



软件安全

徐剑

信息安全系

xuj@mail.neu.edu.com

Software Collge of NEU

第6章 软件漏洞挖掘之实践

6.1 AFL模糊测试框架

6.2 程序插桩技术

6.3 消息Hook技术

6.4 API Hook技术

6.5 Z3约束求解器

6.6 Angr应用示例

6.1 AFL模糊测试框架

AFL模糊测试框架

1

AFL是一款**基于覆盖引导（Coverage-guided）**的模糊测试工具，它通过记录输入样本的代码覆盖率，从而调整输入样本以提高覆盖率，增加发现漏洞的概率。

2

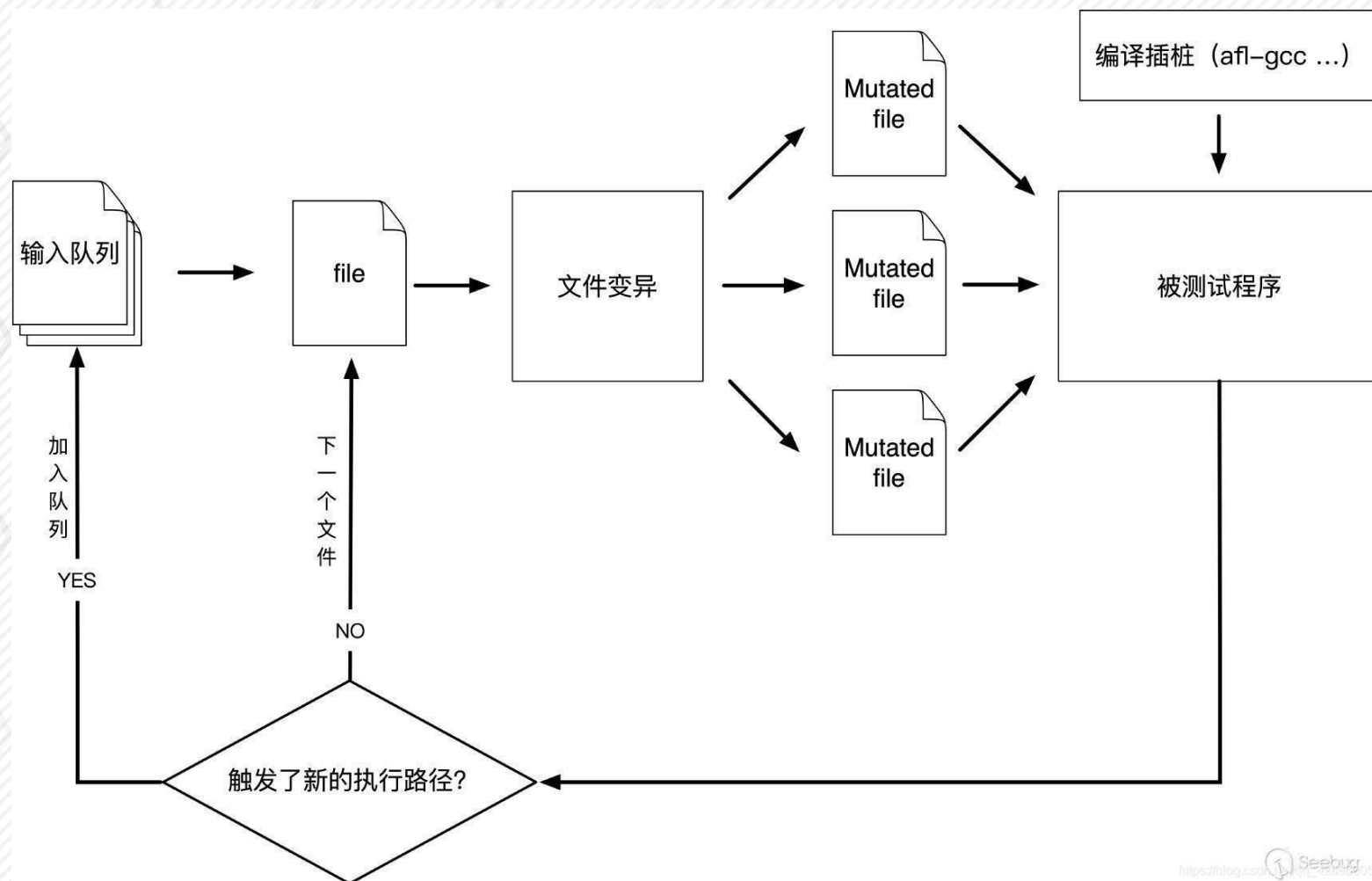
AFL主要用于**C/C++程序的测试**，被测程序**有无程序源码均可**，有源码时可以对源码进行编译时插桩，无源码可以借助QEMU的User-Mode模式进行二进制插装。

3

支持多平台（ARM、X86、X64）、多系统（Linux、BSD、Windows、MacOS），性能高。

AFL工作流程

- 从源码编译程序时进行**插桩**，**以记录代码覆盖率**；
- 选择一些输入文件作为初始测试集加入输入队列；
- 将队列中的文件按策略进行“**突变**”；
- 如果经过变异文件更新了覆盖范围，则保留在队列中；
- 循环进行，期间**触发了crash的文件**会被记录下来。



AFL安装

- ❑ 在Kali 2021下，利用sudo apt-get install afl即可安装。
- ❑ 查看路径可以看到afl安装的文件：ls /usr/bin/afl*

```
(kali㉿kali)-[~/demo]
└─$ ls /usr/bin/afl*
/usr/bin/afl-analyze      /usr/bin/afl-cmin.bash  /usr/bin/afl-showmap
/usr/bin/afl-clang        /usr/bin/afl-fuzz       /usr/bin/afl-system-config
/usr/bin/afl-clang++      /usr/bin/afl-g++        /usr/bin/afl-tmin
/usr/bin/afl-clang-fast   /usr/bin/afl-gcc        /usr/bin/afl-whatsup
/usr/bin/afl-clang-fast++ /usr/bin/afl-gotcpu
/usr/bin/afl-cmin         /usr/bin/afl-plot
```

- afl-gcc和afl-g++分别对应的是gcc和g++的封装。
- afl-fuzz是AFL的主体，用于对目标程序进行fuzz。
- afl-analyze可以对用例进行分析，看能否发现用例中有意义的字段。
- afl-tmin和afl-cmin对用例进行简化。
- afl-showmap用于对单个用例进行执行路径跟踪。


```

if(ptr[0] == 'd') {
    if(ptr[1] == 'e') {
        if(ptr[2] == 'a') {
            if(ptr[3] == 'd') {
                if(ptr[4] == 'b') {
                    if(ptr[5] == 'e') {
                        if(ptr[6] == 'e') {
                            if(ptr[7] == 'f') {
                                abort();
                            }
                            else printf("%c",ptr[7]);
                        }
                        else printf("%c",ptr[6]);
                    }
                    else printf("%c",ptr[5]);
                }
                else printf("%c",ptr[4]);
            }
            else printf("%c",ptr[3]);
        }
        else printf("%c",ptr[2]);
    }
    else printf("%c",ptr[1]);
}
else printf("%c",ptr[0]);

