第七章 漏洞挖掘基础

1 方法概述

1.1 漏洞挖掘方法分类

漏洞挖掘主要分为两种方法：

- 静态分析技术：**词法分析、数据流分析、控制流分析、模型检查、定理证明、符号执行、污点传播分析等**
  - 不需要运行程序、分析效率高、资源消耗低
- 动态分析技术：**模糊测试、动态污点分析、动态符号执行**
  - 需要运行程序、准确率非常高，误报率很低

1.2 符号执行

符号执行：**使用符号值替代具体值，模拟程序的执行**。

三个关键点：

- 变量符号化
- 程序执行模拟
- 约束求解

变量符号化是指用一个符号值表示程序中的变量，所有**与被符号化的变量相关的变量取值都会用符号值或符号值的表达式**表示。

程序执行模拟最重要的是**运算语句和分支语句的模拟**：

- **对于运算语句**，由于符号执行使用符号值替代具体值，所以无法直接计算得到一个明确的结果，需要**使用符号表达式的方式表示变量的值**。
- **对于分支语句**，每当遇到分支语句，原先的一条路径就会分裂成多条路径，符号执行会记录每条分支路径的约束条件。最终，通过采用合适的路径遍历方法，符号执行可以**收集到所有执行路径的约束条件表达式**。

约束求解主要负责**路径可达性进行判定及测试输入生成**的工作。对一条路径的约束表达式，可以采用**约束求解器**进行求解：

- 如有解，该路径是**可达**的，可以得到到达该路径的输入；
- 如无解，该路径是**不可达**的，也无法生成到达该路径的输入。

符号执行的优缺点：优点是代价小，效率高；但是我们的可能的路径会随着程序规模的增长而呈指数级增长，所以可能有时候会难以分析

符号执行的用处：广泛用于软件测试和漏洞挖掘中，通过创建高覆盖率的测试用例，通过约束求解的方法来求解是否存在满足漏洞分析的规则的值

举一个例子：

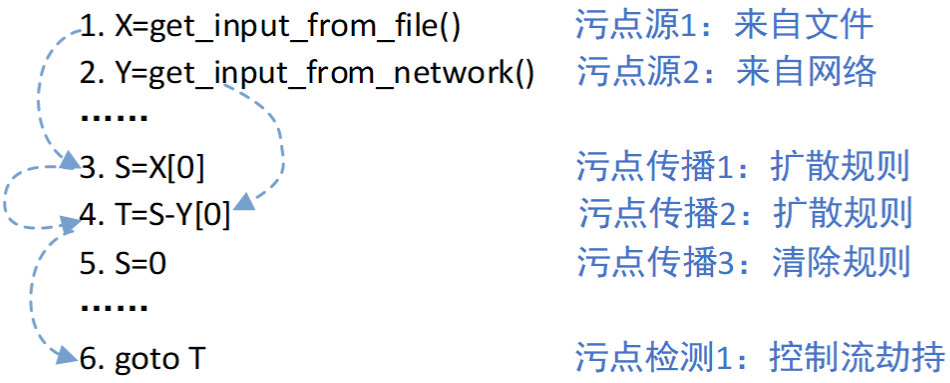
```
int a[10];
scanf("%d", &i);
if (i > 0) {
    if (i > 10)
        i = i % 10;
    a[i] = 1;
}
```

这一段，我们就可以通过符号执行的方式去挖掘漏洞，我们可以发现两个条件判定语句中，存在着对10这个元素的漏判，所以我们输入10就可以成功引发漏洞了

1.3 污点分析

通过标记程序中的数据（外部输入数据或者内部数据）为污点，**跟踪**程序处理污点数据的内部流程，进而帮助人们进行**深入的程序分析和理解**

我们首先需要**确定污点源**，然后需要**标记和分析污点**。



这样我们就可以发现程序的内在逻辑与漏洞了

污点分析**核心三要素**：

- 污点源：是污点分析的目标来源（Source点），通常表示来自程序外部的不可信数据，包括硬盘文件内容、网络数据包等。
- 传播规则：是污点分析的计算依据，通常包括污点扩散规则和清除规则，其中普通赋值语句、计算语句可使用扩散规则，而常值赋值语句则需要利用清除规则进行计算。
- 污点检测：是污点分析的功能体现，其通常在程序执行过程中的敏感位置（Sink点）进行污点判定，而敏感位置主要包括程序跳转以及系统函数调用等。

