《软件安全》实验报告

姓名：蒋枘言　学号：2313546　班级：信息安全班

一、实验名称

程序插桩实验

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、实验过程

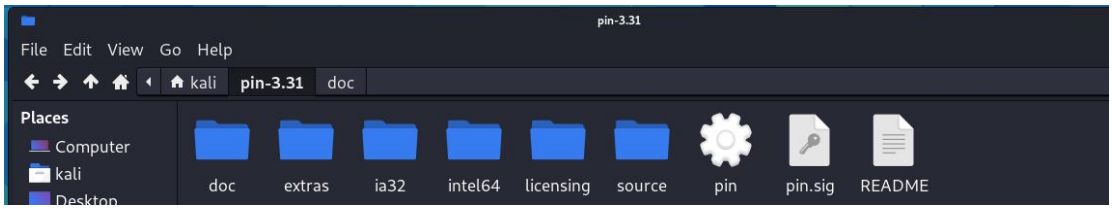
（一）安装 Pin：安装 Linux 版本的 Pin，安装完毕后拖到 Kali Linux 虚拟机中，然后解压。

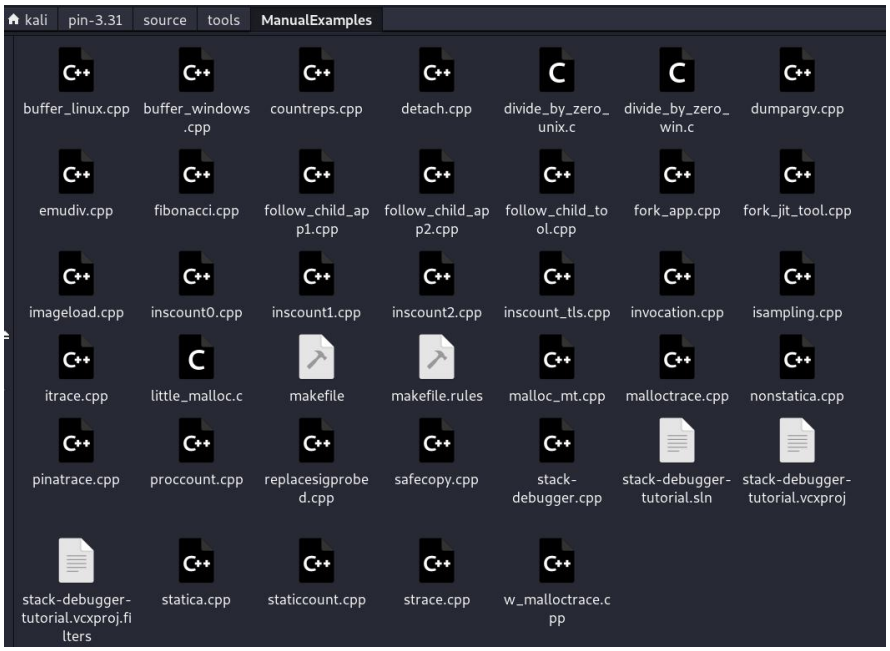
Linux*

IA32 and intel64 (x86 32 bit and 64 bit)

Version	Date	Kit	Signature	Documentation		
Pin 3.31	June 30, 2024	98869	sig	Manual	PinCRT	Release Notes

打开文件夹，可以看到在/source/tools/ManualExamples 里面提供了很多插桩程序的示例。





查看一下 inscount0.cpp 的源代码：

①static UINT64 icount 是静态全局变量存储指令计数。

②docount()是计数回调函数，每次调用时递增 icount，此函数会被插入到每条指令前执行。

③Instruction()是插桩回调函数，每当 Pin 发现一条新指令时，调用此函数。其中关键 API 是 INS_InsertCall()，参数 ins 是当前指令对象，IPOINT_BEFORE 是在指令执行前插入回调，(AFUNPTR)docount 是指定插入的函数指针，IARG_END 是参数列表结束标记。