```

AFL模糊测试

以一个白盒模糊测试为例。

(1) 创建本次实验的程序：新建文件夹demo，并创建实验的程序Test.c，该代码编译后得到的程序如果被传入“deadbeef”则会终止，如果传入其他字符会原样输出。

使用afl的编译器编译，可以使模糊测试过程更加高效。

命令：afl-gcc -o test test.c

AFL模糊测试

编译后会有插桩符号，使用下面的命令可以验证这一点。

命令：readelf -s ./test | grep afl

```
(kali㉿kali)-[~/demo]
$ readelf -s ./test | grep afl
35: 00000000000001628      0 NOTYPE  LOCAL  DEFAULT 14  __afl_maybe_log
37: 000000000000040b0      8 OBJECT  LOCAL  DEFAULT 25  __afl_area_ptr
38: 00000000000001660      0 NOTYPE  LOCAL  DEFAULT 14  __afl_setup
39: 00000000000001638      0 NOTYPE  LOCAL  DEFAULT 14  __afl_store
40: 000000000000040b8      8 OBJECT  LOCAL  DEFAULT 25  __afl_prev_loc
41: 00000000000001655      0 NOTYPE  LOCAL  DEFAULT 14  __afl_return
42: 000000000000040c8      1 OBJECT  LOCAL  DEFAULT 25  __afl_setup_failure
43: 00000000000001681      0 NOTYPE  LOCAL  DEFAULT 14  __afl_setup_first
45: 00000000000001949      0 NOTYPE  LOCAL  DEFAULT 14  __afl_setup_abort
46: 0000000000000179e      0 NOTYPE  LOCAL  DEFAULT 14  __afl_forkserver
47: 000000000000040c4      4 OBJECT  LOCAL  DEFAULT 25  __afl_temp
48: 0000000000000185c      0 NOTYPE  LOCAL  DEFAULT 14  __afl_fork_resume
49: 000000000000017c4      0 NOTYPE  LOCAL  DEFAULT 14  __afl_fork_wait_loop
50: 00000000000001941      0 NOTYPE  LOCAL  DEFAULT 14  __afl_die
51: 000000000000040c0      4 OBJECT  LOCAL  DEFAULT 25  __afl_fork_pid
98: 000000000000040d0      8 OBJECT  GLOBAL  DEFAULT 25  __afl_global_area_ptr
```


AFL模糊测试

(2) 创建测试用例

首先，创建两个文件夹in和out，分别存储模糊测试所需的输入和输出相关的内容。

命令：**mkdir in out**

然后，在输入文件夹中创建一个包含字符串“hello”的文件。

命令：**echo hello> in/foo**

foo就是我们的测试用例，里面包含初步字符串hello。AFL会通过这个语料进行变异，构造更多的测试用例。

(3) 启动模糊测试

运行如下命令，开始启动模糊测试（@@表示目标程序需要从文件读取输入）：

命令：**afl-fuzz -i in -o out -- ./test @@**

AFL模糊测试

(4) 分析crash

观察fuzzing结果，如有crash，定位问题。

```
american fuzzy lop ++2.68c (test) [explore] {0}
- process timing -
  run time : 0 days, 0 hrs, 8 min, 39 sec
  last new path : 0 days, 0 hrs, 5 min, 10 sec
  last uniq crash : 0 days, 0 hrs, 5 min, 7 sec
  last uniq hang : none seen yet
- cycle progress -
  now processing : 2.66 (25.0%)
  paths timed out : 0 (0.00%)
- stage progress -
  now trying : MOpt-core-havoc
  stage execs : 138/256 (53.91%)
  total execs : 1.35M
  exec speed : 2321/sec
- fuzzing strategy yields -
  bit flips : 2/352, 1/344, 0/328
  byte flips : 0/44, 0/36, 0/20
  arithmetics : 1/2464, 0/70, 0/0
  known ints : 0/241, 1/970, 0/879
  dictionary : 0/0, 0/0, 0/0
  havoc/splice : 3/501k, 0/484k
  py/custom : 0/0, 0/0
  trim : 6.67%/4, 0.00%
- overall results -
  cycles done : 65
  total paths : 8
  uniq crashes : 1
  uniq hangs : 0
- map coverage -
  map density : 0.01% / 0.03%
  count coverage : 1.00 bits/tuple
- findings in depth -
  favored paths : 8 (100.00%)
  new edges on : 8 (100.00%)
  total crashes : 2 (1 unique)
  total tmoouts : 0 (0 unique)
- path geometry -
  levels : 6
  pending : 0
  pend fav : 0
  own finds : 7
  imported : n/a
  stability : 100.00%
[cpu000: 50%]
```

- 在out文件夹的crashes子文件夹里面是产生crash的样例，hangs里面是产生超时的样例。
- 通常，得到crash样例后，可以将这些样例作为目标测试程序的输入，重新触发异常并跟踪运行状态，进行分析、定位程序出错的原因或确认存在的漏洞类型。

6.2 程序插桩技术

1. 插桩概念

程序插桩，是借助往被测程序中插入操作，来实现测试目的的方法。简单的说，**插桩就是在代码中插入一段我们自定义的代码**，它的目的在于通过我们插入程序中的自定义的代码，得到期望得到的信息，比如程序的控制流和数据流信息，以此来实现测试或者其他目的。

- 最简单的插桩是在程序中**插入输出语句**，以监测变量的取值或者状态是否符合预期。这种插桩手段在服务类应用程序、基于日志的程序调错等。
- **断言**是一种特殊的插桩，是在程序的特定部位插入语句来检查变量的特性。