优缺点：

适用于由输入参数引发漏洞的检测，比如SQL注入漏洞等。

污点分析技术具有**较高的分析准确率**，然而针对大规模代码的分析，由于**路径数量较多**，因此其**分析的性能会受到较大的影响**。

2 词法分析

通过对代码进行基于文本或字符标识的**匹配分析对比**，以查找符合特定特征和词法规则的**危险函数、API或简单语句组合**。就是说，将代码文本和归纳好的缺陷模式进行匹配，然后去发现漏洞。

优缺点：优点是算法较简单，性能高；缺点是只能进行表面的词法检测，在深度的检测中会出现大量漏报和误报，对于高危漏洞无法进行很好的检测

具体内容见PPT，有两个实验，需要看一下

3 数据流分析

是一种用来获取**相关数据沿着程序执行路径流动**的信息分析技术，分析对象是程序执行路径上的**数据流动或可能的取值**，分为**过程内分析**和**过程间分析**

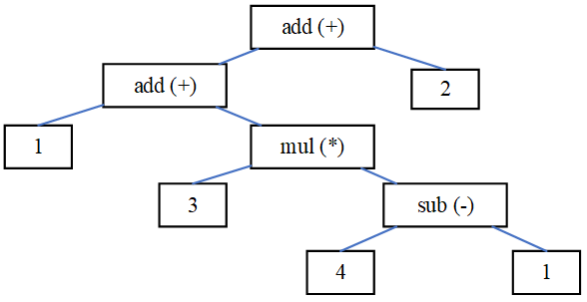
过程内分析只针对程序中函数内的代码进行分析，又分为：

1. 流不敏感分析（flow insensitive）：按代码行号从上而下进行分析；
2. 流敏感分析（flow sensitive）：首先产生程序控制流图（Control Flow Graph，CFG），再按照CFG的拓扑排序正向或逆向分析；
3. 路径敏感分析（path sensitive）：不仅考虑到语句先后顺序，还会考虑语句可达性，即会沿实际可执行到路径进行分析。

过程间分析则考虑函数之间的数据流，即需要跟踪分析目标数据在函数之间的传递过程。

1. 上下文不敏感分析：忽略调用位置和函数参数取值等函数调用的相关信息。
2. 上下文敏感分析：对不同调用位置调用的同一函数加以区分。

程序代码模型：数据流分析使用的，主要包括程序代码中间的表示，以及一些关键的数据结构。我们引入抽象语法树，其描述了程序的过程内代码的控制流结构。

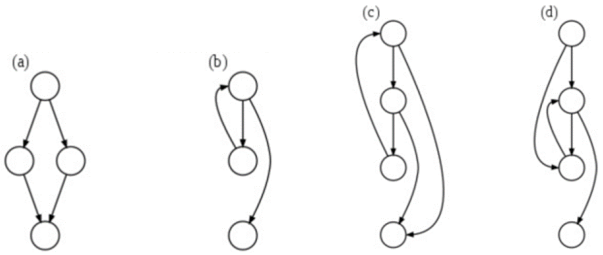


三地址码。三地址码（Three address code，TAC）是一种中间语言，由一组类似于汇编语言的指令组成，每个指令具有**不多于三个的运算分量**。每个运算分量都像是一个寄存器。

