《软件安全》实验报告

姓名：许洋　学号：2313721 班级：1070

实验名称：

程序插桩及Hook实验

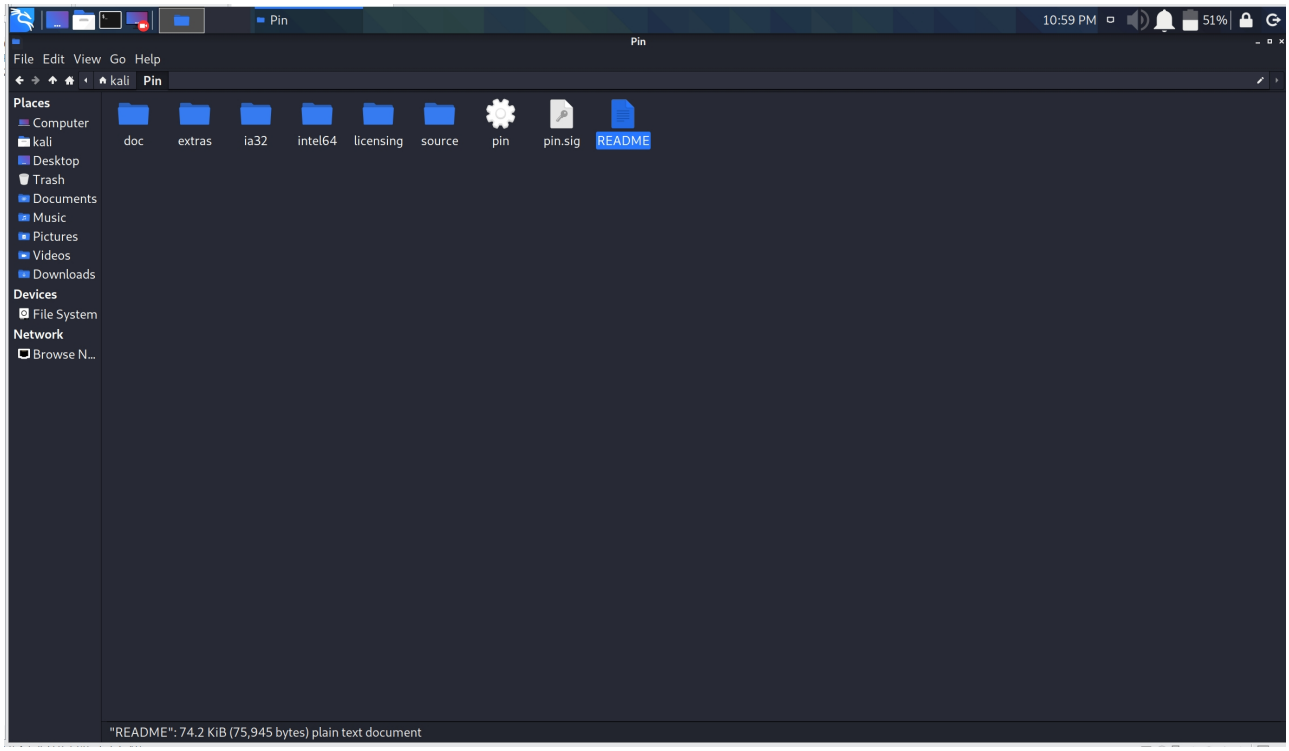
实验要求：

复现实验一，基于WindowsMyPinTool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。