④主函数 main()里的初始化流程为：

PIN_Init()：初始化 Pin 环境，解析命令行参数。

OutFile.open()：打开输出文件。

INS_AddInstrumentFunction()：注册指令级插桩回调。

PIN_AddFiniFunction()：注册程序结束回调。

PIN_StartProgram()：将控制权交给 Pin，启动目标程序。

（二）使用 PinTool（首先以 inscount0.cpp 为例）

现在想要编译 inscount0.cpp，执行指令 make inscount0.test TARGET=intel64 ，成功产生了动态链接库 inscount0.so（.so 是 Linux/Unix 系统的动态链接库文件，在程序运行时被动态加载到内存），结果如下：

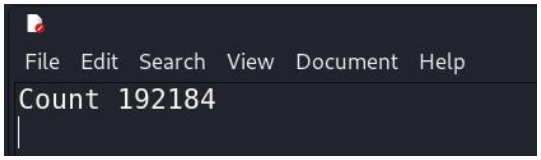
```
┌──(kali㉿kali)-[~/pin-3.31/source/tools/ManualExamples]
└─$ make inscount0.test TARGET=intel64
mkdir -p obj-intel64/
g++ -Wall -Werror -Wno-unknown-pragmas -DPIN_CRT=1 -fno-stack-protector -fno-exceptions -fu
nwind-tables -fasynchronous-unwind-tables -fno-rtti -DTARGET_IA32E -DHOST_IA32E -fPIC -DTAR
GET_LINUX -fabi-version=2 -faligned-new -I../../../source/include/pin -I../../../source/inc
lude/pin/gen -isystem /home/kali/pin-3.31/extras/cxx/include -isystem /home/kali/pin-3.31/e
xtras/crt/include -isystem /home/kali/pin-3.31/extras/crt/include/arch-x86_64 -isystem /hom
e/kali/pin-3.31/extras/crt/include/kernel/uapi -isystem /home/kali/pin-3.31/extras/crt/incl
ude/kernel/uapi/asm-x86 -I../../../extras/components/include -I../../../extras/xed-intel64/
include/xed -I../../../source/tools/Utils -I../../../source/tools/InstLib -O3 -fomit-frame-
pointer -fno-strict-aliasing  -Wno-dangling-pointer -c -o obj-intel64/inscount0.o inscount0
.cpp
g++ -shared -Wl,--hash-style=sysv ../../../intel64/runtime/pincrt/crtbeginS.o -Wl,-Bsymboli
c -Wl,--version-script=../../../source/include/pin/pintool.ver -fabi-version=2   -o obj-int
el64/inscount0.so obj-intel64/inscount0.o -L../../../intel64/runtime/pincrt -L../../../inte
l64/lib -L../../../intel64/lib-ext -L../../../extras/xed-intel64/lib -lpin -lxed ../../../i
ntel64/runtime/pincrt/crtendS.o -lpindwarf -ldwarf -ldl-dynamic -nostdlib -lc++ -lc++abi -l
m-dynamic -lc-dynamic -lunwind-dynamic
make -C ../../../source/tools/Utils dir obj-intel64/cp-pin.exe
make[1]: Entering directory '/home/kali/pin-3.31/source/tools/Utils'
mkdir -p obj-intel64/
g++ -DTARGET_IA32E -DHOST_IA32E -DFUND_TC_TARGETCPU=FUND_CPU_INTEL64 -DFUND_TC_HOSTCPU=FUND
_CPU_INTEL64 -DTARGET_LINUX -DFUND_TC_TARGETOS=FUND_OS_LINUX -DFUND_TC_HOSTOS=FUND_OS_LINUX
 -I../../../source/tools/Utils -O3  -std=c++11 -o obj-intel64/cp-pin.exe cp-pin.cpp -no-pie

make[1]: Leaving directory '/home/kali/pin-3.31/source/tools/Utils'
../../../pin   -t obj-intel64/inscount0.so -- ../../../source/tools/Utils/obj-intel64/cp-pi
n.exe makefile obj-intel64/inscount0.makefile.copy \
  > obj-intel64/inscount0.out 2>&1
cmp makefile obj-intel64/inscount0.makefile.copy
rm obj-intel64/inscount0.makefile.copy
rm obj-intel64/inscount0.out
```