常见的三地址码：

- x = y op z： 表示 y 和 z 经过 op 指示的计算将结果存入 x
- x = op y： 表示 y 经过操作 op 的计算将结果存入 x
- x = y： 表示赋值操作
- goto L： 表示无条件跳转
- if x goto L： 表示条件跳转
- x = y[i]： 表示数组赋值操作
- x = &y 、 x = *y： 表示对地址的操作
- param x1, param x2, call p： 表示过程调用 p(x1, x2)

还有控制流图，**通常是指用于描述程序过程内的控制流的有向图**。控制流由节点和有向边组成。节点可以是单条语句或程序代码段。有向边表示节点之间存在潜在的控制流路径。



上面是一些常见的控制流路径

调用图。调用图（Call Graph，CG）是描述程序中过程之间的调用和被调用关系的有向图，满足如下原则：对程序中的每个过程都有一个节点；对每个调用点都有一个节点；如果调用点c调用了过程p，就存在一条从c的节点到p的节点的边。

基于数据流的漏洞分析：

1. 首先，进行**代码建模**，将代码构造为抽象语法树或程序控制流图；
2. 然后，**追踪获取变量的变化信息**，根据**漏洞分析规则**检测安全缺陷和漏洞。

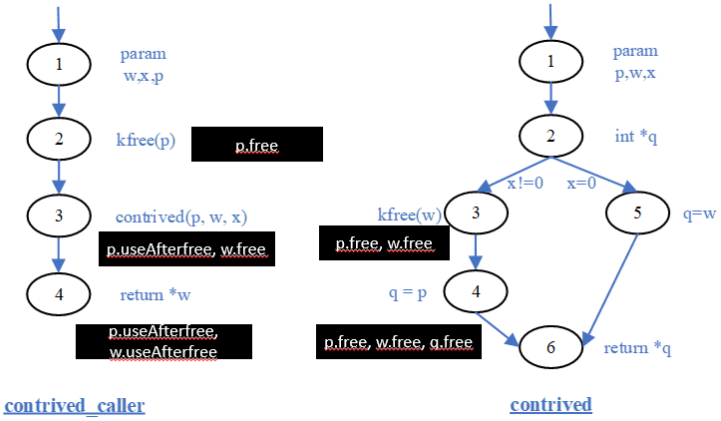
对于逻辑复杂的程序代码，数据流复杂，所以准确率较低，误报率较高

举个例子：

```
int contrived(int *p, int *w, int x) {
    int *q;
    if (x) {
        kfree(w); // w free
        q = p;
    }else
        q=w;
    return *q; // p use after free
}
int contrived_caller(int *w, int x, int *p) {
    kfree(p); // p free
    [...]
    int r = contrived(p, w, x);
```

```
    [...]
    return *w; // w use after free
}
```

我们根据数据流分析来检测指针变量



因此我们发现，只有1256这一条路径是安全的，12346这一条存在着指针的漏洞。

4 模糊测试

是一种自动化或半自动化的安全漏洞检测技术，通过向目标软件输入大量的**畸形数据**并**监测目标系统的异常**来发现潜在的软件漏洞。

模糊测试属于**黑盒测试**的一种，它是一种有效的动态漏洞分析技术，黑客和安全技术人员使用该项技术已经发现了大量的未公开漏洞。

它的**缺点是畸形数据的生成具有随机性**，而随机性造成代码覆盖不充分导致了**测试数据覆盖率不高**。

分类：

- 基于生成的模糊测试：它是指**依据特定的文件格式或者协议规范组合生成测试用例**，该方法的关键点在于既要遵守被测程序的输入数据的规范要求，又要能变异出区别于正常的数据
- 基于变异的模糊测试：它是指**在原有合法的测试用例基础上，通过变异策略生成新的测试用例**。变异策略可以是随机变异策略、边界值变异策略、位变异策略等等，但前提条件是给定的初始测试用例是合法的输入。