2. 插桩分类

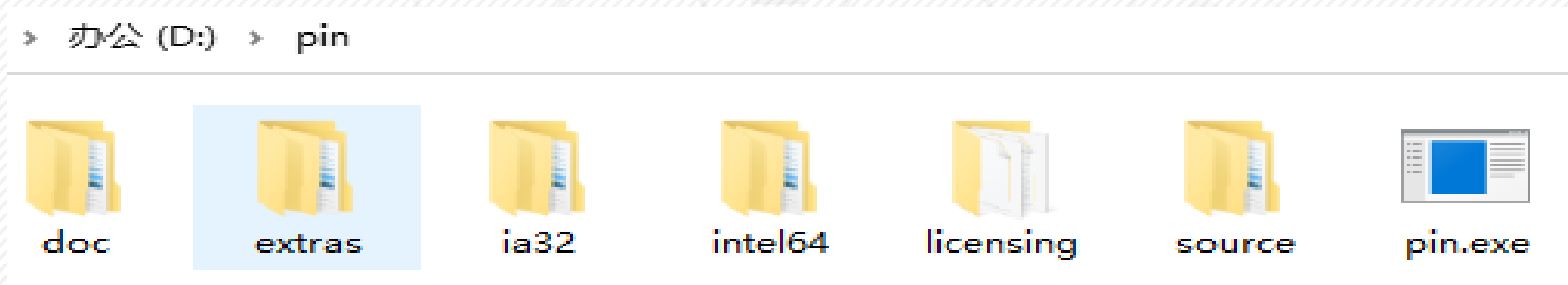
- **源代码插桩**是指在被测程序运行之前，通过自动化工具或者程序员手动在需要收集信息的地方插入探针，之后重新编译运行被测程序。
- **静态二进制插桩**和源代码插桩类似，都是在程序运行之前插入探针，与源代码插桩不同是静态二进制插桩直接对程序编译之后的二进制机器码进行插桩。编写难度更大、可移植性更差。
- **动态二进制插桩**在程序运行时，直接接管被测程序并且截获其二进制指令并插入探针。插桩程序难度更大，程序运行开销也越大。

3. Pin插桩示例

- ❑ 动态二进制插桩技术被广泛的用在各个领域。为了解决动态二进制插桩程序编写难度大、抽象层次低的缺点，提高代码的重用性，人们开发了许多**动态二进制插桩框架**。
- ❑ **Pin是Intel公司开发的动态二进制插桩框架，支持IA-32和x86-64指令集架构，支持windows和linux。**
- ❑ Pin可以监控程序的每一步执行，提供了丰富的API，可以在二进制程序运行过程中插入各种函数，比如说我们要统计一个程序执行了多少条指令，每条指令的地址等信息。

安装及使用Pin

解压下载的Windows版本的Pin压缩包，整体文件夹结构如下所示。



文件夹ia32和intel64包含了英特尔不同体系架构下的相关库和可执行文件，文件夹doc包含了Pin相关的用户手册、API文档等，而文件夹source\tools里包含了大量的PinTool。

Pintool。Pin通过已经定义的tools或者自己开发的tool来完成对目标程序的插桩。通常，PinTool以动态链接库方式使用，即Linux下是.so文件，而Windows下是.dll文件。

Pin用法

```
pin [OPTION] [-t <tool> [<toolargs>]] -- <command line>
```

注：<command line>: <App EXE> [App args]

举例，在Linux下使用如下命令来进行动态插桩，并得到输出信息文件：

```
$ ./pin -t ./source/tools/.../obj-intel64/xxxx.so -- TargetApp args
```

这里的**xxxx.so**指代所要使用的Pintool，如**inscount0.so**，“--”之后要输入需要运行的目标程序（**TargetApp**）和其相关参数（**args**）。默认输出结果将保存到**xxxx.out**，也可以使用在Pintool中实现函数**KnobOutputFile**后通过**toolargs: -o filepath**指定。

使用Pintool

在Pin的安装文件里，在source\tools里已经定义了大量PinTool，可以编译后直接使用，也可以自己开发自己的定制的PinTool来完成特定的插桩任务。

(1) Linux下编译现有Pintool

Linux PinTool编译在Linux下，可以使用通过以下命令可以对所有Pintool进行编译：

```
$ cd source/tools/ManualExamples
```

```
$ make all TARGET=intel64
```

也可以指定某个具体的Pintool工具，如inscount0：

```
$ cd source/tools/ManualExamples
```

```
$ make inscount0.test TARGET=intel64
```

使用Pintool

在pin\source\tools\ManualExamples里，已经定了好多PinTool，这些常用的Pintool功能介绍如下表所示：

Pintool	功能说明
inscount	统计执行的指令数量
itrace	记录执行指令的eip
malloctrace	记录malloc和free的调用情况
pinatrace	记录读取内存的位置和值
proccount	统计Procedure的信息，包括名称、镜像、地址、指令数
w_malloctace	记录PtlAllocateHeap的调用情况

使用Pintool

(2) Inscount插桩示例

首先，进入source/tools/ManualExamples，对inscount0.cpp进行编译来产生其对应的动态链接库，所使用的命令为：make inscount0.test TARGET=intel64。

```
└─$ make inscount0.test TARGET=intel64
mkdir -p obj-intel64/
g++ -Wall -Werror -Wno-unknown-pragmas -D__PIN__=1 -DPIN_CRT=1 -fno-stack-protector -fno-exceptions -funwind-tables -fasynchronous-unwind-tables -fno-rtti -DTARGET_IA32E -DHOST_IA32E -fPIC -DTARGET_LINUX -fabi-version=2 -faligned-new -I../..../source/include/pin -I../..../source/include/pin/gen -isystem /home/kali/Downloads/pin-3.18/extras/stlport/include -isystem /home/kali/Downloads/pin-3.18/extras/libstdc++/include -isystem /home/kali/Downloads/pin-3.18/extras/crt/include -isystem /home/kali/Downloads/pin-3.18/extras/crt/include/arch-x86_64 -isystem /home/kali/Downloads/pin-3.18/extras/crt/include/kernel/uapi -isystem /home/kali/Downloads/pin-3.18/extras/crt/include/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 -c -o obj-intel64/inscount0.o inscount0.cpp
```

使用Pintool

编写一个简单的控制台命令程序FirstC.c，并进行测试。

```
#include <stdio.h>
void main(){
    printf("hello world!");
}
```

在Linux下编译c文件的命令为: **gcc -o First FirstC.c**。

然后，对First可执行程序进行程序插桩的Pin命令为：

`./pin -t ./source/tools/ManualExamples/obj-intel64/inscount0.so -- ../testCPP/First`

```
(kali@kali)-[~/Downloads/pin-3.18]
$ ./pin -t ./source/tools/ManualExamples/obj-intel64/inscount0.so -- ../tes
tCPP/First
hello world!
```

在pin-3.18路径下增加了一个输出文件inscount.out，文件内容如下：“Count 192994”，即对指令数进行了插桩

Pintool基本用法

(1) 插桩框架：打开inscout0.cpp

```
ofstream OutFile;
static UINT64 icount = 0; // 静态变量，保存运行的指令数的计数
VOID docount() { icount++; } //这个函数在每条指令执行以前被调用

VOID Instruction(INS ins, VOID *v) //Pin工具每次遇到一个新指令都会调用该函数
{
    //在每个指令之前插入一个函数docount的调用，没有任何参数
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

//指定输出文件为inscount.out
KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool", "o", "inscount.out", "specify output
file name");

//当应用退出的时候调用本函数
VOID Fini(INT32 code, VOID *v)
{
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}
```

Pintool基本用法

(1) 插桩框架：打开inscout0.cpp

```
int main(int argc, char * argv[])
{
    //初始化pin
    if (PIN_Init(argc, argv)) return Usage();

    OutFile.open(KnobOutputFile.Value().c_str());
    //注册了一个名为Instruction的回调函数，该函数
    INS_AddInstrumentFunction(Instruction, 0);

    //当应用退出的时候，注册函数Fini来进行处理
    PIN_AddFiniFunction(Fini, 0);

    //启动程序
    PIN_StartProgram();
    return 0;
}
```