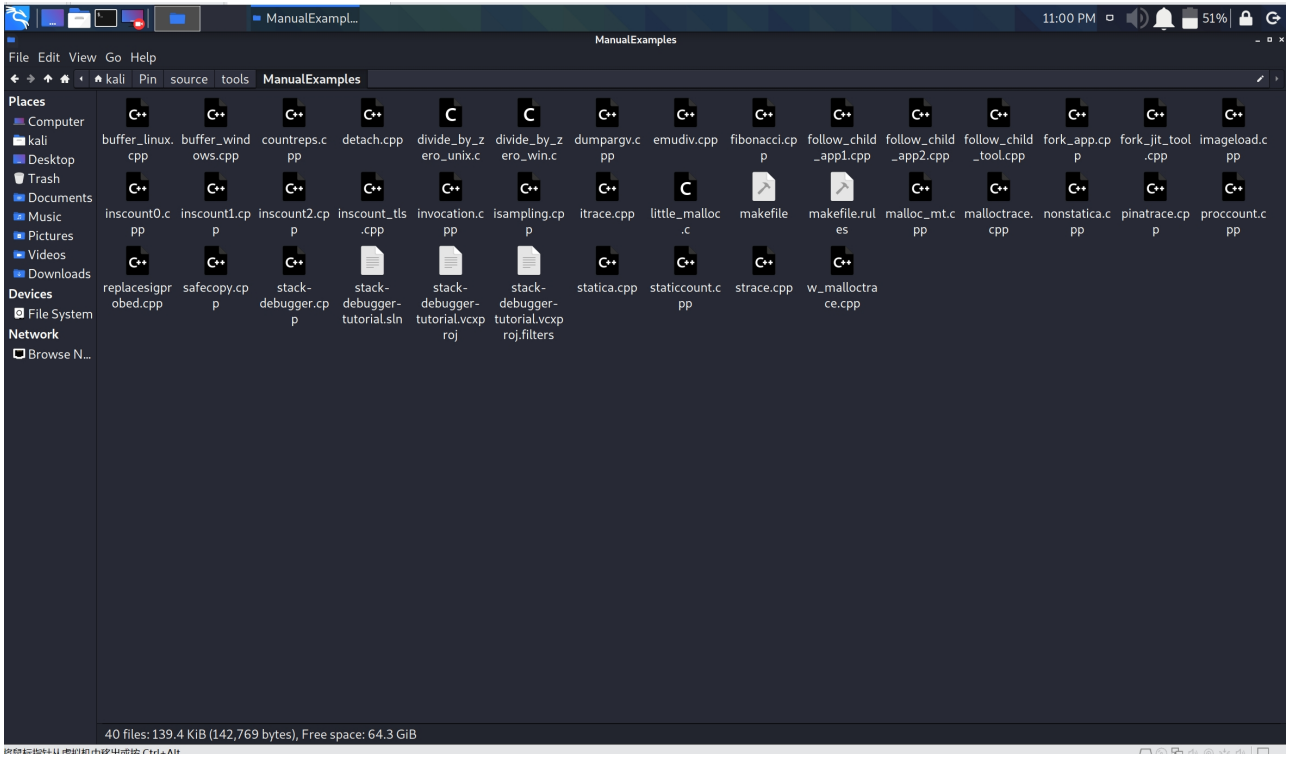
实验过程：

在kali虚拟机中安装Pin

首先，我们需要安装Pin，但是因为我们在虚拟机上连不上网，所以我选择了在windows系统下先进行下载，然后解压缩拖到kali_linux系统中。我们打开文件夹，发现里面有以下这些文件。