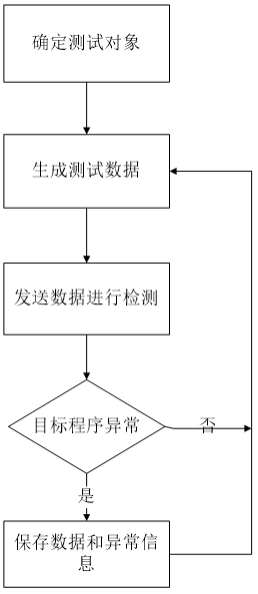
步骤：

1. 确定测试对象和输入数据：对模糊测试来说**首要的问题是确定可能的输入数据，畸形输入数据的枚举对模糊测试至关重要**。
2. 生成模糊测试数据：一旦确定了输入数据，接着就可以生成模糊测试用的畸形数据。根据目标程序及输入数据格式的不同，可相应选择不同的测试数据生成算法。

3. 检测模糊测试数据：检测模糊测试数据的过程首先要**启动目标程序**，然后**把生成的测试数据输入到应用程序中进行处理**。

4. 监测程序异常：实时监测目标程序的运行，就能追踪到引发目标程序异常的源测试数据。

5. 确定可利用性：还需要**进一步确定所发现的异常情况是否能被进一步利用**。



该方法的测试具有一定的随机性，不是所有的错误都能被检测出来

我们为了解决模糊测试的随机性，我们往里面引入了**基于符号执行、污点传播分析等可进行程序理解的方法，在实现程序理解的基础上，有针对性的设计测试数据的生成**，从而实现了比传统的随机模糊测试更高的效率，这种结合了程序理解和模糊测试的方法，称为智能模糊测试(smart Fuzzing)技术。

智能模糊测试的步骤：

1. 反汇编：智能模糊测试的前提，是对可执行代码进行输入数据、控制流、执行路径之间相关关系的分析。为此，首先对**可执行代码进行反汇编得到汇编代码**，在汇编代码的基础上才能进行上述分析。

2. 中间语言转换：需要**将汇编代码转换成中间语言**，由于中间语言易于理解，所以为可执行代码的分析提供了一种有效的手段。

3. 采用智能技术分析输入数据和执行路径的关系：通过符号执行和约束求解技术、污点传播分析、执行路径遍历等技术手段，**检测出可能产生漏洞的程序执行路径集合和输入数据集合**。

4. 利用分析获得的输入数据集合，对执行路径集合进行测试：采用上述智能技术获得的输入数据集合进行安全检测，使**后续的安全测试检测出安全缺陷和漏洞的机率大大增加。**

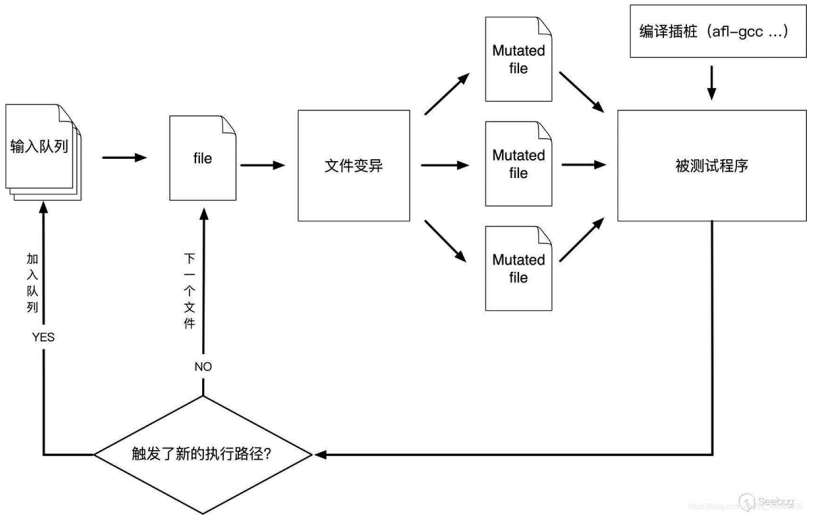
核心思想：**在于以尽可能小的代价找出程序中最有可能产生漏洞的执行路径集合，从而避免了盲目地对程序进行全路径覆盖测试，使得漏洞分析更有针对性。**

