第八章　漏洞挖掘技术进阶

知识点一：程序切片技术
知识点二：程序切片方法
知识点三：程序插桩技术
知识点四：消息Hook技术
知识点五：API Hook技术

知识点一：程序切片技术

1. 程序切片定义

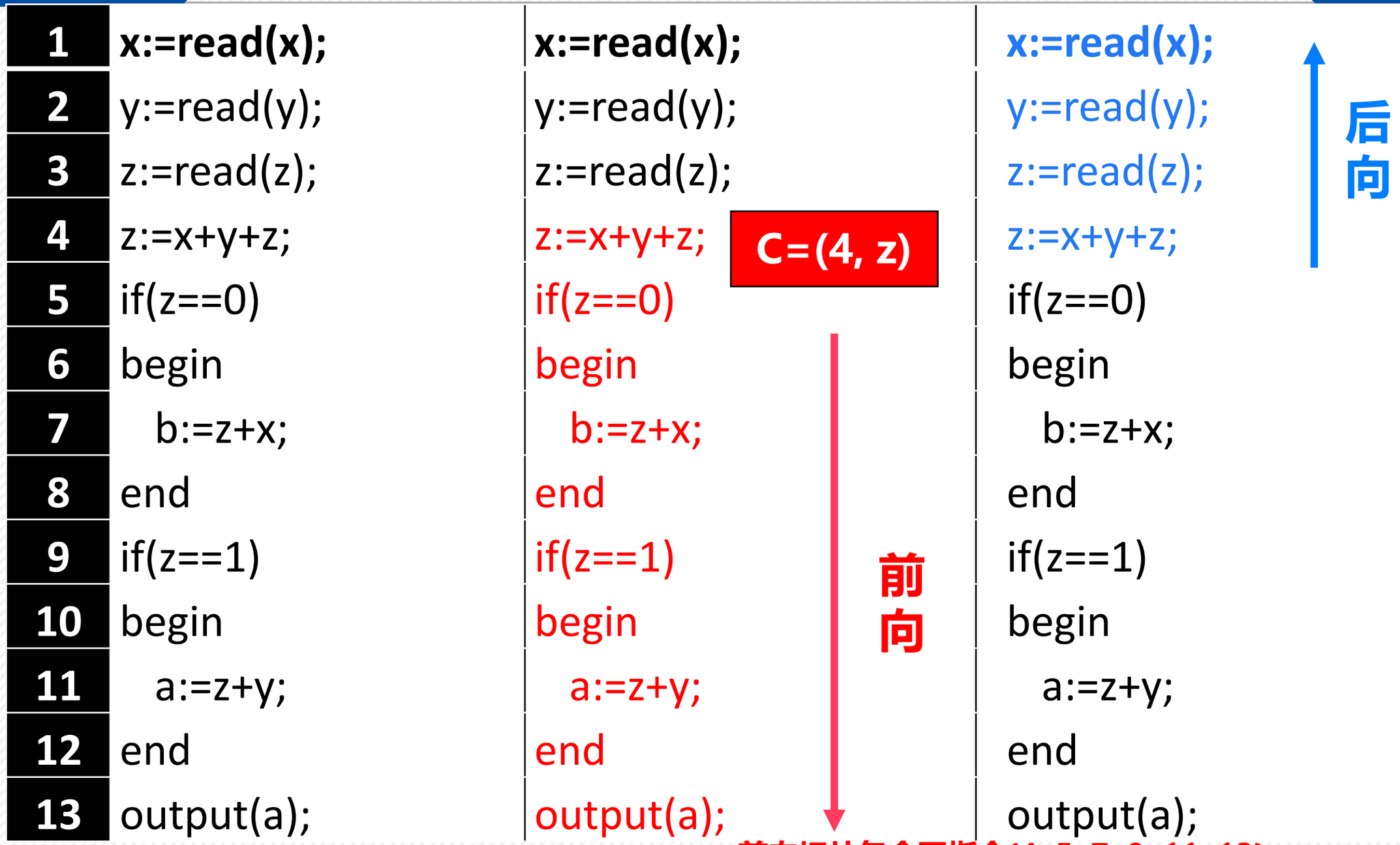
程序切片旨在从程序中提取满足一定约束条件的代码片段（对指定变量施加影响的代码指令，或者指定变量所影响的代码片段），是一种重要的程序分解技术。