我们按照给出的路径，打开文件夹中的source中的tools的ManualExamples，就可以查看到很多已经编写好的pintool，如下所示：

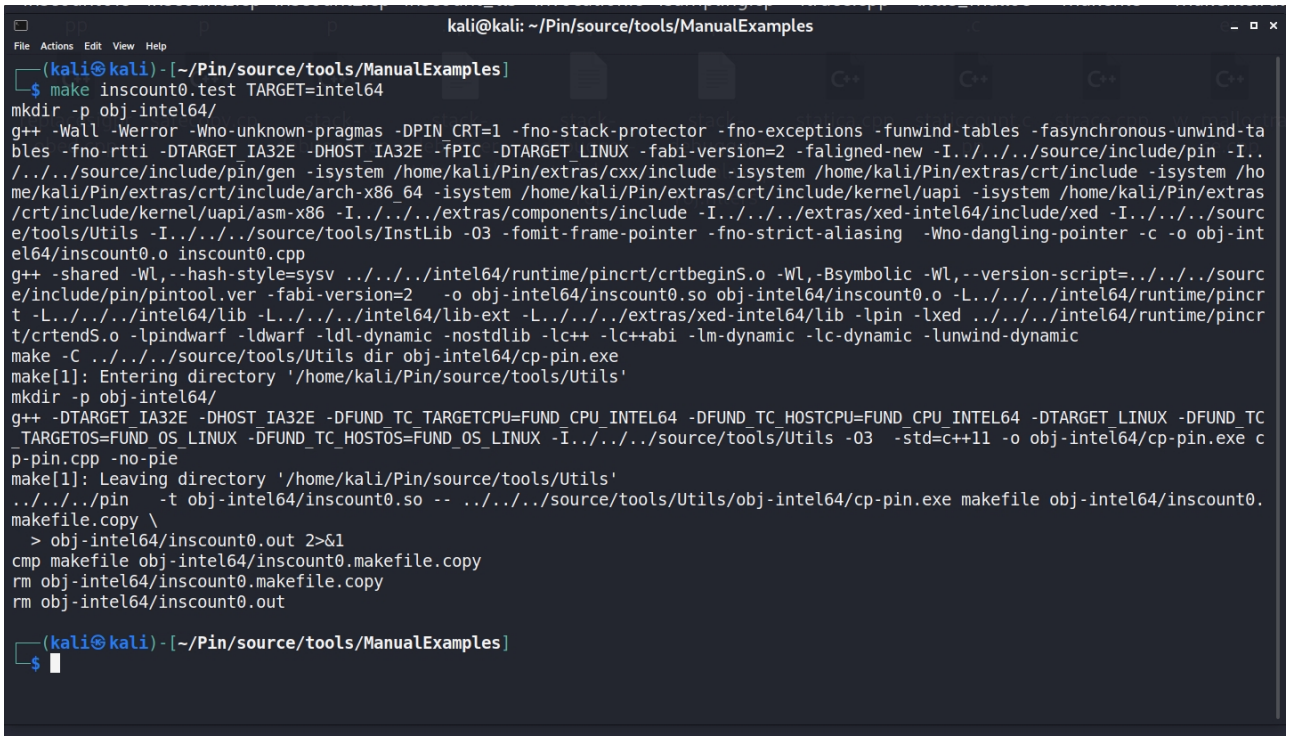


以上就是我们的Pin的安装流程，接下来我们就可以开始实验了。

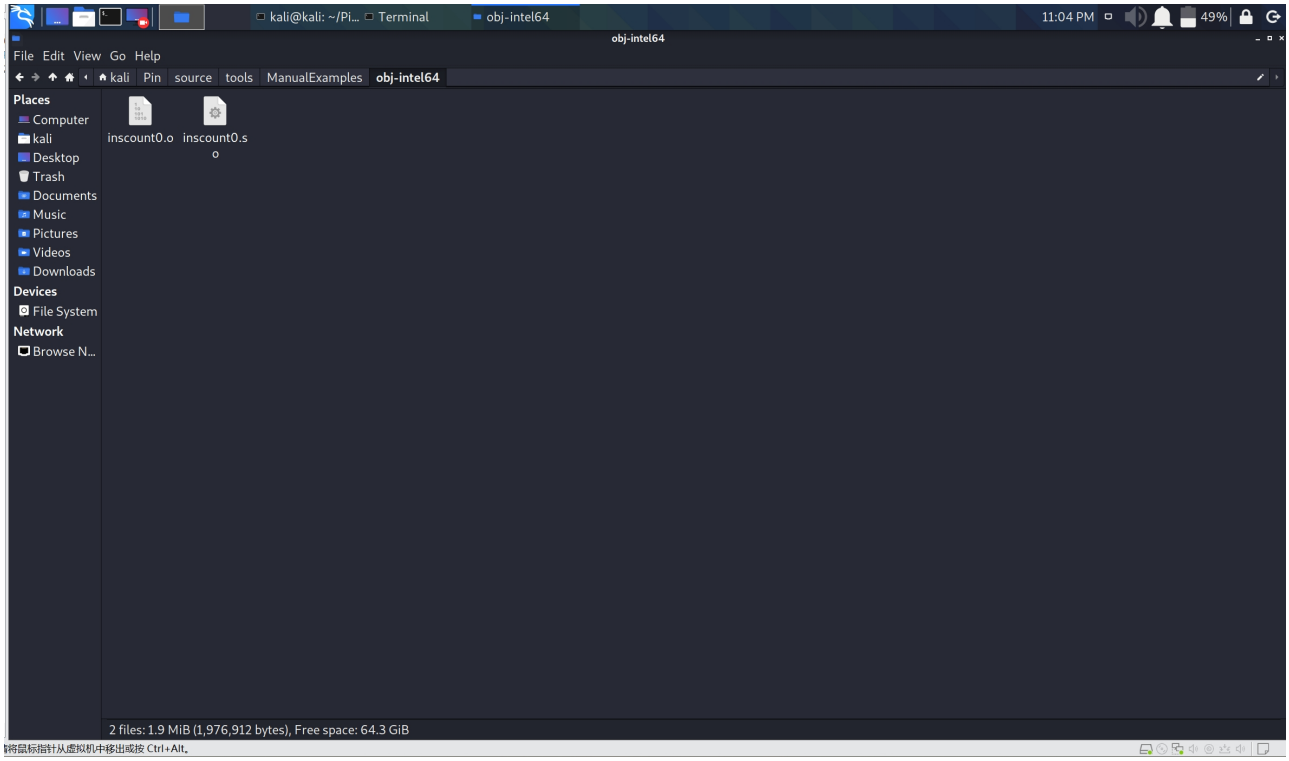