5 AFL模糊测试框架

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。

AFL工作流程：

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



第八章 漏洞挖掘技术

1 程序切片技术

1.1 概述

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

定义：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**。

分类：

- 前向切片，计算方向与程序相同
- 后向切片

C=(4, z)指的就是，从C代码的第四行开始，做前向切片，只关注我们的z变量

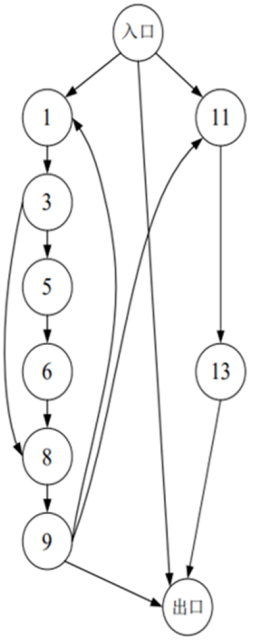
1.2 控制流图(CFG)

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

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

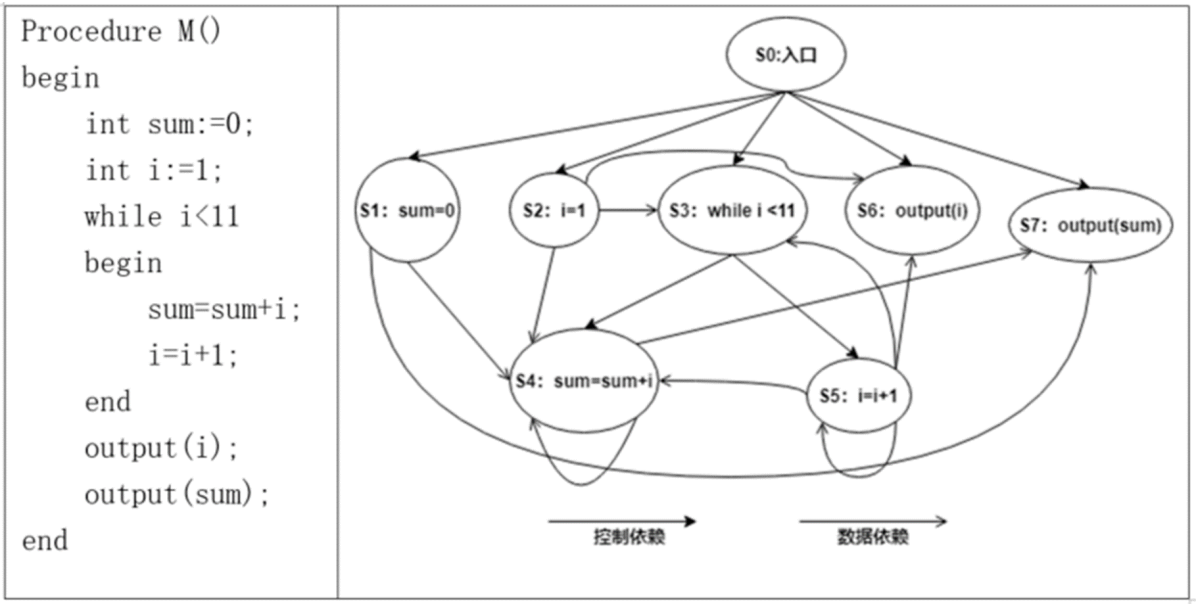
比如说我们的356语句受到1的影响，那么我们就将其标在1的后面

控制依赖的来源：**程序上下文；控制指令**

1.3 程序依赖图(PDG)