程序切片可以从大规模程序中精确定位分析员所关心的代码片段，有效缓解程序规模日益增长带来的分析效率难以同步提高的问题。比如，在漏洞挖掘中，我们可以只关注可执行文件或者源代码某一行敏感函数调用相关的代码片段，来分析是否存在缓冲区溢出漏洞等。

程序切片定义

定义(Mark Weise博士，1979年)：给定一个切片准则 C=(N，V)，其中N表示程序P中的**指令**，V表示**变量集**，程序P关于C的映射即为程序切片。换句话说，一个程序切片是由程序中的一些语句和判定表达式组成的集合。

根据计算方向的不同，程序切片可以分为**前向切片**和**后向切片**。**前向切片的计算方向和程序的运行方向是一致的**。


1 x:=read(x);
2 y:=read(y);
3 z:=read(z);
4 z:=x+y+z;
5 if(z==0)
6 begin
7 b:=z+x;
8 end
9 if(z==1)
10 begin
11 a:=z+y;
12 end
13 output(a);
x:=read(x);
y:=read(y);
z:=read(z);
z:=x+y+z;
C=(4, z)
if(z==0)
begin
b:=z+x;
end
if(z==1)
begin
a:=z+y;
end
output(a);
前向
x:=read(x);
y:=read(y);
z:=read(z);
z:=x+y+z;
if(z==0)
begin
b:=z+x;
end
if(z==1)
begin
a:=z+y;
end
output(a);
后向
前向切片包含了指令{4, 5, 7, 9, 11, 13}

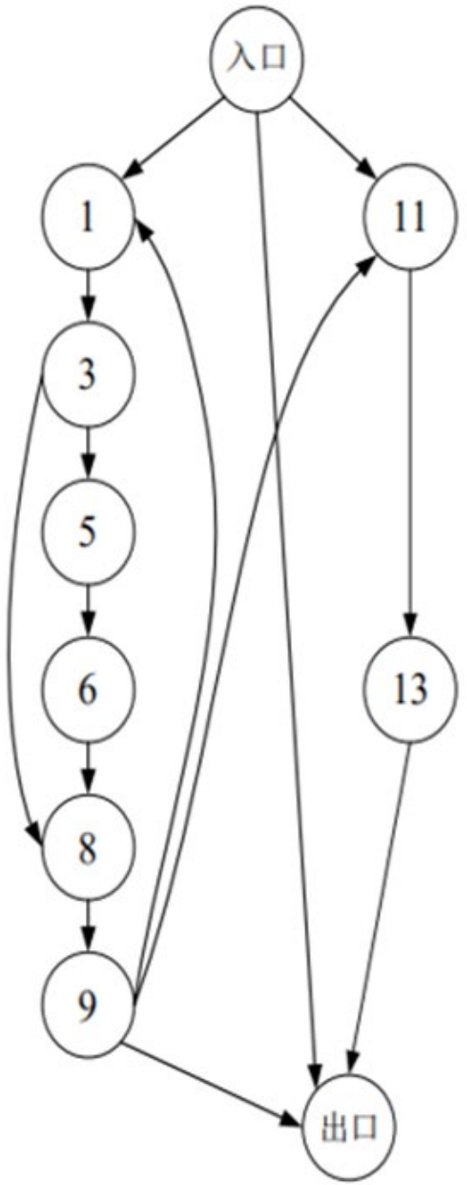
2. 控制流图

控制流图（Control Flow Graph，简称CFG）也叫控制流程图，**是一个过程或程序的抽象表现，代表了一个程序执行过程中会遍历到的所有路径。**

控制流图：一个程序的控制流图CFG可以表示为一个四元组，形如G = (V, E, s, e)，其中V表示变量的集合，**E表示表示边的集合**，s表示控制流图的入口，e表示控制流图的出口。

程序中的每一条指令都映射为CFG上的一个结点，具有**控制依赖**关系的结点之间用一条边连接。

```
1   while i<1
2   begin
3      if c==2
4      begin
5         c:=y;
6         x:=25;
7      end
8      i:=i+c;
9      j:=j+1;
10  end
11  if j==0
12  begin
13     temp:=i;
14  end
```