编写一个简单的C程序FirstCpp.c 执行命令 gcc -o First FirstCpp.c 进行编译，生成了可执行文件 First。

执行命令 .pin -t ./source/tools/ManualExamples/obj-intel64/inscount.so -- ../testCPP/First ，对可执行文件 First 进行插桩，结果如下：

```
┌──(kali㉿kali)-[~/pin-3.31]
└─$ ./pin -t ./source/tools/ManualExamples/obj-intel64/inscount0.so -- ../testCPP/First
hello world
```

同时可以发现在 pin-3.31 路径下新增了一个文件 inscount0.out，内容如下：



（三）使用 PinTool 里的 malloctrace.cpp

查看一下 malloctrace.cpp 的源代码，如下所示，对代码的解释直接放在注释：

```
/*
  * Copyright (C) 2004-2021 Intel Corporation.
  * SPDX-License-Identifier: MIT
  */

 // Pin 插桩框架头文件
#include "pin.H"
// 标准输入输出和文件流头文件
#include <iostream>
#include <fstream>
using std::cerr;
using std::endl;
using std::hex;
using std::ios;
using std::string;

/* ================================================================== */
/* 定义 malloc 和 free 的函数名（跨平台兼容） */
/* ================================================================== */
#if defined(TARGET_MAC)   // macOS 系统下函数名前加下划线
#define MALLOC "_malloc"
#define FREE "_free"
#else                     // Linux/Windows 等系统
#define MALLOC "malloc"
#define FREE "free"
#endif

/* ================================================================== */
```

```
/* 全局变量 */
/* ================================================================== */

std::ofstream TraceFile; // 输出文件流，用于记录跟踪信息

/* ================================================================== */
/* 命令行参数处理 */
/* ================================================================== */

// 定义 "-o" 选项指定输出文件名（默认 malloctrace.out）
KNOB< string > KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o",
"malloctrace.out", "specify trace file name");

/* ================================================================== */
/* 分析函数（记录日志的逻辑） */
/* ================================================================== */

// 在 malloc/free 调用前记录函数名和参数
VOID Arg1Before(CHAR* name, ADDRINT size) {
    TraceFile << name << "(" << size << ")" << endl; // 格式示例：malloc(1024)
}

// 在 malloc 调用后记录返回值（分配的内存地址）
VOID MallocAfter(ADDRINT ret) {
    TraceFile << "    returns " << ret << endl; // 格式示例：    returns 0x7f8a5c000260
}

/* ================================================================== */
/* 插桩函数（在目标函数中插入分析代码） */
/* ================================================================== */

VOID Image(IMG img, VOID* v) {
    // 遍历目标程序加载的所有镜像（可执行文件、共享库），插桩 malloc 和 free

    // 查找当前镜像中的 malloc 函数
    RTN mallocRtn = RTN  FindByName(img, MALLOC);
    if (RTN_Valid(mallocRtn)) { // 如果找到
        RTN  Open(mallocRtn);      // 打开函数对象

        // 在 malloc 调用前插入 Arg1Before，记录参数（分配大小）
        RTN_InsertCall(mallocRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before,
            IARG_ADDRINT, MALLOC,            // 传递函数名 "malloc"
            IARG_FUNCARG_ENTRYPOINT_VALUE, 0, // 获取第一个参数（size_t size）
            IARG_END);

        // 在 malloc 调用后插入 MallocAfter，记录返回值（分配的内存地址）
        RTN_InsertCall(mallocRtn, IPOINT_AFTER, (AFUNPTR)MallocAfter,
            IARG_FUNCRET_EXITPOINT_VALUE, // 获取返回值（void* ptr）
            IARG_END);

        RTN_Close(mallocRtn);
    }

    // 查找当前镜像中的 free 函数
    RTN freeRtn = RTN_FindByName(img, FREE);
    if (RTN_Valid(freeRtn)) {
        RTN_Open(freeRtn);
        // 在 free 调用前插入 Arg1Before，记录参数（释放的地址）
```

