《软件安全》实验报告

姓名：王昱　　学号：2212046　　班级：信息安全班

一、实验名称：

程序插桩及 Hook 实验

二、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。

三、实验原理：

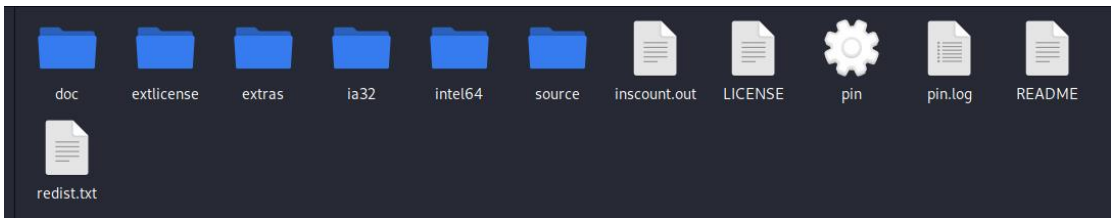
Pin 是由英特尔开发的一个强大的动态二进制插桩工具，广泛用于程序分析、性能优化和调试。它允许用户在不修改源代码或二进制文件的情况下，在运行时插入自定义的分析代码（称为 Pintool）。下面是 Pin 的特点：

- 动态插桩：Pin 可以在程序运行时动态地插入分析代码，而无需对源代码进行修改。这使得它非常灵活，适用于现有的二进制程序。
- 平台支持：Pin 支持多种平台，包括 x86 和 x86-64 架构的 Windows、Linux 和 macOS。
- 透明性：被插桩的程序的行为基本不受影响，Pin 尽量保证程序的性能和正确性不被显著改变。
- 丰富的 API：Pin 提供了丰富的 API，允许用户在程序的任意位置插入自定义代码，并访问程序的内部状态和上下文。

四、实验过程：

（一）在 Linux 环境下安装 Pin

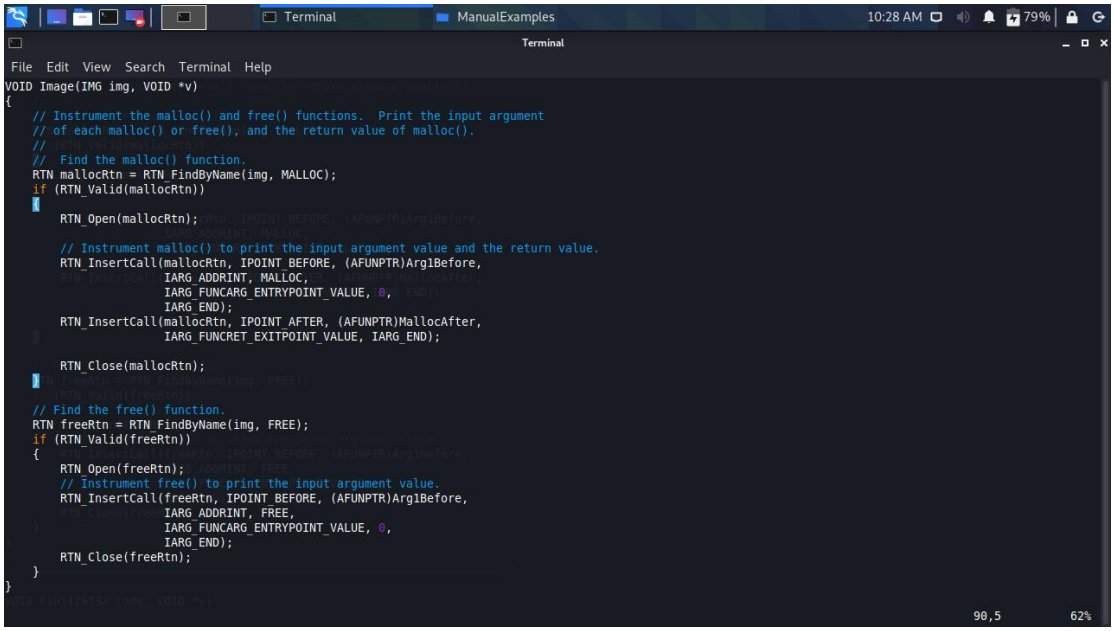
在官网中下载 Linux 版本的 Pin 并且放入 Kali 虚拟机中，如下图所示：