CFG图

程序中的**控制依赖**关系有两种来源：

（1）程序上下文；

（2）控制指令。

控制指令对应了分支结构或循环结构，结构里面的所有指令对结构入口的控制指令存在控制依赖关系。如果一条指令不在分支结构或循环结构里面，则该指令依赖于程序的入口。

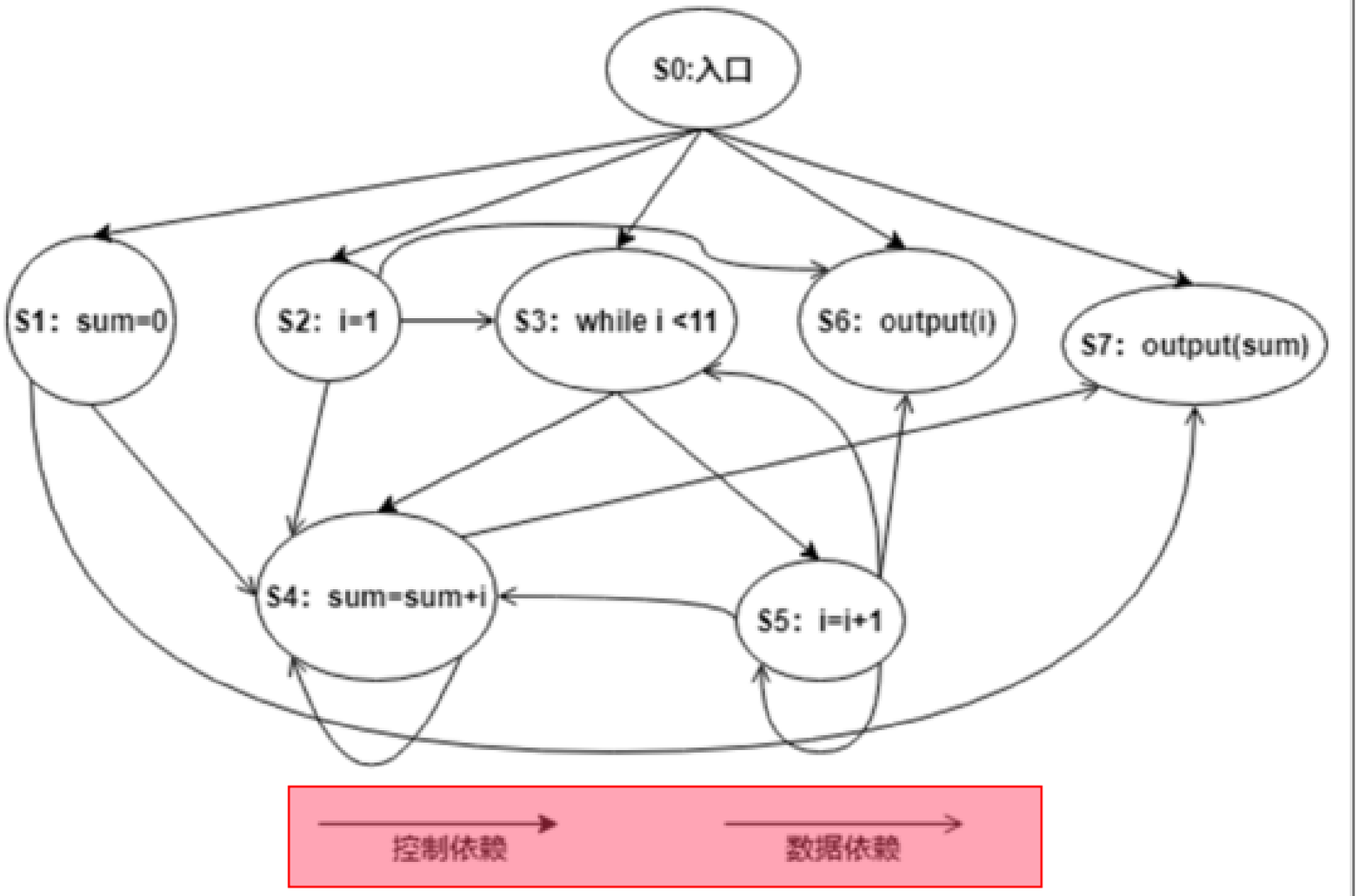
3.程序依赖图

程序依赖图：程序依赖图（Program Dependence Graph，PDG）可以表示为一个五元组，形如G = (V, DDE, CDE, s, e)，其中V表示变量的集合，**DDE表示数据依赖边的集合，CDE表示控制依赖边的集合**，每条边连接了图中的两个结点，程序中的每一条指令都映射为PDG上的一个结点。s表示程序依赖图的入口结点，e表示程序依赖图的出口结点。