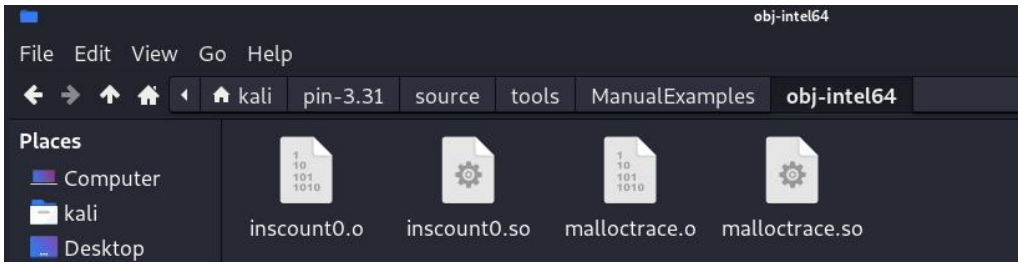
```
            RTN_InsertCall(freeRtn, IPOINT_BEFORE, (AFUNPTR)Arg1Before,
                IARG_ADDRINT, FREE,                    // 传递函数名 "free"
                IARG_FUNCARG_ENTRYPOINT_VALUE, 0, // 获取第一个参数（void* ptr）
                IARG_END);
            RTN_Close(freeRtn);
        }
}

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

// 程序结束时关闭输出文件
VOID Fini(INT32 code, VOID* v) { TraceFile.close(); }

/* ===================================================================== */
/* 帮助信息 */
/* ===================================================================== */

INT32 Usage() {
    cerr << "This tool produces a trace of calls to malloc." << endl;
    cerr << endl << KNOB_BASE::StringKnobSummary() << endl; // 打印支持的命令行选项
    return -1;
}

/* ===================================================================== */
/* 主函数 */
/* ===================================================================== */

int main(int argc, char* argv[]) {
    // 初始化 Pin 和符号管理器（用于解析函数名）
    PIN_InitSymbols();
    if (PIN_Init(argc, argv)) { // 解析命令行参数
        return Usage();
    }

    // 打开输出文件（用户可通过 -o 指定文件名）
    TraceFile.open(KnobOutputFile.Value().c_str());
    TraceFile << hex;                    // 以十六进制格式输出地址
    TraceFile.setf(ios::showbase); // 显示十六进制前缀（如 0x）

    // 注册镜像级插桩函数 Image，当镜像加载时被调用
    IMG_AddInstrumentFunction(Image, 0);
    // 注册程序结束回调 Fini
    PIN_AddFiniFunction(Fini, 0);

    // 启动目标程序，控制权交给 Pin
    PIN_StartProgram();

    return 0; // 实际不会执行到这里
}

/* ===================================================================== */
/* eof */
/* ===================================================================== */
```

主要功能：跟踪目标，记录程序中所有 malloc 和 free 的调用信息，包括：①malloc 的申请大小和返回地址；②free 的释放地址。

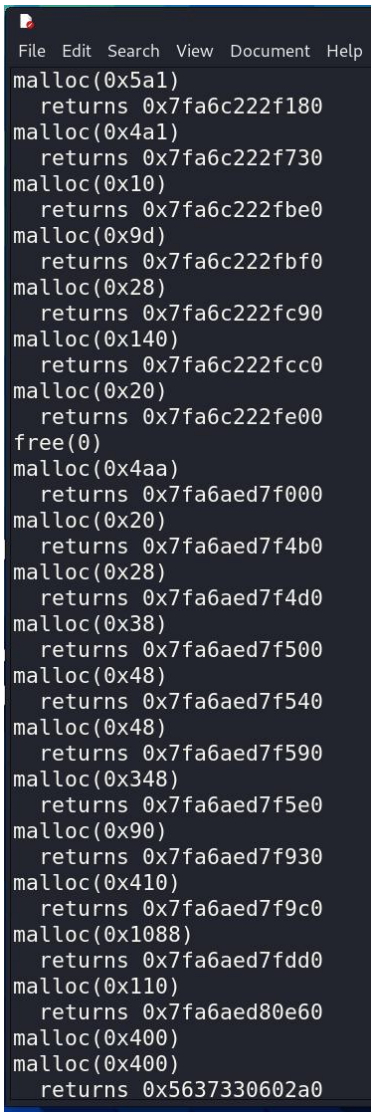
实现原理：①插桩位置：在 malloc 和 free 函数的入口和出口插入分析函数。②参数获