Pintool——malloctrace的使用

编译inscount0.cpp

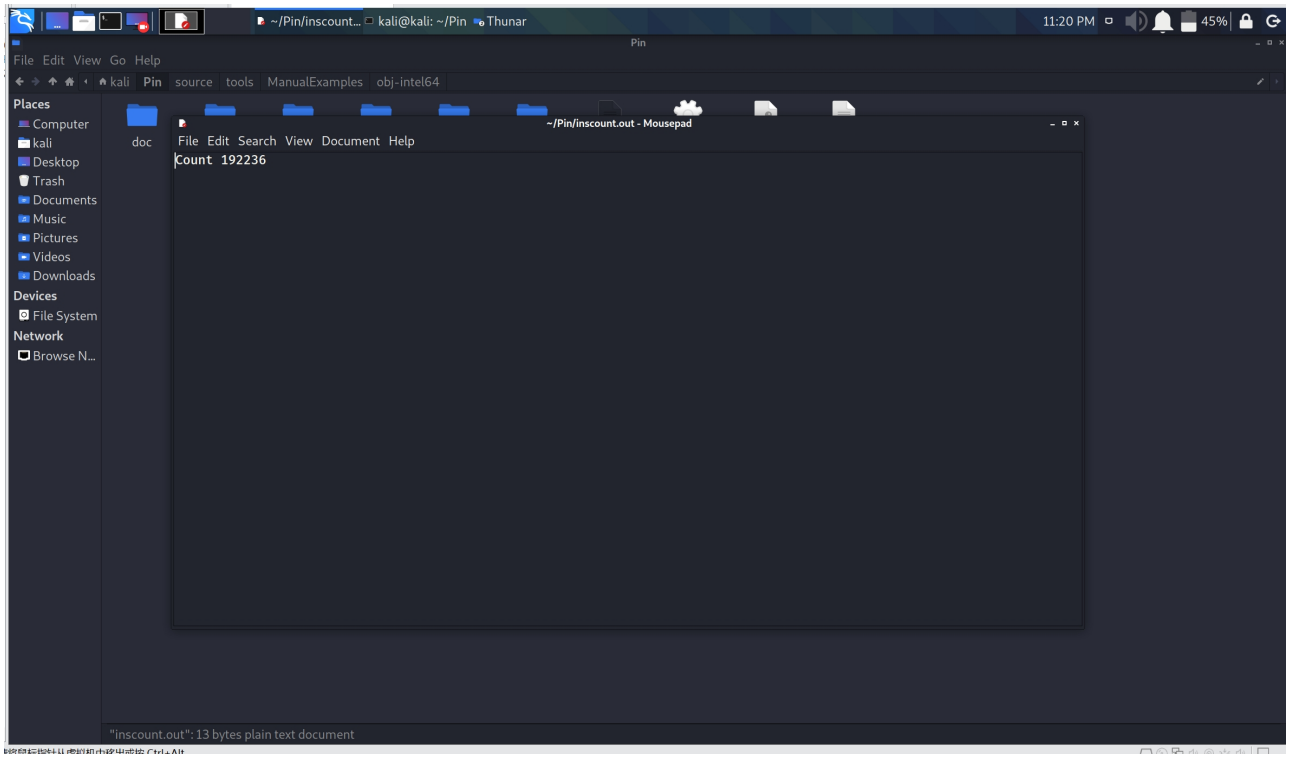
我们首先打开文件夹ManualExamples，选择文件inscount0.cpp，输入命令行make inscount0.test TARGET=intel64进行编译，输出如下所示：