控制依赖：表示两个基本块在程序**流程上存在的依赖关系**。

数据依赖：表示程序中引用某变量的基本块（或者语句）**对定义该变量的基本块的依赖**，即是一种“**定义-引用**”依赖关系。

```
Procedure M()
begin
    int sum:=0;
    int i:=1;
    while i<11
    begin
        sum=sum+i;
        i=i+1;
    end
    output(i);
    output(sum);
end
```



控制流图结点之间的边只反映出了程序指令之间的部分控制依赖关系。程序依赖图PDG需要将一个函数中所有的数据依赖和控制依赖关系遍历出来。

4. 系统依赖图

系统依赖图（System Dependence Graph，SDG）：可以表示为一个七元组，形如G = (V,DDE, CDE, CE, TDE, s, e)，其中V变量的集合，**DDE表示数据依赖边的集合，CDE表示控制依赖边的集合，CE表示函数调用边，TDE表示参数传递造成的传递依赖边的集合**，结点s表示系统依赖图的入口结点，结点e表示系统依赖图的出口结点。

SDG在PDG的基础上进行了扩充，系统依赖图中加入了对函数调用的处理。

知识点二：程序切片方法

1. 工作原理

- **在实际的程序调试过程中，通常程序员只关注程序的部分行为。**

切片准则包含两个要素，即切片目标变量（如变量z），以及开始切片的代码位置（如z所在的代码位置：第12行）。严格来说，程序P的切片准则是二元组<n,V>，其中n是程序中一条语句的编号，V是切片所关注的变量集合，该集合是P中变量的一个子集。

- **切片语句可以利用数据依赖和控制依赖分析方法来获取。**

```
1:  int main(){
2:      int x,y,z;
3:      int i=0;
4:      z=0;
5:      y=getchar();
6:      for(;i<100;i++)
7:        if(i%2==1)
8:            x+=y*i;
9:        else
10:           z+=1;
11:     printf("%d\n",x);
12:     printf("%d\n",z);
13: }
```

关于<12，{z}>的后向切片

数据依赖：10和4

控制依赖：6和7

6的数据依赖 3

切片：3 4 6 7 9 10

关于<11，{x}>的后向切片

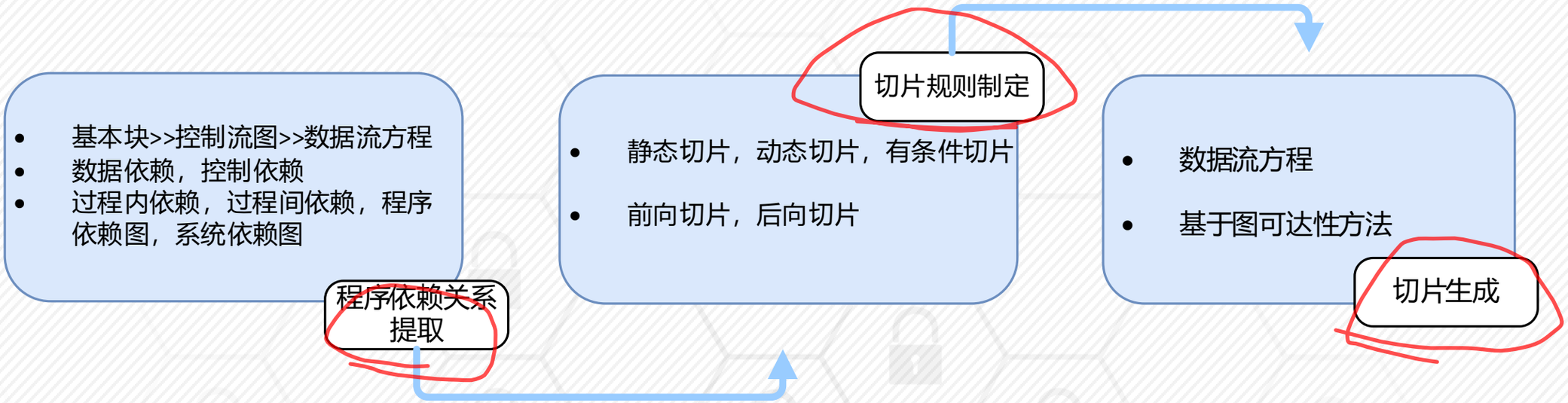
数据依赖：8~

8的数据依赖 5 3

控制依赖：6和7

6的数据依赖 3

切片：3 5 6 7 8



程序切片通常包括3个步骤：程序依赖关系提取、切片规则制定和切片生成。

- **程序依赖关系提取**主要是从程序中提取各类消息，包括控制流和数据流信息，形成程序依赖图。
- **切片规则制定**主要是依据具体的程序分析需求设计切片准则。
- **切片生成**则主要是依据前述的切片准则选择相应的程序切片方法，然后对第一步中提取的依赖关系进行分析处理，从而生成程序切片。