调用函数PIN_Init完成初始化

通过使用INS_AddInstrumentFunction注册一个插桩函数，在原始程序的每条指令被执行前，都会进入Instruction这个函数中

注册退出回调函数，退出时调用该函数

使用函数PIN_StartProgram启动程序

Pintool基本用法

(2) 插桩模式：

插桩粒度	API	执行时机
指令级插桩 (instruction)	INS_AddInstrumentFunction	执行一条新指令
轨迹级插桩 (trace)	TRACE_AddInstrumentFunction	执行一个新trace
镜像级插桩 (image)	IMG_AddInstrumentFunction	加载新镜像时
函数级插桩 (routine)	RTN_AddInstrumentFunction	执行一个新函数时

在各种粒度的插装函数调用时，可以在代码中添加自己的处理函数，程序被加载后，在被插装的代码运行时，自己添加的函数会被调用。

Pintool基本用法

(3) 指令级插桩

docount的作用即是将一个全局变量加1

指令级插桩的对象就是所有指令。很明显，inscount0.cpp这个Pintool是指令级插桩，通过调用INS_AddInstrumentFunction注册了一个回调函数Instruction。

```
VOID Instruction(INS ins, VOID *v){  
    //在每个指令之前插入一个函数docount的调用，没有任何参数  
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);  
}
```

```
VOID LEVEL_PINCLIENT::INS_InsertCall( INS ins, IPOINT action, AFUNPTR funptr, ...)  
// 插入相对于指令ins的funptr调用  
参数值:    ins 被插桩的指令  
            action 指定插桩位置，比如之前(IPOINT_BEFORE)、之后(IPOINT_AFTER)等  
            funptr 插入一个funptr的调用  
            ... funptr的参数列表,以IARG_END结尾，查看IARG_TYPE了解细节
```

(3) 指令级插桩

将回调函数Instruction修改如下：

```
VOID Instruction(INS ins, VOID *v)
{
    if (INS_Opcode(ins) == XED_ICLASS_MOV &&
        INS_IsMemoryRead(ins) &&
        INS_OperandIsReg(ins, 0) &&
        INS_OperandIsMemory(ins, 1))
    {
        icount++;
    }
}
```

在现在的函数中，设定了复杂的指令插桩条件，只有当下述条件满足的时候才会计数：
命令是mov指令、是一条内存读指令、指令的第一个操作数是寄存器、指令的第二个操作数是内存。实际上，通过组合这些API就可以非常精确地筛选出想要插桩的指令了。

6.3 消息Hook

1. Hook概念

Hook（钩子）

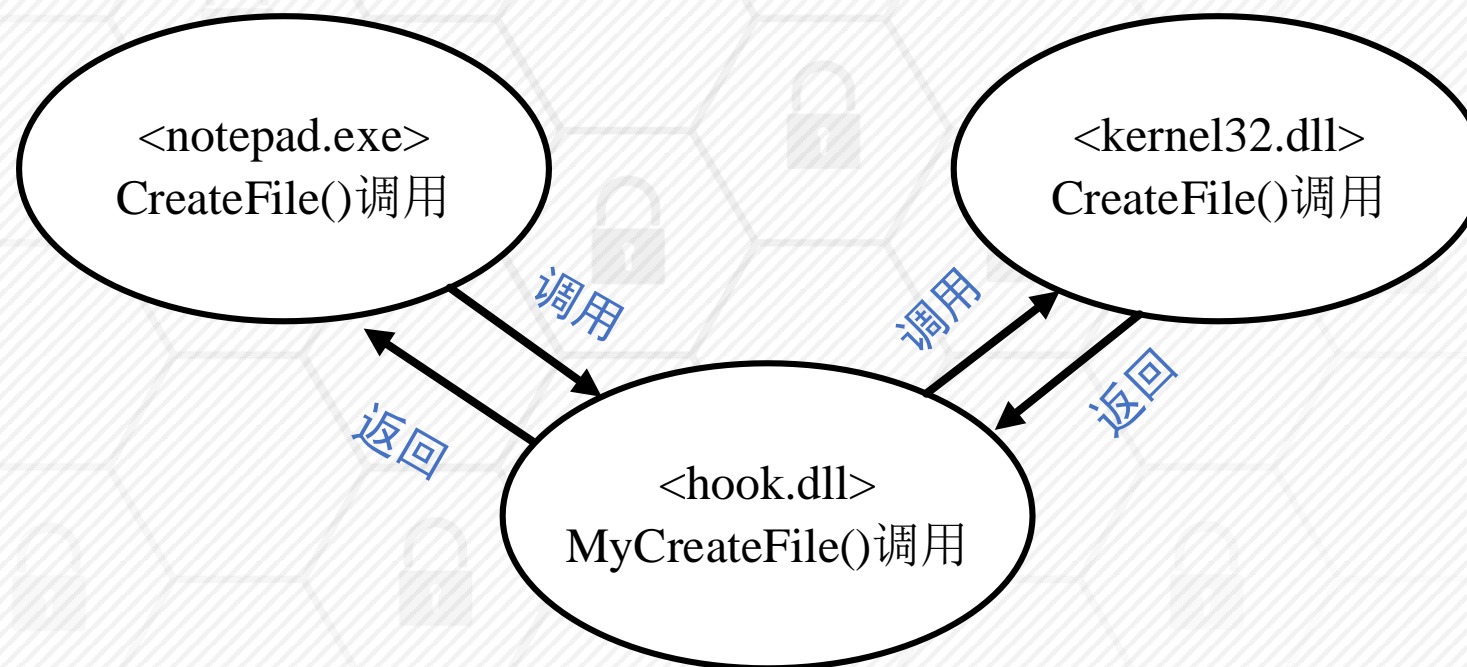
Hook（钩子），是一种过滤（或叫挂钩）消息的技术。

Hook的目的是过滤一些关键函数调用，在函数执行前，先执行自己的挂钩函数，达到监控函数调用，改变函数功能的目的。

Hook技术已经被广泛应用于安全的多个领域，比如杀毒软件的主动防御功能，涉及到对一些敏感API的监控，就需要对这些API进行Hook；窃取密码的木马病毒，为了接收键盘的输入，需要Hook键盘消息；甚至是Windows系统及一些应用程序，在打补丁时也需要用到Hook技术。当然，Hook技术也可以用在软件分析和漏洞挖掘等领域。

Hook技术按照实现原理来分的话可以分为两种：

- API HOOK：拦截Windows API;
- 消息HOOK：拦截Windows消息。



Hook方法很多，主要包括调试法和注入法

2. 消息Hook

Windows系统建立在事件驱动机制上，整个系统通过消息传递实现的。在Windows系统里，消息Hook就是一个Windows消息的拦截机制，可以拦截单个进程的消息（线程钩子），也可以拦截所有进程的消息（系统钩子），也可以对拦截的消息进行自定义的处理：

