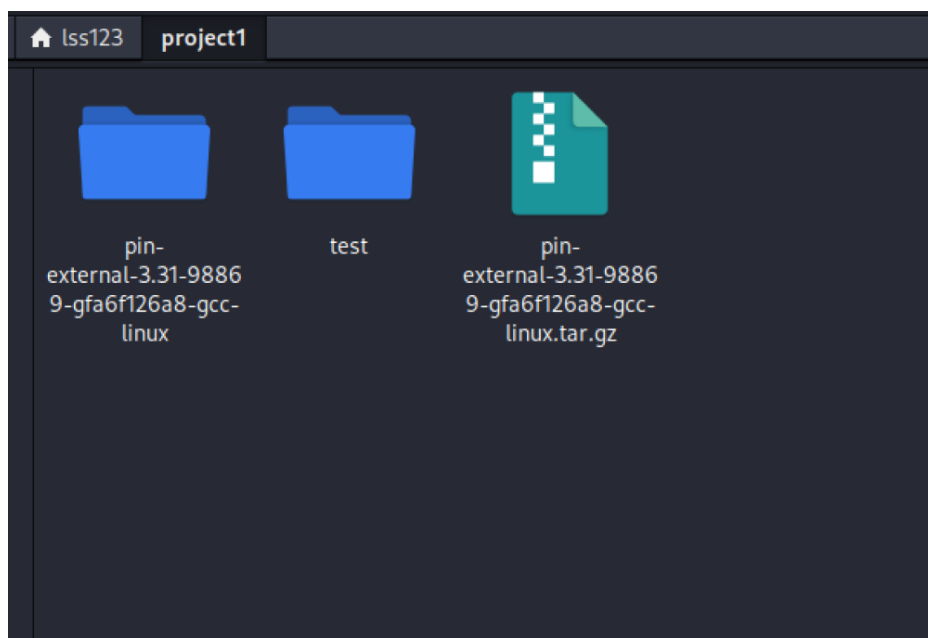


图 3.1

对其进行解压缩，进入到目录。

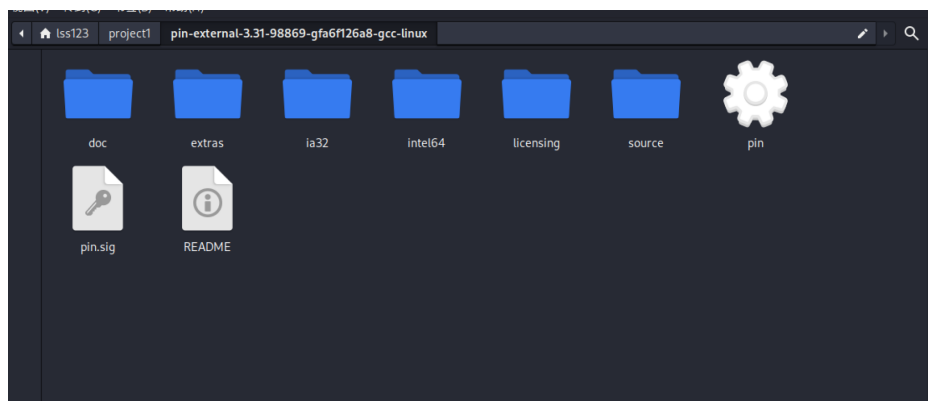


图 3.2

在子目录 source 下依次查看 tools，可以在 manuaexamples 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）。