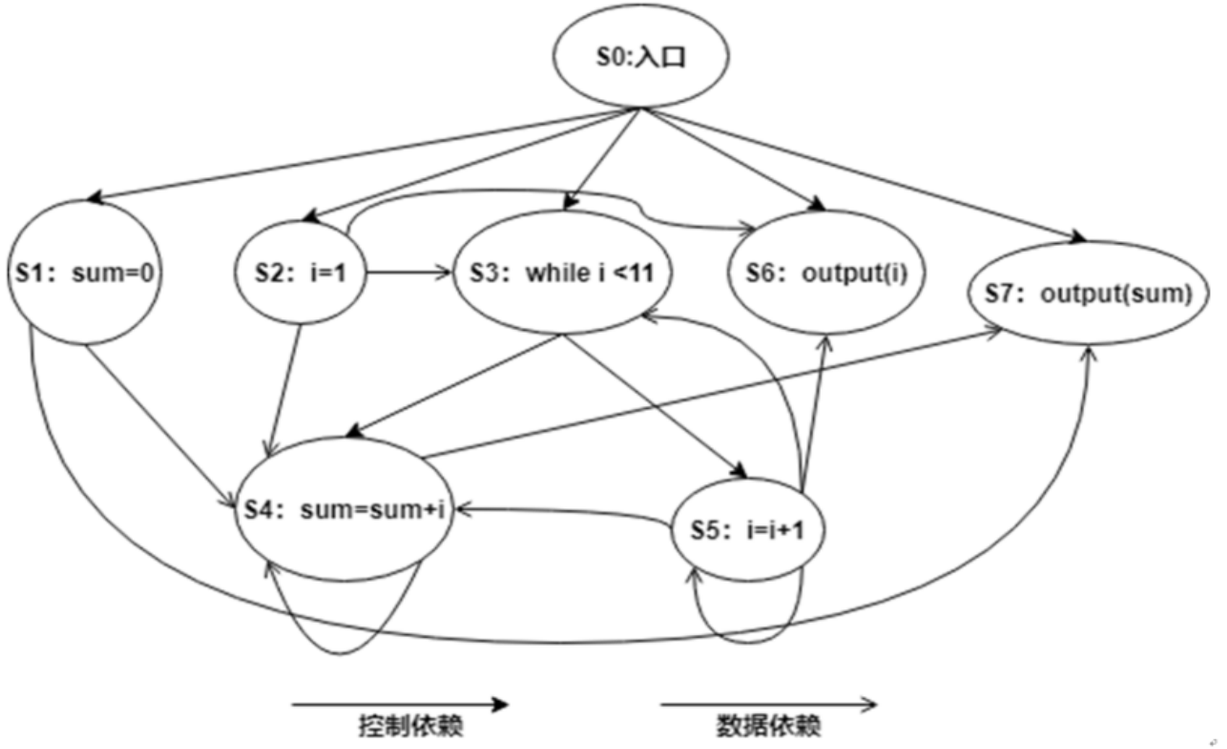
2. 图可达算法

- 程序切片技术有多种计算方法，例如：数据流方程算法、图可达性算法、基于波动图的切片算法、基于信息流关系的切片算法等。其中，**最常用和最主流的算法是数据流方程算法与图可达性算法。**
- 图可达性算法根据程序建模的不同分为许多子类，最常用的包括基于程序依赖图的图可达性算法和基于系统依赖图的图可达性算法。
- 在程序依赖图PDG中，具有直接依赖关系和间接依赖关系的结点都用一条边连结，因此基于 PDG 的图可达性切片算法只需从指定结点遍历每一个具有依赖关系的结点即可，计算过程比较简单直观。