（二）找到 malloctrace 的代码

通过以下路径找到 malloctrace 的 Pintool:

/home/kali/pin-3.7/source/tools/ManualExamples/

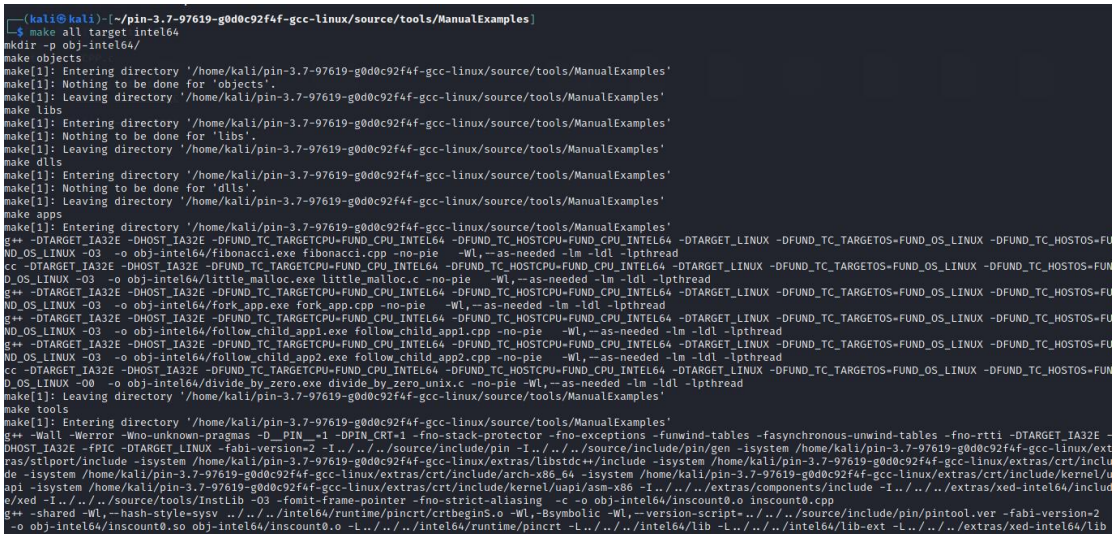


（三）编译 Pintool

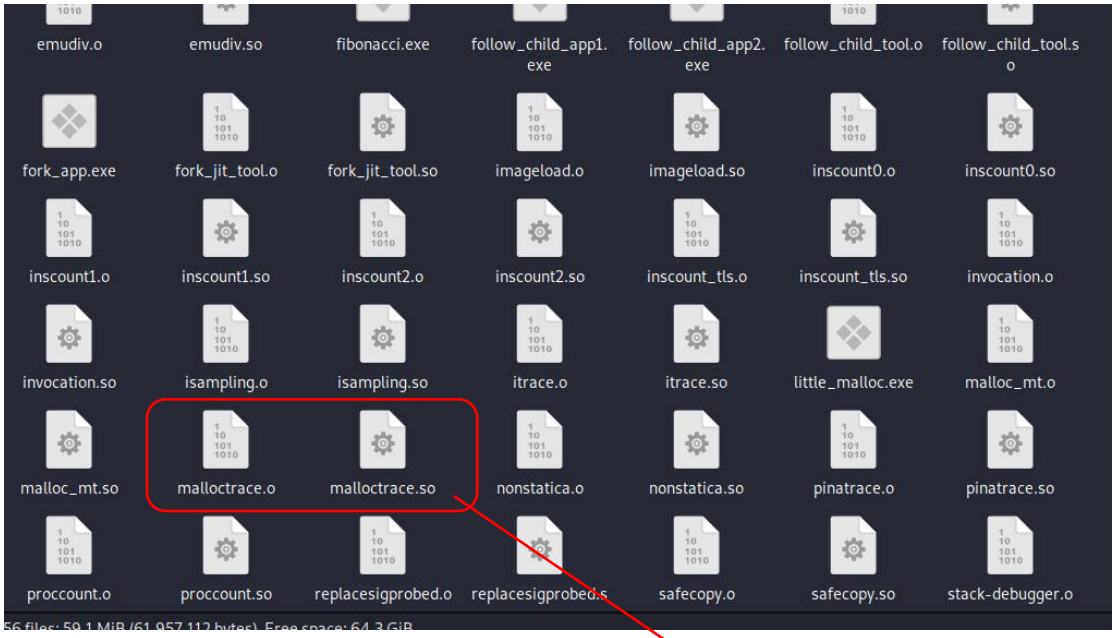
首先进行编译，在 Pintool 所在文件夹打开终端后输入命令：

Make all TARGET=intel64

编译所有的 Pintool，编译结果如下所示：



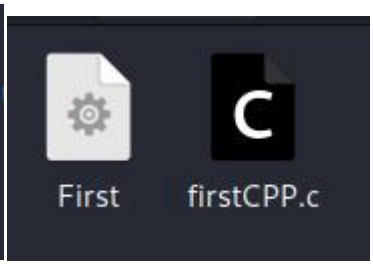
编译后即可找到动态链接库和 malloctrace.co，malloctrace.o：