- **如果对于同一事件（如鼠标消息）既安装了线程钩子又安装了系统钩子，那么系统会自动先调用线程钩子，然后调用系统钩子。**
- **对同一事件消息可安装多个钩子处理过程，这些钩子处理过程形成了钩子链。后加入的有优先控制权。**

Windows提供了一个官方函数SetWindowsHookEx用于设置消息Hook，编程时只要调用该API就能简单地实现Hook，其定义如下：

```
HHOOK SetWindowsHookEx(  
    int_idHook,          //hook类型  
    HOOKPROC lpfn,       //hook函数  
    HINSTANCE hMod,      //hook函数所属DLL的Handle  
    DWORD dwThreadId     //设定要Hook的线程ID，0表示“全局钩  
子” (Global Hook) 监视所有进程  
);
```

基于消息Hook的DLL注入

DLL注入技术是向一个正在运行的进程插入自有DLL的过程。DLL注入的目的是将代码放进另一个进程的地址空间中，现在被广泛应用于软件分析、软件破解、恶意代码等领域，注入方法也很多，比如利用注册表注入、CreateRemoteThread远程线程调用注入等。

在Windows中，利用SetWindowsHookEx函数创建钩子（Hooks）可以实现DLL注入。**设计实验如下：**

- 编制键盘消息的Hook函数—KeyHook.dll中的KeyboardProc函数
- 通过SetWindowsHookEx创建键盘消息钩子实现DLL注入（执行DLL内部代码）

第一步：编写DLL文件

新建一个VC 6的动态链接库工程，命名为KeyHook，添加一个代码文件KeyHook.cpp：

```
#include "stdio.h"
#include "windows.h"
#define DEF_PROCESS_NAME
HINSTANCE g_hInstance = NULL;
HHOOK g_hHook = NULL;
HWND g_hWnd = NULL;

BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD dwReason, LPVOID lpvReserved){
    switch( dwReason )
    {
        case DLL_PROCESS_ATTACH:
            g_hInstance = hinstDLL;
            break;

        case DLL_PROCESS_DETACH:
            break;
    }
    return TRUE;
}
```

相当于初始化程序

//Hook函数（键盘消息处理函数）

LRESULT CALLBACK KeyboardProc(int nCode, WPARAM wParam, LPARAM lParam)

{

char szPath[MAX_PATH] = {0};

char *p = NULL;

对记事本消息拦截，其他不做动作
Return 1处可以编码做任意动作

if(nCode >= 0){

// lParam 第31位bit: 0 => key press, 1 => key release

if(!(lParam & 0x80000000)) { //当按键被释放时

GetModuleFileNameA(NULL, szPath, MAX_PATH);

p = strrchr(szPath, '\\');

//比较当前进程名称，若为notepad.exe，则消息不会继续传递

if(!_stricmp(p + 1, DEF_PROCESS_NAME))

return 1; //丢弃该Keyboard消息

}

}

//若不为notepad.exe，调用CallNextHookEx()函数将消息传递给下一个“钩子”或应用程序

return CallNextHookEx(g_hHook, nCode, wParam, lParam);

}


```
#ifdef __cplusplus
extern "C" {
#endif

    __declspec(dllexport) void HookStart()
    {
        g_hHook = SetWindowsHookEx(WH_KEYBOARD, KeyboardProc, g_hInstance, 0);
    }

    __declspec(dllexport) void HookStop()
    {
        if( g_hHook )
        {
            UnhookWindowsHookEx(g_hHook);
            g_hHook = NULL;
        }
    }

}

#ifdef __cplusplus
}
#endif
```

定义两个导出函数

第二步：编写DLL注入功能的可执行文件

新建一个VC6的控制台程序，添加源文件HookMain.cpp如下：

```
#include "stdio.h"
#include "conio.h"
#include "windows.h"

#define DEF_DLL_NAME        "KeyHook.dll"
#define DEF_HOOKSTART      "HookStart"
#define DEF_HOOKSTOP       "HookStop"

typedef void (*PFN_HOOKSTART)();
typedef void (*PFN_HOOKSTOP)();
void main()
{
    HMODULE          hDll = NULL;
    PFN_HOOKSTART    HookStart = NULL;
    PFN_HOOKSTOP     HookStop = NULL;
    char             ch = 0;
```

```
hDll = LoadLibraryA(DEF_DLL_NAME); // 加载KeyHook.dll
```

```
if( hDll == NULL ){  
    printf("LoadLibrary(%s) failed!!! [%d]", DEF_DLL_NAME, GetLastError());  
    return;  
}
```

```
// 获取导出函数地址
```

```
HookStart = (PFN_HOOKSTART)GetProcAddress(hDll, DEF_HOOKSTART);
```

```
HookStop = (PFN_HOOKSTOP)GetProcAddress(hDll, DEF_HOOKSTOP);
```

```
HookStart(); // 开始Hook
```

```
// 等待直到用户输入'q'
```

```
printf("press 'q' to quit!\n");
```

```
while( _getch() != 'q' )    ;
```