将基于PDG的图可达性切片过程记为PDGSlice，它的详细步骤如下：

- **输入：结点Node**
- **输出：结点集VisitedNodes**
- 步骤1：判断Node是否在结点集VisitedNodes，结果为是，则return；结果为否，则进入步骤2；
- 步骤2：将Node添加到VisitedNodes中；
- 步骤3：在程序依赖图中遍历Node依赖的结点，得到结点集Pred；
- 步骤4：对于每一个pred∈Pred，迭代调用PDGSlice(pred)。

```
Procedure M()
begin
    int sum:=0;
    int i:=1;
    while i<11
    begin
        sum=sum+i;
        i=i+1;
    end
    output(i);
    output(sum);
end
```



切片准则C=(S7, sum) 后向

当前节点	前驱节点	VisitedNodes
S7	{ S0，S1，S4 }	Ø
S4	{ S1，S2，S3，S4，S5 }	{ S7 }
S5	{ S3，S5 }	{ S4，S7 }
S3	{ S0，S2，S5 }	{ S4，S5，S7 }
S2	{ S0 }	{ S3，S4，S5，S7 }
S0	-	{ S2，S3，S4，S5，S7 }
S1	{ S0 }	{ S0，S2，S3，S4，S5，S7 }
		{ S0，S1, S2，S3，S4，S5，S7 }

3. 动态切片

- 从切片角度，切片分为静态程序切片、动态程序切片和条件切片等。
- 由于静态切片中包含了到达兴趣点的所有可能路径，而对于程序的某一次特定执行，其中的许多路径实际上是不会被执行的。
- **动态切片需要考虑程序的特定输入**，切片准则是一个三元组(N，V，I)，其中 N 是指令集合，V 是变量集合，**I 是输入集合。**

1	**x:=read(x);**	**x:=read(x);**
2	y:=read(y);	y:=read(y);
3	z:=read(z);	z:=read(z);
4	z:=x+y+z;	z:=x+y+z;
5	if(z==0)	if(z==0)
6	begin	begin
7	b:=z+x;	b:=z+x;
8	end	end
9	if(z==1)	if(z==1)
10	begin	begin
11	a:=z+y;	a:=z+y;
12	end	end
13	output(a);	output(a);

C1=(13, a, x=1, y=1, z=0)

C2=(13, a, x=0, y=1, z=0)

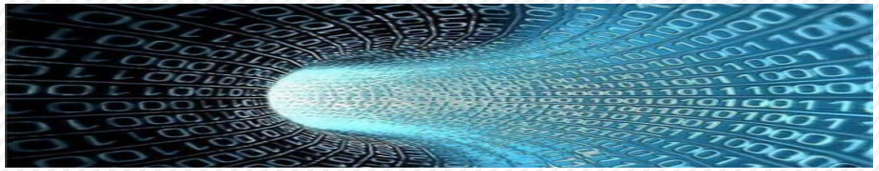
- 动态切片可以看做静态切片的子集
- 当图可达算法应用到动态切片中，可以通过**裁剪程序依赖图**来实现
- **条件切片**的切片准则也是一个三元组，形为C = (N, V, F_V)，其中 N 和 V 的含义同静态准则相同，F_V是 V 中变量的逻辑约束。
- **静态切片和动态切片可以看做条件切片的两个特例**：当F_V中的约束条件为空时，得到的切片是静态切片；当F_V中的约束固定为某一特定条件时，得到的切片是动态切片。