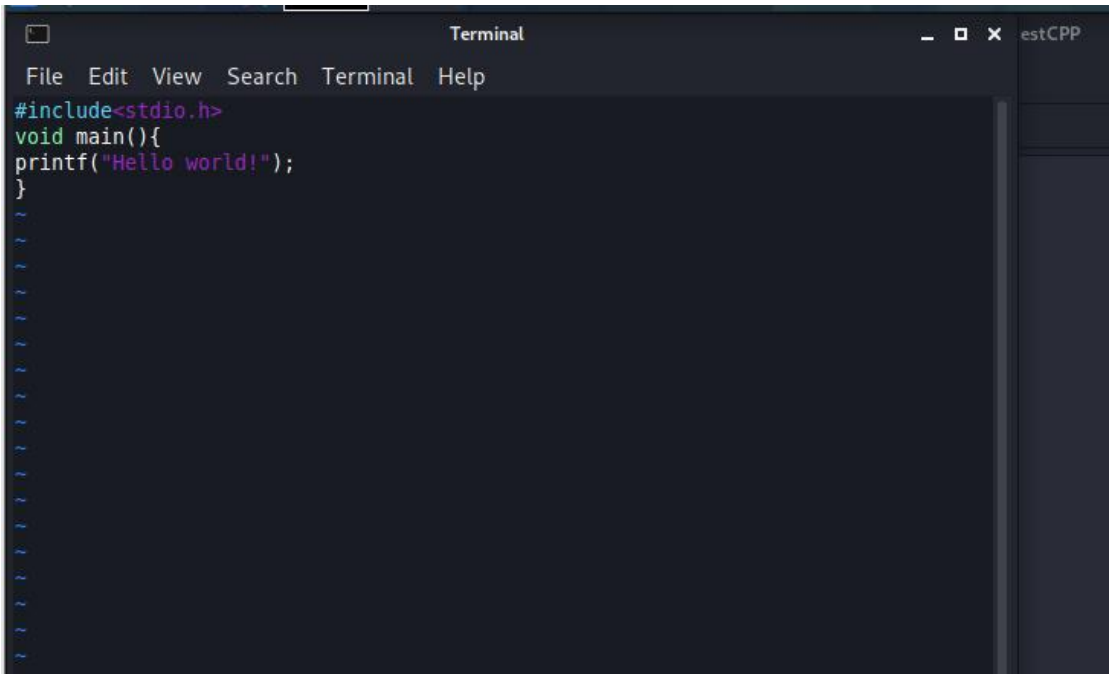
框中即为我们这次实验需要的动态链接库文件

（四）测试 Pintool

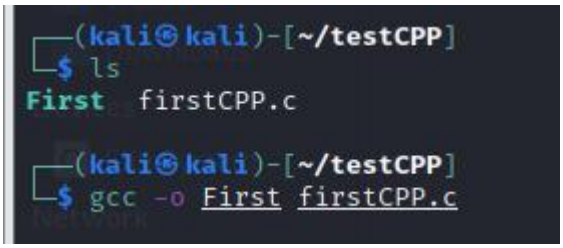
创建一个 testCPP 文件夹，并且写测试程序 firstCPP.c



firstCPP.c 内容如下：



编译 firstCPP.c,输入命令：gcc-o First firstCPP.c

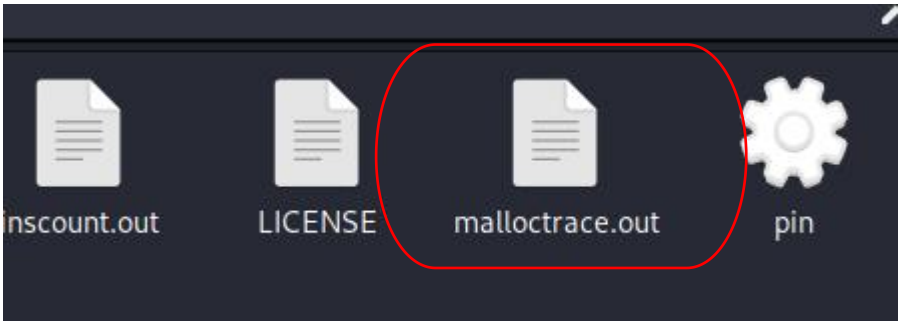


输入命令：

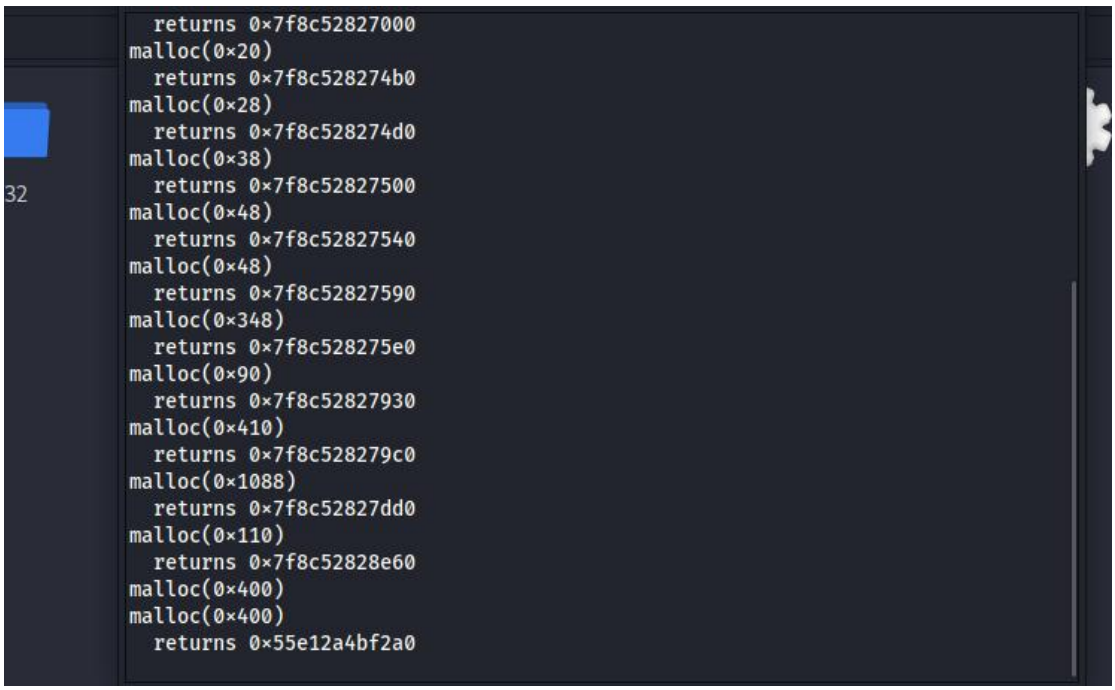
./pin -t ./source/tools/ManualExamples/obj-intel64/malloctrace.so -- ../testCPP/First

如图，成功执行文件。

同时，在指定文件夹下生成了 malloctrace.out 文件：



其内容如下图所示：



这个输出结果显示了程序在运行时进行的多次 malloc 调用及其返回的内存地址。每一行表示一次 malloc 调用，包括请求的内存大小和分配的内存地址。具体解释如下：

malloc(0x20)：表示调用了 malloc 函数来分配 0x20 字节的内存。

returns 0x7f8c52827000：表示 malloc 调用成功并返回了内存地址 0x7f8c52827000。

以下是具体的解析：

malloc(0x20) returns 0x7f8c52827000：请求分配 32 字节的内存，返回的内存地址是 0x7f8c52827000。

malloc(0x28) returns 0x7f8c528274b0：请求分配 40 字节的内存，返回的内存地址是 0x7f8c528274b0。

malloc(0x38) returns 0x7f8c528274d0：请求分配 56 字节的内存，返回的内存地址是 0x7f8c528274d0。

依此类推，输出中每对 malloc 和 returns 表示一次内存分配请求及其结果。通过分析这些数据，开发者可以了解程序在运行过程中内存分配的情况，比如分配了多少内存，每次分配的大小和地址等。这对于调试和优化内存使用非常有用，特别是在检测内存泄漏和不合理的内存分配时。