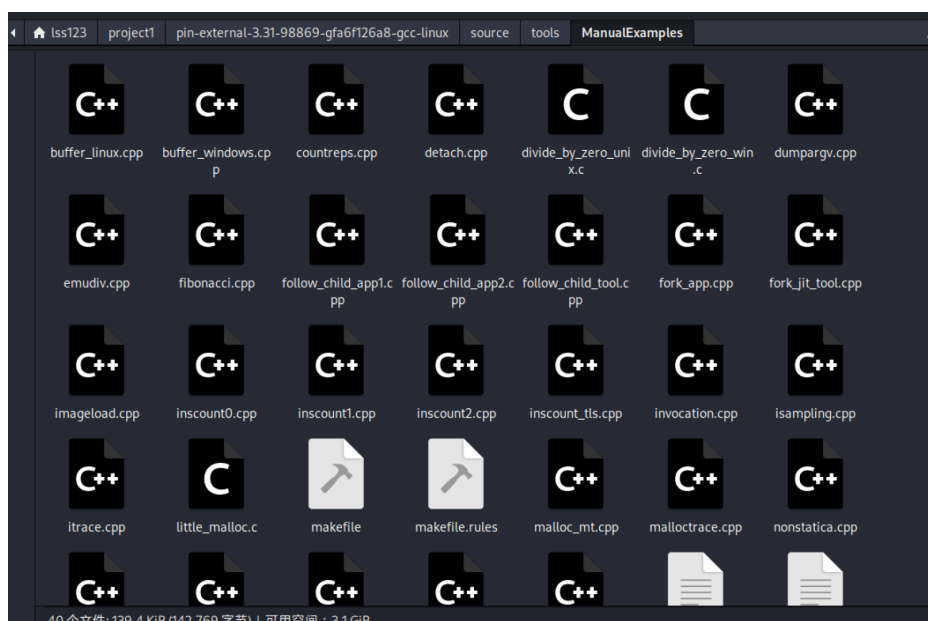


图 3.3

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

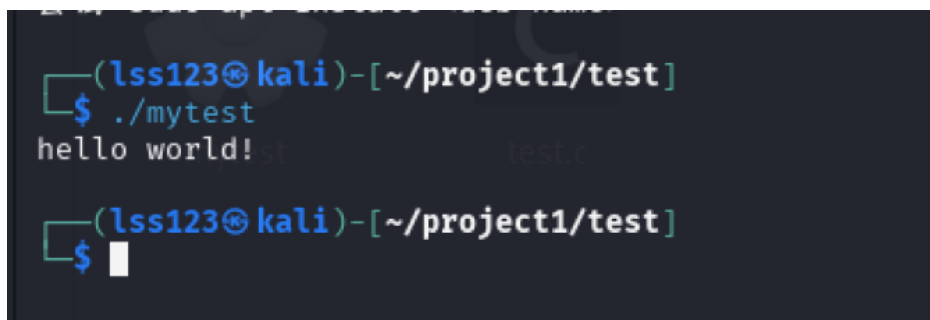


图 3.7

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

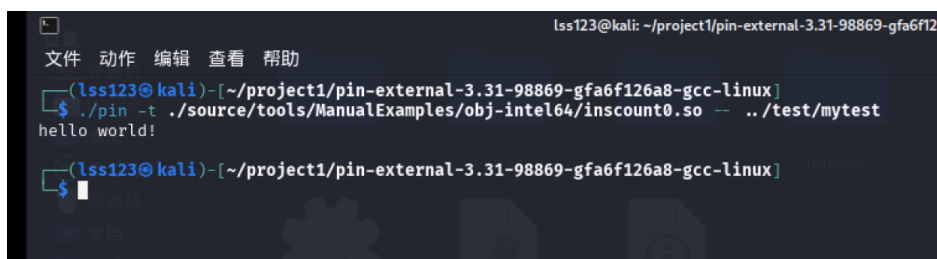


图 3.8

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。

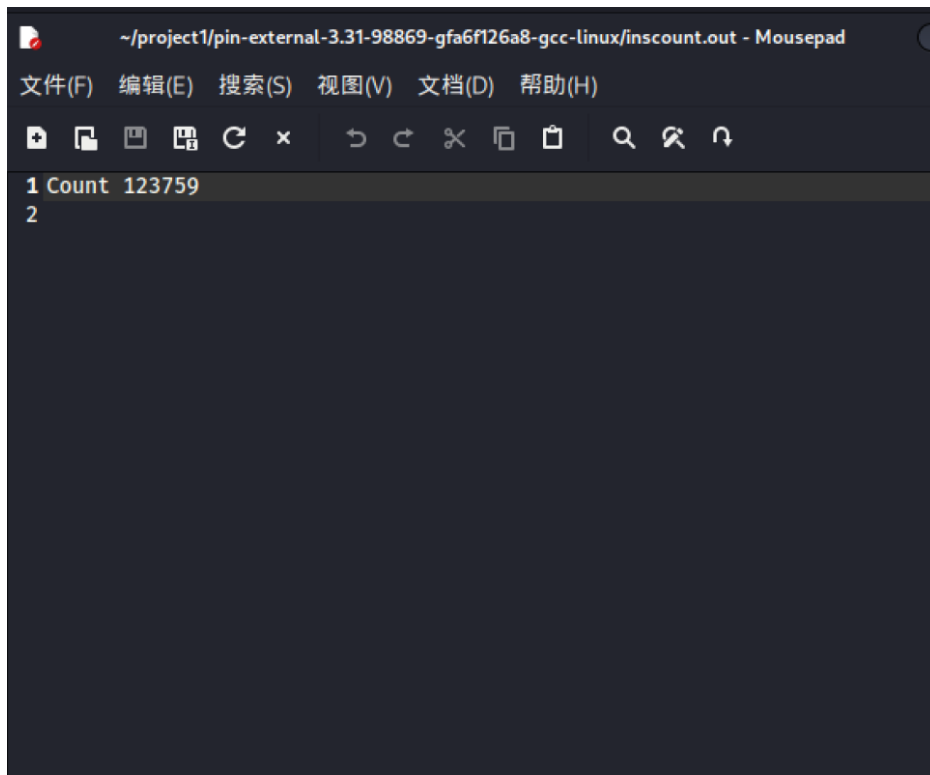


图 3.9

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

```

1 #include "pin.H"
2 #include <iostream>
3 #include <fstream>
4 using std::cerr;
5 using std::endl;
6 using std::hex;
7 using std::ios;
8 using std::string;
9
10 /* ===== */
11 /* Names of malloc and free */
12 /* ===== */
13 #if defined(TARGET_MAC)
14 #define MALLOC "_malloc"
15 #define FREE "_free"
16 #else
17 #define MALLOC "malloc"
18 #define FREE "free"
19 #endif
20
21 /* ===== */
22 /* Global Variables */
23 /* ===== */
24
25 std::ofstream TraceFile;
26
27 /* ===== */
28 /* Commandline Switches */
29 /* ===== */
30
31 KNOB< string > KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "malloctrace.out",
    "specify trace file name");
32
33 /* ===== */
34
35 /* ===== */
36 /* Analysis routines */
37 /* ===== */
38
39 VOID Arg1Before(CHAR* name, ADDRINT size) { TraceFile << name << "(" << size << ")"
    << endl; }
40
41 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_ADDDRINT,
60             MALLOC, IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
61             IARG_END);
62         RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR) MallocAfter,
63             IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);
64
65         RTN_Close(mallocRtn);
66     }
67
68     // Find the free() function.
69     RTN_freeRtn = RTN_FindByName(img, FREE);
70     if (RTN_Valid(freeRtn))
71     {
72         RTN_Open(freeRtn);
73         // Instrument free() to print the input argument value.
74         RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR) Arg1Before, IARG_ADDDRINT,
75             FREE, IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
76             IARG_END);
77         RTN_Close(freeRtn);
78     }
79 }
80
81 /* ===== */
82
83 VOID Fini(INT32 code, VOID* v) { TraceFile.close(); }
84
85 /* ===== */
86 /* Print Help Message */
87 /* ===== */
88
89 INT32 Usage()
90 {