知识点三：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

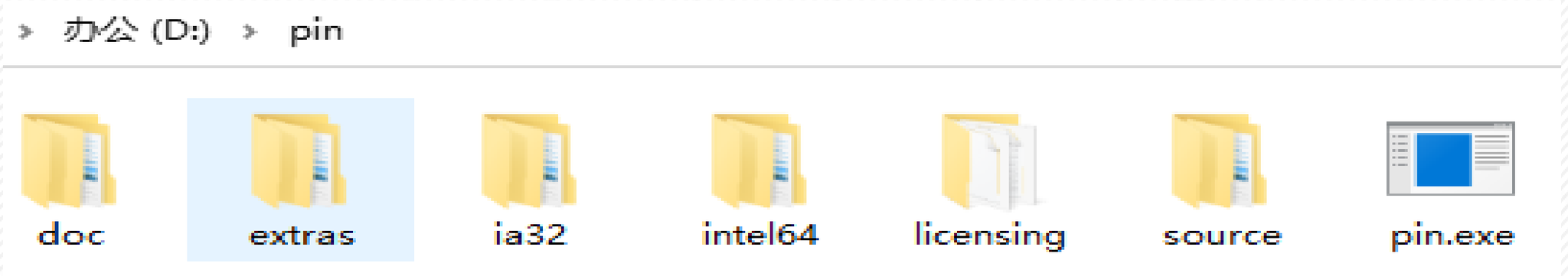
- **源代码插桩**是指在被测程序运行之前，通过自动化工具或者程序员手动在需要收集信息的地方插入探针，之后重新编译运行被测程序。
- **静态二进制插桩**和源代码插桩类似，都是在程序运行之前插入探针，与源代码插桩不同是静态二进制插桩直接对程序编译之后的二进制机器码进行插桩。编写难度更大、可移植性更差。
- **动态二进制插桩**在程序运行时，直接接管被测程序并且截获其二进制指令并插入探针。插桩程序难度更大，程序运行开销也越大。

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，已经定了好了很多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
pinatracce	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_mallocttace	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-pro
tector -fno-exceptions -funwind-tables -fasynchronous-unwind-tables -fno-rtti
 -DTARGET_IA32E -DHOST_IA32E -fPIC -DTARGET_LINUX -fabi-version=2 -faligned-n
ew -I../../../source/include/pin -I../../../source/include/pin/gen -isystem /
home/kali/Downloads/pin-3.18/extras/stlport/include -isystem /home/kali/Downl
oads/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/a
sm-x86 -I../../../extras/components/include -I../../../extras/xed-intel64/inc
lude/xed -I../../../source/tools/Utils -I../../../source/tools/InstLib -O3 -f
omit-frame-pointer -fno-strict-aliasing  -c -o obj-intel64/inscount0.o inscou
nt0.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
    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。

知识点四：消息Hook

1. Hook概念

Hook（钩子）

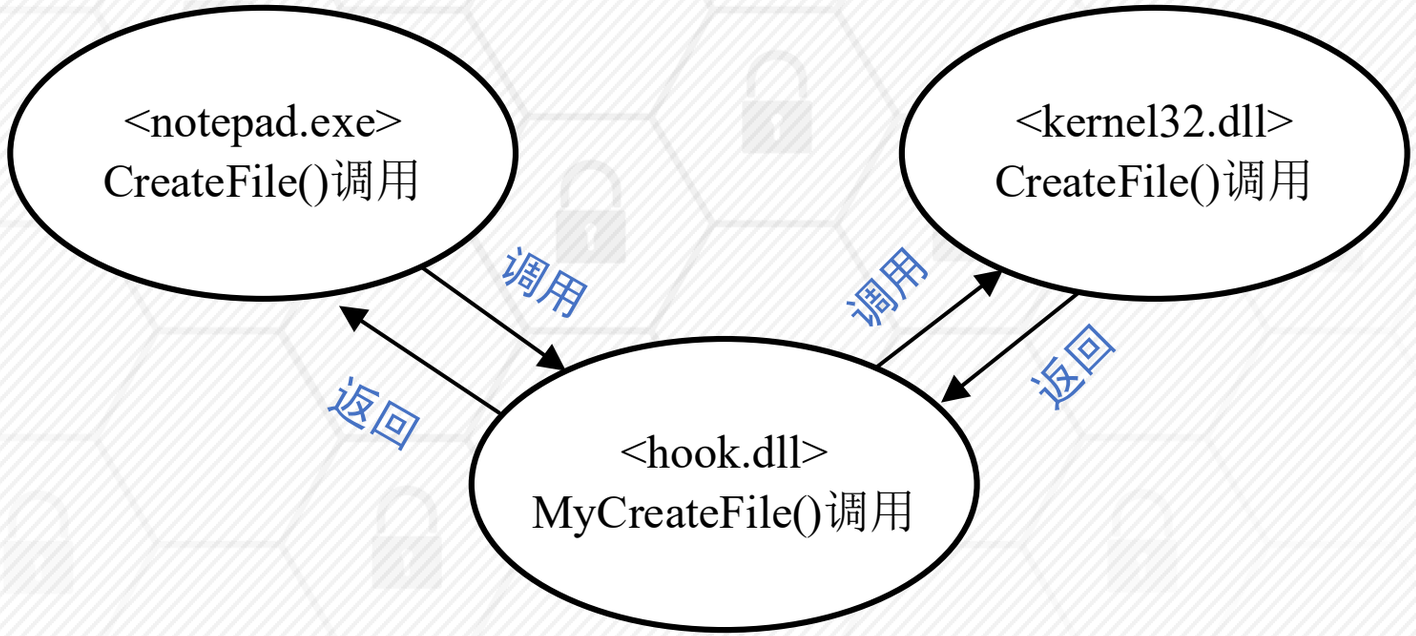
Hook（钩子），是一种过滤（或叫挂钩）消息的技术。

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，现在被广泛应用在软件分析、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;