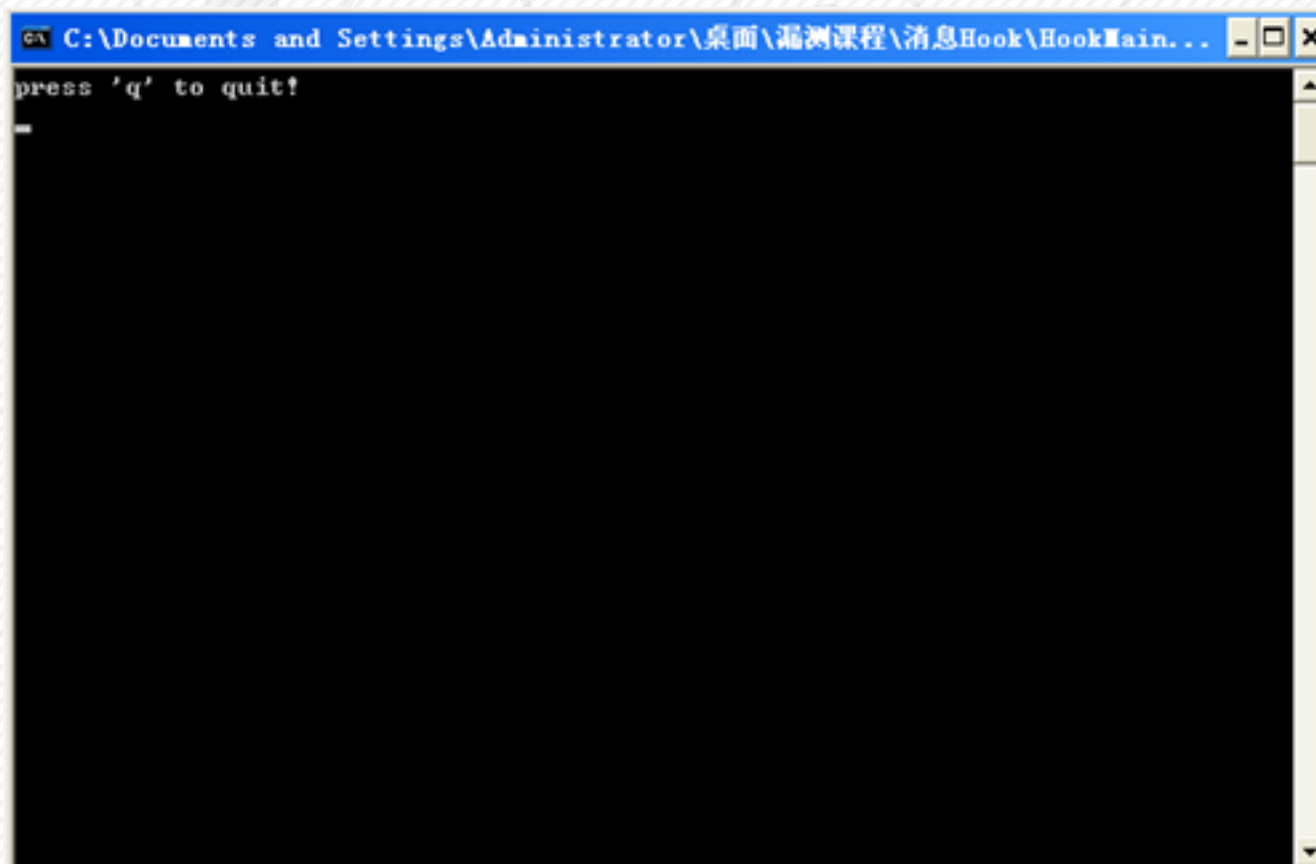
```
HookStop(); // 结束Hook
```

```
FreeLibrary(hDll); // 卸载KeyHook.dll
```

```
}
```

第三步：实验验证

将HookMain.exe和KeyHook.dll放在相同目录下，运行HookMain.exe安装键盘消息Hook后，将实现notepad.exe进程的键盘消息拦截，使之无法显示在记事本中。直到输入“q”才可停止键盘Hook。



6.4 API Hook

1. API Hook概念

API Hook

API HOOK技术是对API函数进行Hook（挂钩）的技术。API HOOK的基本方法就是通过hook“接触”到需要修改的API函数入口点，改变它的地址指向新的自定义的函数。

API Hook方法多种：IAT Hook、代码Hook、EAT Hook

EAT: export address table, 导出地址表

IAT Hook：将输入函数地址表IAT内部的API地址更改为Hook函数地址。

它的优点是实现起来较简单，缺点是无法钩取不在IAT而在程序中使用的API（如：动态加载并使用DLL时）。

代码Hook：系统库 (*.dll) 映射到进程内存时，从中查找API的实际地址，并直接修改代码。

该方法应用范围广泛，具体实现中常通过以下方式：

- 使用JMP指令修改起始代码；
- 覆写函数局部；
- 仅修改必需部分的局部。

修改起始代码示例

在动态链接库被动态加载到进程的地址空间中后，将要使用的API函数的所在位置的前几个字节修改为一条跳转指令，跳转到代理函数去执行，在需要调用原API函数时，再将源代码复制过去或者跳转回去。例如：设自定义函数My_Send的地址为0x0157143F，为了使对Send函数调用转到这里执行，可以嵌入如下汇编代码：

```
mov eax, 0157143F; //将自定义函数地址放入寄存器eax, 对应机器码B83F145701  
jmp eax;          //跳转到eax处对应机器码： FFE0
```

CPU仅能识别机器码，所以要将汇编代码对应的最原始的机器码写入到目标API所在内存。上面两行汇编代码对应的机器码为：B83F145701FFE0，一共7个字节。其中第2-5个字节的取值会随自定义函数的地址不同而不同。

2. IAT Hook示例

实验三：利用API Hook技术对敏感函数lstrcpy函数进行Hook，获取函数的输入参数，进行记录分析。

步骤：

- [1] 编写自定义函数：实现检测等需要的功能；
- [2] Hook实现：根据PE文件结构寻找IAT，并将IAT中的目标函数的地址更换为自定义的函数地址；
- [3] Dll注入：将包含IAT Hook代码及自定义的Hook函数的Dll注入到目标文件中。

IAT HOOK函数

编写一个动态链接库文件，其中编写自己的Hook函数及其逻辑。

```
BOOL hook_iat(LPCSTR szDllName, PROC pfnOrg, PROC pfnNew)
{
// szDllName指目标API所在系统Dll名称，即"kernel32.dll"
// pfnOrg指原始API地址，即lstrcpyW()的地址
// pfnNew指用于替换lstrcpyW()的自定义函数的地址，即MylstrcpyW()的地址

// hMod, pAddr = 可执行文件的ImageBase
//           = VA of MZ signature (IMAGE_DOS_HEADER)
    hMod = GetModuleHandle(NULL);
    pAddr = (PBYTE)hMod;

// pAddr = VA of PE signature (IMAGE_NT_HEADERS)
    pAddr += *((DWORD*)&pAddr[0x3C]);

// dwRVA = RVA of IMAGE_IMPORT_DESCRIPTOR Table
    dwRVA = *((DWORD*)&pAddr[0x80]);

// pImportDesc = VA of IMAGE_IMPORT_DESCRIPTOR Table
    pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)((DWORD)hMod+dwRVA);
```


更改内存可读写属性, 更改IAT值

```
for( ; pImportDesc->Name; pImportDesc++)  
{  
    szLibName = (LPCSTR)((DWORD)hMod + pImportDesc->Name);  
    if( !_stricmp(szLibName, szDllName) )  
    {  
        // pThunk = IMAGE_IMPORT_DESCRIPTOR.FirstThunk  
        pThunk = (PIMAGE_THUNK_DATA)((DWORD)hMod + pImportDesc->FirstThunk);  
  
        // pThunk->u1.Function = VA of API  
        for( ; pThunk->u1.Function; pThunk++){  
            if( pThunk->u1.Function == (DWORD*)pfnOrg ){  
                // 更改内存属性为E/R/W  
                VirtualProtect((LPVOID)&pThunk->u1.Function, 4, PAGE_EXECUTE_READWRITE,  
&dwOldProtect);  
  
                // 修改IAT值（钩取）  
                pThunk->u1.Function = (DWORD*)pfnNew;  
  
                // 恢复内存属性  
                VirtualProtect((LPVOID)&pThunk->u1.Function, 4, dwOldProtect, &dwOldProtect);  
            }  
        }  
        return TRUE;  
    }  
}
```