```



```

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。

```

[ss123@kali] ~/pin-external-3.31-98869-gfa6f126a8-gcc-linux/source/tools/ManualExamples
$ make malloctrace.test TARGET=intel64
g++ -Wall -Werror -Wno-unknown-pragmas -DPIN_CRT=1 -fno-stack-protector -fno-exceptions -funwind-tables -fasynchronous-unwind-tables -fno-rtti -DTARGET_I
2E -DHOST_IA32E -fPIC -DTARGET_LINUX -fabi-version=2 -faligned-new -I../..../source/include/pin -I../..../source/include/pin/gen -isystem /home/ss123/p
object1/pin-external-3.31-98869-gfa6f126a8-gcc-linux/extras/cxx/include -isystem /home/ss123/project1/pin-external-3.31-98869-gfa6f126a8-gcc-linux/extras
rt/include -isystem /home/ss123/project1/pin-external-3.31-98869-gfa6f126a8-gcc-linux/extras/crt/include/arch-x86_64 -isystem /home/ss123/project1/pin-
ternal-3.31-98869-gfa6f126a8-gcc-linux/extras/crt/include/kernel/uapi -isystem /home/ss123/project1/pin-external-3.31-98869-gfa6f126a8-gcc-linux/extras/
t/include/kernel/uapi/asm-x86 -I../..../extras/components/include -I../..../extras/xed-intel64/include/xed -I../..../source/tools/Utils -I../..../sou
e/tools/IntelLib -O3 -fomit-frame-pointer -fno-strict-aliasing -Wno-dangling-pointer -c -o obj-intel64/malloctrace.o malloctrace.cpp
g++ -shared -Wl,-hash-style=sysv ../..../intel64/runtime/pincrt/crtbegin5.o -Wl,-Bsymbolic -Wl,-version-script=../..../source/include/pin/pintool.ver
fabi-version=2 -o obj-intel64/malloctrace.so obj-intel64/malloctrace.o -L../..../intel64/runtime/pincrt -L../..../intel64/lib -L../..../intel64/lib-
t -L../..../extras/xed-intel64/lib -lpin -lxd ../..../intel64/runtime/pincrt/crtend5.o -lpindwarf -ldwarf -ldl -dynamic -nostdlib -lc++ -lm-dy
mic -lc -dynamic -lunwind-dynamic
/usr/bin/ld: warning: util_host_ia32e.sdp.o: missing .note.GNU-stack section implies executable stack
/usr/bin/ld: NOTE: This behaviour is deprecated and will be removed in a future version of the linker
../..../pin -t obj-intel64/malloctrace.so -- ../..../source/tools/Utils/obj-intel64/cp-pin.exe makefile obj-intel64/malloctrace.makefile.copy \
> obj-intel64/malloctrace.out 2>&1
cmp makefile obj-intel64/malloctrace.makefile.copy
rm obj-intel64/malloctrace.makefile.copy
rm obj-intel64/malloctrace.out
[ss123@kali] ~/pin-external-3.31-98869-gfa6f126a8-gcc-linux/source/tools/ManualExamples
$
  
```

图 3.10

可以看到成功生成了动态链接库 `malloctrace.so`。

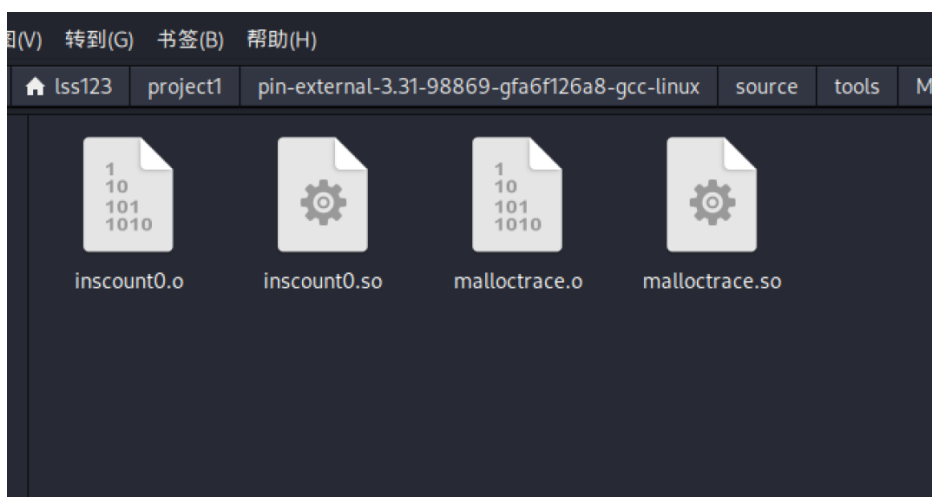


图 3.11

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：

```

1 malloc(0x12000)
2 malloc(0x12000)
3   returns 0x3afbe2a0
4 malloc(0x208)
5 malloc(0x208)
6   returns 0x3afd02b0
7 malloc(0x1d8)
8   returns 0x3afd04c0
9 malloc(0x2000)
10 malloc(0x2000)
11   returns 0x3afd06a0
12 malloc(0x200)
  
```

```
13 malloc(0x200)
14     returns 0x3afd26b0
15 malloc(0x1d8)
16     returns 0x3afd28c0
17 malloc(0x2000)
18 malloc(0x2000)
19     returns 0x3afd2aa0
20 free(0x3afd06a0)
21 free(0x3afd06a0)
22 free(0x3afd04c0)
23 free(0x3afd2aa0)
24 free(0x3afd2aa0)
25 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。