        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);
}
```

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

```
#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

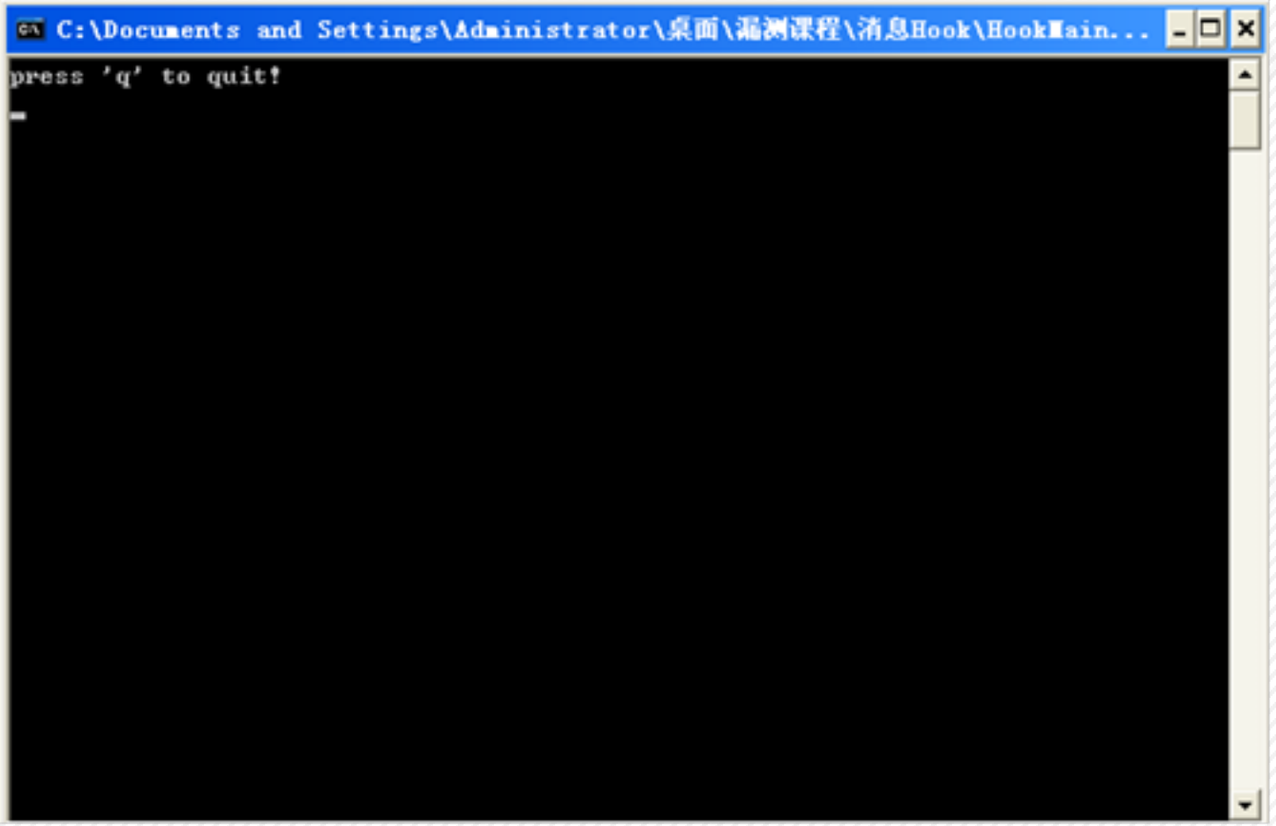
新建一个VC6的控制台程序，添加源文件HookMain.cpp如下：

```
#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。



知识点五：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

第一步 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! MylstrcpyW ()钩取kernel32! lstrcpyW ()
        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;
}
```

第二步 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