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
{
    switch( fdwReason )
    {
        case DLL_PROCESS_ATTACH :
            // 保存原始API地址
            g_pOrgFunc = GetProcAddress(GetModuleHandle((LPCTSTR)"kernel32.dll"), "lstrcpyW");

            // # hook
            // 用hookiat!MySetWindowText()钩取user32!SetWindowTextW()
            hook_iat("kernel32.dll", g_pOrgFunc, (PROC)MylstrcpyW);
            break;

            case DLL_PROCESS_DETACH :
                // # unhook
                // 将..exe的IAT恢复原值
                hook_iat("kernel32.dll", (PROC)MylstrcpyW, g_pOrgFunc);
                break;
    }

    return TRUE;
}
```

注入DLL文件

新建Windows控制台程序实现DLL文件注入。 - **USAGE** : InjectDll.exe <i|e> <PID> <dll_path>。 调用InjectDll完成注入。

```
BOOL InjectDll (DWORD dwPID, LPCTSTR szDllName)
// dwPID - 待注入目标进程的PID值
// szDllName – 待注入Dll的path
{
    HANDLE hProcess, hThread;
    LPVOID pRemoteBuf;
    DWORD dwBufSize = (DWORD)(_tcslen(szDllName) + 1) * sizeof(TCHAR);
    LPTHREAD_START_ROUTINE pThreadProc;

    // #1.使用dwPID获取目标进程(notepad.exe)句柄
    if ( !(hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwPID)) )
    {
        DWORD dwErr = GetLastError();
        return FALSE;
    }
}
```

```
if(hProcess!=NULL)
```

```
// #2.在目标进程(notepad.exe)中分配szDllName大小的内存
```

```
pRemoteBuf = VirtualAllocEx(hProcess, NULL, dwBufSize, MEM_COMMIT, PAGE_READWRITE);
```

```
if(pRemoteBuf!=NULL)
```

```
// #3.将szDll路径写入分配的内存
```

```
WriteProcessMemory(hProcess, pRemoteBuf, (LPVOID)szDllName, dwBufSize, NULL);
```

```
// #4.获取LoadLibraryA() API的地址
```

```
pThreadProc =
```

```
(LPTHREAD_START_ROUTINE)GetProcAddress(GetModuleHandle(LPCTSTR("kernel32.dll")), "LoadLibraryA");
```

```
if(pThreadProc!=NULL)
```

```
// #5.在exe进程中运行线程
```

```
hThread = CreateRemoteThread(hProcess,
```

```
//hProcess
```

```
NULL,
```

```
//lpThreadAttributes
```

```
0,
```

```
//dwStackSize
```

```
pThreadProc,
```

```
//lpStartAddress
```

```
pRemoteBuf,
```

```
//lpParameter
```

```
0,
```

```
//dwCreationFlags
```

```
NULL);
```

```
//lpThreadId
```

函数CreateRemoteThread可以在目标文件进程中创建远程线程。

通过Loadlibrary DLL实现注入

6.5 Z3约束求解器

1. Z3

- **Z3是一个微软出品的SMT问题的开源约束求解器**，能够解决很多种情况下的给定部分约束条件寻求一组满足条件的解的问题（可以理解为自动解方程组）。
- Z3在工业应用中常见于软件验证、程序分析等。由于Z3功能实在强大，也被用于很多其他领域：软件/硬件验证和测试、约束解决、混合系统分析、安全性、生物学（计算机模拟分析）和几何问题。著名的二进制分析框架angr也内置了一个修改版的Z3。

Z3是个开源项目，Github链接：<https://github.com/z3prover>。

(1) Window下安装Z3

下载x64-win版：<https://github.com/Z3Prover/z3/releases>。

解压到D:\z3-4.8.10，可以看到文件夹里包含bin子文件夹，里面有可执行文件z3.exe。

配置PATH。打开“我的电脑”的属性窗口，选择“高级系统设置”，在“高级 环境变量”里，编辑path，添加D:\z3-4.8.10\bin。

安装Python。Windows 10系统中，在命令控制台里输入python3会自动弹出商店安装，也可以自己到网上下载环境进行安装。

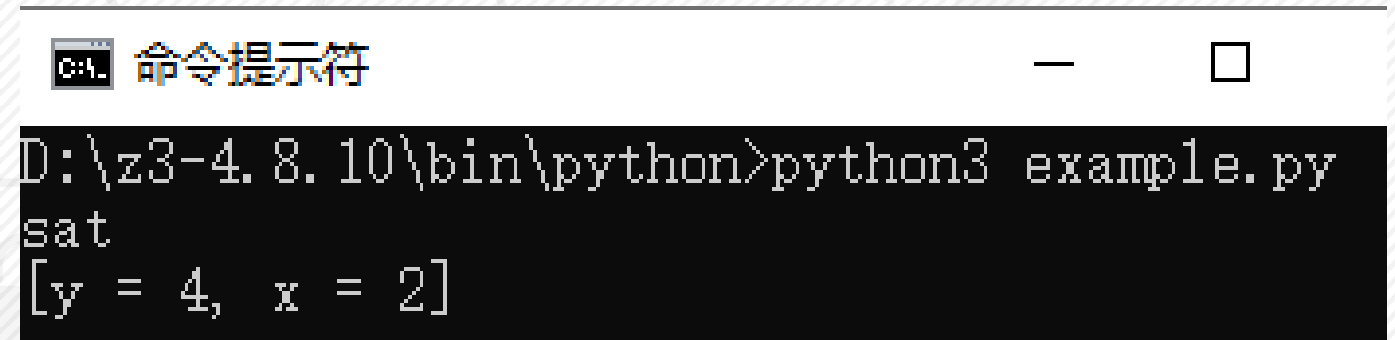
2. Z3常用API

- **Solver()**: 创建一个通用求解器，创建后可以添加约束条件，进行下一步的求解。
- **add()**: 添加约束条件，通常在solver()命令之后。
- **check()**: 通常用来判断在添加完约束条件后，来检测解的情况，有解的时候会回显sat，无解的时候会回显unsat。
- **model()**: 在存在解的时候，该函数会将每个限制条件所对应的解集取交集，进而得出正解。

3. Z3简单示例

```
from z3 import *  
  
x = Real('x')  
y = Real('y')  
s = Solver()  
s.add(x + y > 5, x > 1, y > 1)  
print(s.check())  
print(s.model())
```

打开命令控制台，进入D:\z3-4.8.10\bin\python，
执行example.py，如下：



```
C:\> 命令提示符  
D:\z3-4.8.10\bin\python>python3 example.py  
sat  
[y = 4, x = 2]
```

6.6 Angr应用示例

1. Angr安装

- **Angr是一个基于python的二进制漏洞分析框架**，它将以前多种分析技术集成进来，它能够进行动态的符号执行分析（如KLEE和Mayhem），也能够进行多种静态分析。
- **Windows下安装Angr。首先安装Python3**，如果安装了就忽略。可以到python官方网站下载安装版本，选择将python增加到path中。然后，打开命令控制台，**使用PIP命令安装angr**：`pip install angr`。
- **测试安装**。输入命令python，进入python界面，然后输入`import angr`，如果成功，则说明安装没有问题。

```
C:\Users\liuzheli>python
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import angr
>>>
```


2. Angr示例

- **Angr官方手册。** GitHub上有angr的开源项目<https://github.com/angr>以及相关的文档信息，建议将<https://github.com/angr/angr-doc>里的所有文档以zip方式下载到本地。
- **angr-doc里有各类Example**，展示了Angr的用法，比如cmu_binary_bomb、simple_heap_overflow等二进制爆破、堆溢出等漏洞挖掘、软件分析的典型案例。
- **以sym-write为例子，来说明angr的用法：**
 - ① 怎么使用angr?
 - ② 使用angr能解决什么问题?