我们进入文件夹查看，发现文件夹中多出了一个已经完成编译的inscount0.so文件，说明文件编译完成！

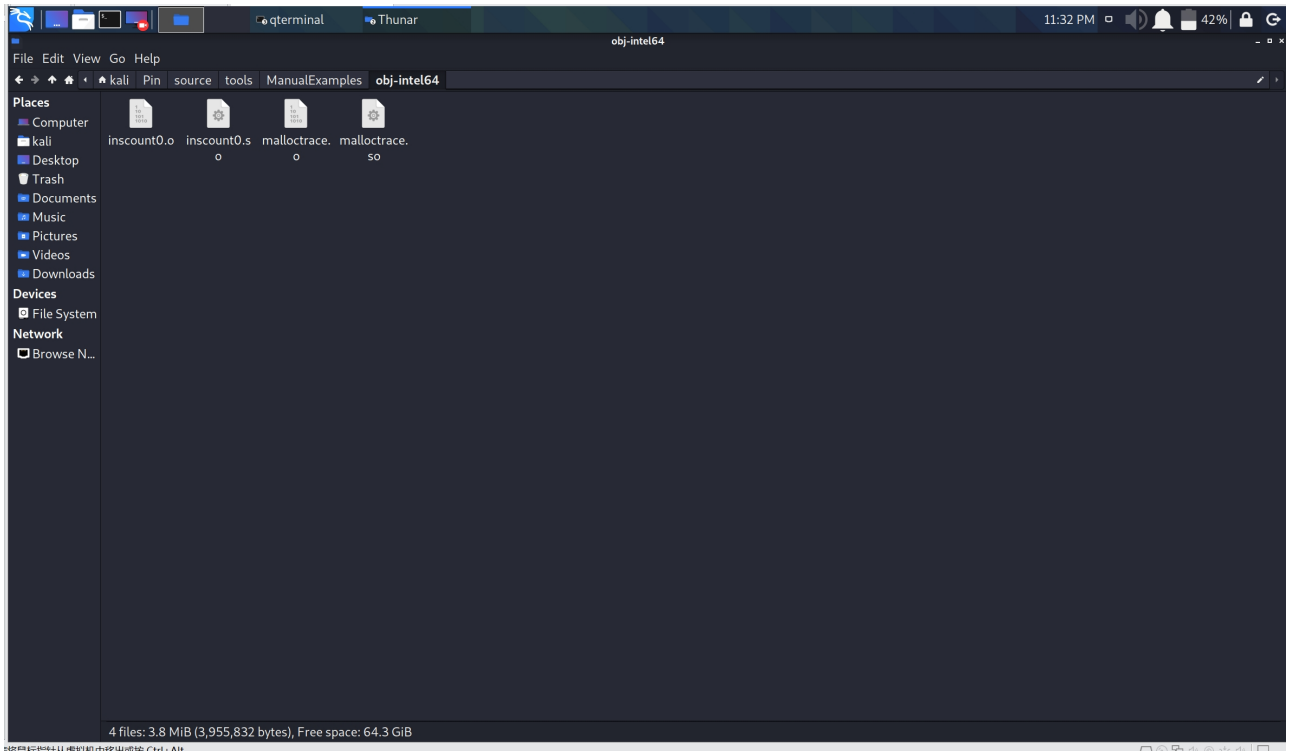


我们还可以查看编译运行后的out文件来查看编译所使用的指令数。我们打开文件inscount.out，就可以看到输出的count数量了。



编译malloctrace.cpp产生动态链接库

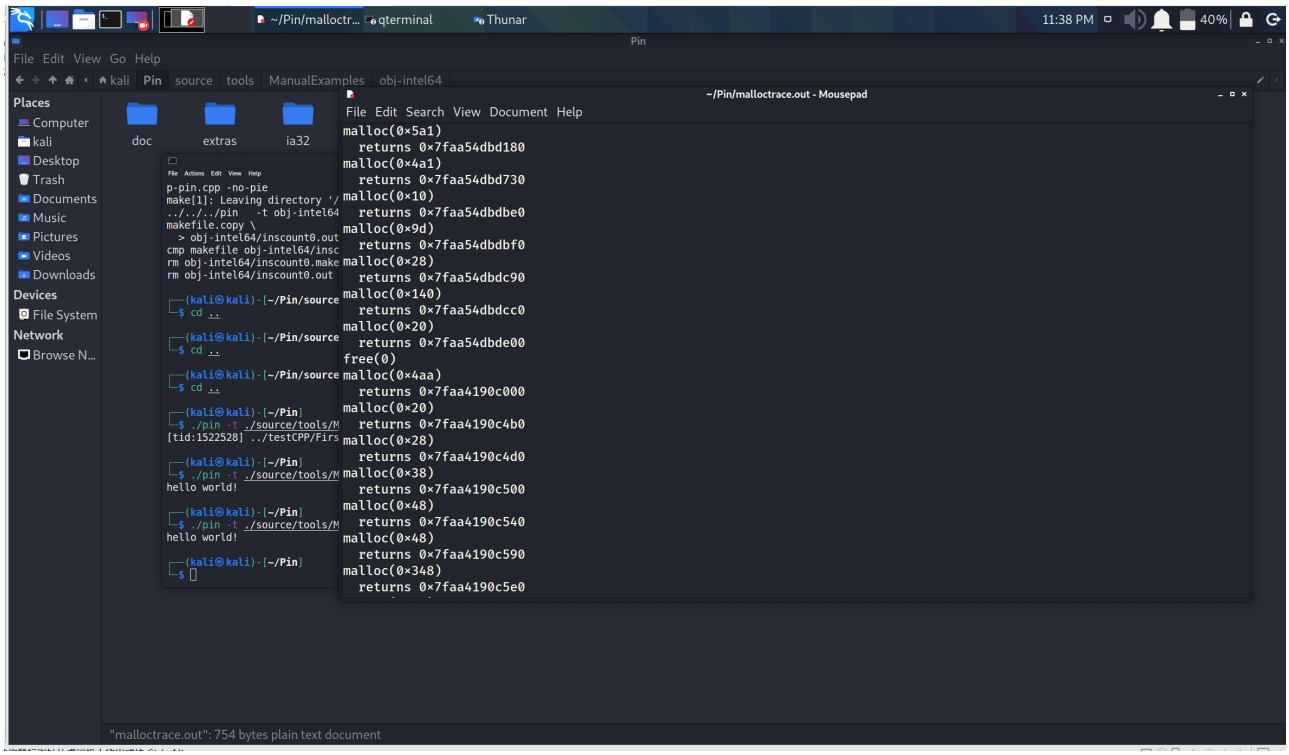
我们按照实验要求，打开source/tools/ManualExamples，打开malloctrace.cpp进行查看源码，之后我们输入命令行make malloctrace.test TARGET=intel64，编译运行该文件，得到以下的结果：



我们看到其中产生了malloctrace.so文件，即成功产生了动态链接库。

进行插桩实验

在进行完以上的步骤之后，我们就可以应用这个tool来进行插桩实验了！我们上一部分已经完成了对文件的编译，我们直接使用上面的编译生成的out文件进行查看，发现确实生成了一大串文字！我们打开文件，查看里面的内容，发现其输出了malloc和free函数的具体数值。



我们发现输出的文字由malloc函数、returns、free函数构成，我们在下一个部分我们具体进行介绍。可以发现，每次使用malloc申请堆内存空间的时候，都会输出申请的空间的大小，并且得到申请的空间的起始地址。当使用free释放内存空间的时候，会输出释放的空间的起始地址。

malloc和free函数输入输出的理解

我们根据上面的输出内容来进一步分析。

```
malloc(0x5a1)
  returns 0x7faa54dbd180
malloc(0x4a1)
  returns 0x7faa54dbd730
malloc(0x10)
  returns 0x7faa54dbdbe0
malloc(0x9d)
  returns 0x7faa54dbdbf0
malloc(0x28)
  returns 0x7faa54dbdc90
malloc(0x140)
```

```
  returns 0x7faa54dbdcc0
malloc(0x20)
  returns 0x7faa54dbde00
free(0)
malloc(0x4aa)
  returns 0x7faa4190c000
malloc(0x20)
  returns 0x7faa4190c4b0
malloc(0x28)
  returns 0x7faa4190c4d0
malloc(0x38)
  returns 0x7faa4190c500
malloc(0x48)
  returns 0x7faa4190c540
malloc(0x48)
  returns 0x7faa4190c590
malloc(0x348)
  returns 0x7faa4190c5e0
malloc(0x90)
  returns 0x7faa4190c930
malloc(0x410)
  returns 0x7faa4190c9c0
malloc(0x1088)
  returns 0x7faa4190cdd0
malloc(0x110)
  returns 0x7faa4190de60
malloc(0x400)
malloc(0x400)
  returns 0x5624eb0c22a0
```