五、实验分析：

```
#include "pin.H"
#include <iostream>
#include <fstream>
```

```
4.
5.  std::ofstream TraceFile;
6.
7.  VOID Arg1Before(CHAR* name, ADDRINT size) {
8.      TraceFile << name << "(" << size << ")" << std::endl;
9.  }
10.
11. VOID MallocAfter(ADDRINT ret) {
12.     TraceFile << " returns " << ret << std::endl;
13. }
14.
15. VOID Image(IMG img, VOID* v) {
16.     RTN mallocRtn = RTN_FindByName(img, MALLOC);
17.     if (RTN_Valid(mallocRtn)) {
18.         RTN_Open(mallocRtn);
19.         RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before, IARG_A
    DDRINT, MALLOC, IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
20.         RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter, IARG_F
    UNCRET_EXITPOINT_VALUE, IARG_END);
21.         RTN_Close(mallocRtn);
22.     }
23.
24.     RTN freeRtn = RTN_FindByName(img, FREE);
25.     if (RTN_Valid(freeRtn)) {
26.         RTN_Open(freeRtn);
27.         RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before, IARG_ADD
    RINT, FREE, IARG_FUNCARG_ENTRYPOINT_VALUE, 0, IARG_END);
28.         RTN_Close(freeRtn);
29.     }
30. }
31.
32. VOID Fini(INT32 code, VOID* v) {
33.     TraceFile.close();
34. }
35.
36. INT32 Usage() {
37.     cerr << "This tool produces a trace of calls to malloc." << endl;
38.     cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
39.     return -1;
40. }
41.
42. int main(int argc, char* argv[]) {
43.     // 初始化 Pin 和符号管理器
44.     PIN_InitSymbols();
```

```
45.     if (PIN_Init(argc, argv)) {
46.         return Usage();
47.     }
48.
49.     // 打开输出文件
50.     TraceFile.open(KnobOutputFile.Value().c_str());
51.     TraceFile << hex;
52.     TraceFile.setf(ios::showbase);
53.
54.     // 注册回调函数
55.     IMG_AddInstrumentFunction(Image, 0);
56.     PIN_AddFiniFunction(Fini, 0);
57.
58.     // 启动程序
59.     PIN_StartProgram();
60.
61.     return 0;
62. }
```

- **相关 API**

1. PIN_InitSymbols()
2. PIN_Init()　初始化操作
3. IMG_AddInstrumentFunction()　镜像级插桩，注册一个插桩函数。
4. PIN_AddFiniFunction()　注册一个程序退出时的回调函数用来进行一些结束处理。
5. PIN_StartProgram()　使用该函数来启动需要进行插桩的程序。
6. RTN_FindByName()　该函数能够找到执行函数的标识。
7. RTN_InsertCall()　注册一个回调函数，在某一指令执行时，执行指定的操作

- **Pin 插桩的核心步骤：**

在 main 函数中：

1、初始化。通过调用函数 PIN_Init 完成初始化。

2、注册插桩函数。通过使用 INS_AddInstrumentFunction 注册一个插桩函数，其第 2 个 参数为一个额外传递给 Instruction 的参数，即对应 VOID *v 这个参数，这里没有使用。 而 Instruction 接受的第一个参数为 Image 结构，用来表示 malloc 和 free 的调用。

3、注册退出回调函数。通过使用 PIN_AddFiniFunction 注册一个程序退出时的回调函 数 Fini，当应用退出的时候会调用函数 Fini。

4、启动程序。使用函数 PIN_StartProgram 启动程序。

五、心得体会：

通过本次实验，不仅学会了使用 Pin 插桩工具跟踪 malloc 和 free 函数，还加深了对动态二进制插桩技术的理解，提升了编写高效分析工具的能力。这些收获为将来在程序性能优化、内存管理分析和安全性检查等领域的进一步探索打下了坚实基础。