```
#include <stdio.h>
char u=0;
int main(void){
    int i, bits[2]={0,0};
    for (i=0; i<8; i++) {
        bits[(u&(1<<i))!=0]++;
    }
    if (bits[0]==bits[1]) {
        printf("you win!");
    }
    else {
        printf("you lose!");
    }
    return 0;
}
```

两条路径

什么样的u可以win?

如果u二进制中1和0个数不同, lose

□ 变量符号化：将u进行符号化

□ 动态符号执行：以具体的数值作为输入执行程序代码，在程序实际执行路径的基础上，用符号执行技术对路径进行分析，提取路径的约束表达式。

□ 获取路径约束条件

□ 约束求解

```
import angr
```

```
import claripy
```

```
def main():
```

```
    p = angr.Project('./issue', load_options={"auto_load_libs": False})
```

```
    state = p.factory.entry_state(add_options={angr.options.SYMBOLIC_WRITE_ADDRESSES})
```

```
    u = claripy.BVS("u", 8)
```

```
    state.memory.store(0x804a021, u)
```

新建一个工程，导入二进制文件，选项是选择不自动加载依赖项，不会自动载入依赖的库

初始化模拟程序状态的SimState对象state，该对象包含了程序内存、寄存器、符号信息等模拟运行时动态数据

创建符号变量u，以8位bitvector形式存在。存储到二进制文件.bss段u的地址

```
sm = p.factory.simulation_manager(state)
```

```
def correct(state):
```

```
    try:
```

```
        return b'win' in state.posix.dumps(1)
```

```
    except:
```

```
        return False
```

```
def wrong(state):
```

```
    try:
```

```
        return b'lose' in state.posix.dumps(1)
```

```
    except:
```

```
        return False
```

```
sm.explore(find=correct, avoid=wrong)
```

```
return sm.found[0].solver.eval_upto(u, 256)
```

```
if __name__ == '__main__':
```

```
    print(repr(main()))
```

创建一个Simulation Manager
对象，管理运行得到的状态对象

定义函数：state.posix.dumps(1)
获得所有标准输出

动态符号执行&得到想要的状态→使用
explore函数进行状态搜寻
也可以写成：sm.explore(find=0x80484e3,
avoid=0x80484f5)

约束求解→获得state之后，通
过solver求解器，求解u的值

实验验证。在windows 10环境下，选择填写的solve.py，点右键选择Edit with IDLE Edit with IDLE 3.9 (64 bit)，将弹出界面，选择Run run model，界面如下



```
IDLE Shell 3.9.2
File Edit Shell Debug Options Window Help
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: D:\angr\angr-doc-master\examples\sym-write\solve.py =====
WARNING | 2021-03-15 15:13:05,188 | [32mangr.storage.memory_mixins.default_filler_mixin[0m | [32mThe program is accessing memory or registers with an unspecified value. This could indicate unwanted behavior. [0m
WARNING | 2021-03-15 15:13:05,241 | [32mangr.storage.memory_mixins.default_filler_mixin[0m | [32mangr will cope with this by generating an unconstrained symbolic variable and continuing. You can resolve this by: [0m
WARNING | 2021-03-15 15:13:05,256 | [32mangr.storage.memory_mixins.default_filler_mixin[0m | [32m1) setting a value to the initial state [0m
WARNING | 2021-03-15 15:13:05,263 | [32mangr.storage.memory_mixins.default_filler_mixin[0m | [32m2) adding the state option ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to make unknown regions hold null [0m
WARNING | 2021-03-15 15:13:05,280 | [32mangr.storage.memory_mixins.default_filler_mixin[0m | [32m3) adding the state option SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to suppress these messages. [0m
WARNING | 2021-03-15 15:13:05,295 | [32mangr.storage.memory_mixins.default_filler_mixin[0m | [32mFilling register edi with 4 unconstrained bytes referenced from 0x8048521 (__libc_csu_init+0x1 in issue (0x8048521)) [0m
WARNING | 2021-03-15 15:13:05,312 | [32mangr.storage.memory_mixins.default_filler_mixin[0m | [32mFilling register ebx with 4 unconstrained bytes referenced from 0x8048523 (__libc_csu_init+0x3 in issue (0x8048523)) [0m
[51, 57, 60, 240, 75, 139, 78, 197, 23, 142, 90, 29, 209, 154, 99, 212, 163, 102, 108, 166, 172, 105, 169, 114, 53, 225, 120, 184, 178, 71, 135, 77, 83, 202, 141, 147, 89, 92, 153, 150, 156, 106, 101, 86, 165, 43, 46, 226, 232, 177, 116, 113, 180, 58, 198, 15, 195, 201, 85, 204, 30, 210, 149, 27, 216, 39, 45, 170, 228, 54]
>>>
```

3. 其他解法

```
import angr
```

```
import claripy
```

```
def hook_demo(state):
```

```
    state.regs.eax = 0
```

```
p = angr.Project("./issue", load_options={"auto_load_libs": False})
```

```
p.hook(addr=0x08048485, hook=hook_demo, length=2)
```

```
state = p.factory.blank_state(addr=0x0804846B, add_options={"SYMBOLIC_WRITE_ADDRESSES"})
```

演示Hook: 0x08048485处指令为xor eax, eax, hook一个函数, 指令长度为2, 实际并没有带来任何变化, 仅为Hook演示

其他创建state方式: 使用blank_state创建状态对象, 指定了程序入口点位置为主函数第一行代码

3. 其他解法

```
u = claripy.BVS("u", 8)
```

```
state.memory.store(0x0804A021, u)
```

```
sm = p.factory.simulation_manager(state)
```

```
sm.explore(find=0x080484DB)
```

状态搜寻：只给定一个条件，因为是分支语句，已经足以确定唯一路径

```
st = sm.found[0]
```

```
print(repr(st.solver.eval(u)))
```

eval(u)替代了原来的eval_upto，
将打印一个结果出来