首先，前面的malloc都是系统读入程序后，进行的默认的内存空间的分配，括号中的内容应该是内存空间的大小，returns的值应该是内存进行存储的地址。然后，在使用完内存空间之后，程序会使用free函数进行释放，注意，释放的时候是读入内存，进行对应位置的内存释放即可。

Pintool基本框架

我们在此处分析一下Pintool的基本框架与源代码。

首先是malloctrace的分析。

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

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

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

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

VOID Image(IMG img, VOID* v)
{
    RTN mallocRtn = RTN_FindByName(img, MALLOC);
    if (RTN_Valid(mallocRtn))
    {
        RTN_Open(mallocRtn);
        RTN_InsertCall(mallocRtn, IPOINT_BEFORE,
(AFUNPTR)Arg1Before, IARG_ADDRINT, MALLOC,
IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
        RTN_InsertCall(mallocRtn, IPOINT_AFTER,
(AFUNPTR)MallocAfter, IARG_FUNCRET_EXITPOINT_VALUE, IARG_END);

        RTN_Close(mallocRtn);
    }

    RTN freeRtn = RTN_FindByName(img, FREE);
    if (RTN_Valid(freeRtn))
    {
        RTN_Open(freeRtn);
        RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before,
IARG_ADDRINT, FREE, IARG_FUNCARG_ENTRYPOINT_VALUE, 0,
                       IARG_END);
```

```
        RTN_Close(freeRtn);
    }

}

VOID Fini(INT32 code, VOID* v) { TraceFile.close(); }

INT32 Usage()
{
    cerr << "This tool produces a trace of calls to malloc." <<
endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl;
    return -1;
}

int main(int argc, char* argv[])
{
    // Initialize pin & symbol manager
    PIN_InitSymbols();
    if (PIN_Init(argc, argv))
    {
        return Usage();
    }

    // Write to a file since cout and cerr maybe closed by the
application
    TraceFile.open(KnobOutputFile.Value().c_str());
    TraceFile << hex;
    TraceFile.setf(ios::showbase);

    // Register Image to be called to instrument functions.
    IMG_AddInstrumentFunction(Image, 0);
    PIN_AddFiniFunction(Fini, 0);

    // Never returns
    PIN_StartProgram();

    return 0;

}
```

两个分析函数是 Arg1Before 和 MallocAfter。Arg1Before 在调用 malloc 或free之前执行，用于记录函数名称和参数。MallocAfter 在调用 malloc 之后执行，记录返回值。Image 函数是用于在程序映像中找到并插桩 malloc 和 free 函数的代码。它会在程序加载时被调用。对于找到的 malloc 和 free 函数，代码使用 RTN_InsertCall 在函数调用前后插入分析函数（如 Arg1Before 和 MallocAfter）。Fini 函数在 Pin 工具完成后关闭输出文件。Usage 函数提供了一个帮助信息，当命令行参数有误时显示。

通过学习这个程序的Pintool的基本框架，我们可以了解一般的Pintool框架：

首先要进行初始化，通过调用函数PIN_Init实现注册插桩函数。通过使用XXX_AddInstrumentFunction注册一个插桩函数，由于使用了指令级插桩，因此在原始程序的每条指令执行前，都会进入到我们注册的插桩函数中，然后执行相应的操作。注册退出回调函数。通过使用PIN_AddFiniFunctino注册一个程序退出时的回调函数，当应用退出的时候会调用该函数。启动程序。使用函数PIN_StartProgram启动程序。

心得体会：

通过本次实验，我掌握了动态插桩工具Pin的核心原理与使用方法，并深入理解了内存管理监控在安全分析中的价值。

Pin工具实践
通过编写Pintool成功捕获了malloc/free函数的参数与返回值，体会到动态插桩无需修改源码即可监控程序行为的优势，其跨平台特性为分析复杂软件提供了便利。

内存行为可视化
实验中观察到内存分配地址与释放顺序，直观揭示了堆管理的底层机制，这种实时追踪能力为检测内存泄漏、Use-After-Free等漏洞奠定了基础。

关键技术点突破
学习到通过`RTN_InsertCall`插入钩子函数、利用`IARG_FUNCARG`捕获参数等核心API用法，同时解决了Kali环境配置与编译依赖问题，提升了跨平台调试能力。

安全分析启发
认识到插桩技术可扩展至监控敏感函数（如strcpy），未来计划结合符号执行技术探索更深层次的漏洞挖掘方法。