取：通过 IARG_FUNCARG_ENTRYPOINT_VALUE 捕获函数参数，通过 IARG_FUNCRET_EXITPOINT_VALUE 捕获返回值。

现在想要编译 malloctrace.cpp，执行指令 `make malloctrace.test TARGET=intel64`，成功产生了动态链接库 malloctrace.so。



执行命令 `.pin -t ./source/tools/ManualExamples/obj-intel64/malloctrace.so -- ../testCPP/First`，对可执行文件 First 进行插桩，得到 pin-3.31 路径下的一个新增文件 malloctrace.out，内容如下：



关键字段解析

1. malloc(十六进制数值)
表示程序调用 malloc 申请内存，括号内为申请的内存大小（十六进制）。
例如：malloc(0x5a1)表示申请 1441 字节（0x5a1 = 1441）。

2. returns 十六进制地址
表示 malloc 返回的内存起始地址。
例如：returns 0x7fa6c222f180 表示分配的内存块起始地址为 0x7fa6c222f180。

3. free(十六进制地址)
表示程序调用 free 释放内存，括号内为待释放的内存地址。
例如：free(0) 表示释放空指针（合法操作，实际无效果）。

malloctrace 工具可用于内存泄漏检测：对比 malloc 和 free 的记录，查找未释放的地址。例如：malloc(0x5a1) → 0x7fa6c222f180 未释放 → 可能泄漏 1441 字节。

四、心得体会

通过本次实验，我理解了动态二进制插桩技术的核心原理与实际应用，并掌握了 Pin 工具的基本使用方法。以下是我的主要收获与思考：

1.动态插桩的灵活性：Pin 框架通过运行时修改目标程序二进制代码，无需重新编译即可插入分析逻辑。这种非侵入式方法为性能分析、漏洞检测等场景提供了极大便利。

2.内存行为分析的实践意义：使用 malloctrace 工具跟踪 malloc 和 free，直观展示了程序的内存管理行为。通过日志分析，可快速定位未释放的内存块（如 malloc(0x5a1)未匹配 free），为内存泄漏排查提供直接依据。

总结使用 Pin 插桩工具的主要步骤如下：

1.安装 Pin 框架

下载适用于 Linux 的 Pin 工具包，解压到 Kali 虚拟机中。

主要目录结构：

/source/tools/ManualExamples：包含示例 Pintool 源码（如 inscount0.cpp、malloctrace.cpp）。

/obj-intel64：编译后生成的动态链接库（.so 文件）存放路径。

2. 编写或选择 Pintool

使用示例工具：直接使用 ManualExamples 中的示例代码（如 inscount0.cpp 统计指令数，malloctrace.cpp 跟踪内存操作）。

自定义工具：按需修改源码，通过 Pin API 插入分析逻辑（如捕获函数参数、返回值）。

3. 编译 Pintool

进入示例目录，执行编译命令生成 .so 文件：

```
make <工具名>.test TARGET=intel64
```

生成文件：obj-intel64/<工具名>.so（如 inscount0.so、malloctrace.so）。

4. 运行 Pintool 分析目标程序

使用 pin 命令加载 Pintool 并启动目标程序：

```
./pin -t obj-intel64/<工具名>.so -- <目标程序路径> [参数]
```

5. 分析输出结果

工具默认生成输出文件（如 inscount0.out、malloctrace.out），内容为统计或跟踪信息。