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



我们将控制依赖用粗的箭头表示，将数据依赖用细的箭头表示

举个例子：S4语句，sum的值必定受到S1的影响，所以是数据依赖；还收到S5中的i的数据依赖影响；还收到自己的控制依赖影响，因此是三个箭头

1.4 系统依赖图(SDG)

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

2 程序切片方法

2.1 定义

包含两个要素，即

- 切片目标变量（如变量z）
- 开始切片的代码位置（如z所在的代码位置：第12行）

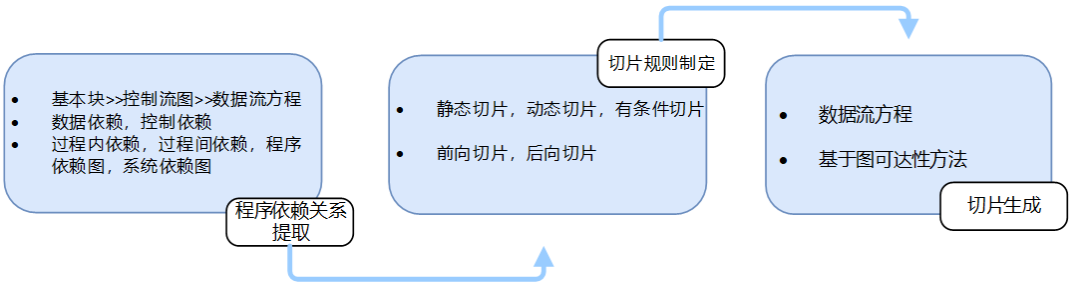
程序P的切片准则是**二元组<n,V>**

- 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

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。



2.2 图可达算法

切片过程

- 输入：结点Node
- 输出：结点集VisitedNodes

步骤

- 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)

2.3 动态切片

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

我们输入C1=(13, a, x=1, y=1，z=0)，这样就是说明了z的值，这样我们就可以将一些没有用的代码给剪掉

动态切片就是跟我们的未知数的取值有关，是静态切片的子集

2.4 条件切片

条件切片的切片准则也是一个三元组，形为C = (N, V, FV)，其中 N 和 V 的含义同静态准则相同，FV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FV中的约束条件为空时，得到的切片是静态切片；当FV中的约束**固定为某一特定条件时，得到的切片是动态切片**

3　程序插桩技术

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

分类：

- 源代码插桩
- 静态二进制插桩
- 动态二进制插桩

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**

具体见**实验8**

4　消息Hook技术

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**

官方函数SetWindowsHookEx用于设置消息Hook

```
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
);
```

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

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）

DLL基本格式：

导入函数

```
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD dwReason, LPVOID lpvReserved){
    switch( dwReason )
    {
        case DLL_PROCESS_ATTACH://动态链接库加载
            g_hInstance = hinstDLL;
            break;

        case DLL_PROCESS_DETACH://动态链接库卸载
            break;
    }
    return TRUE;
}
```

导出函数

```
#ifdef __cplusplus
extern "C" {
#endif
    __declspec(dllexport) void HookStart()//hook开始执行
    {
        g_hHook = SetWindowsHookEx(WH_KEYBOARD, KeyboardProc, g_hInstance, 0);
    }

    __declspec(dllexport) void HookStop()//hook停止
    {
        if( g_hHook )
        {
            UnhookWindowsHookEx(g_hHook);
            g_hHook = NULL;
        }
    }
#ifdef __cplusplus
}
#endif
```

5 API Hook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

分类：

- 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
- 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
- EAT Hook

6 符号执行基本原理

6.1 程序执行状态

符号执行具体执行时，程序状态中通常包括：程序变量的具体值、程序指令计数和路径约束条件pc（path constraint）

pc是符号执行过程中对路径上条件分支走向的选择情况，根据状态中的pc变量就可以确定一次**符号执行的完整路径**。pc初始值为true

遇到条件分支就是左右分叉，运算语句就是代入未知数

6.2 符号传播

建立符号变量传播的关系，并且更新映射的关系——将对应**内存地址的数据**进行变化

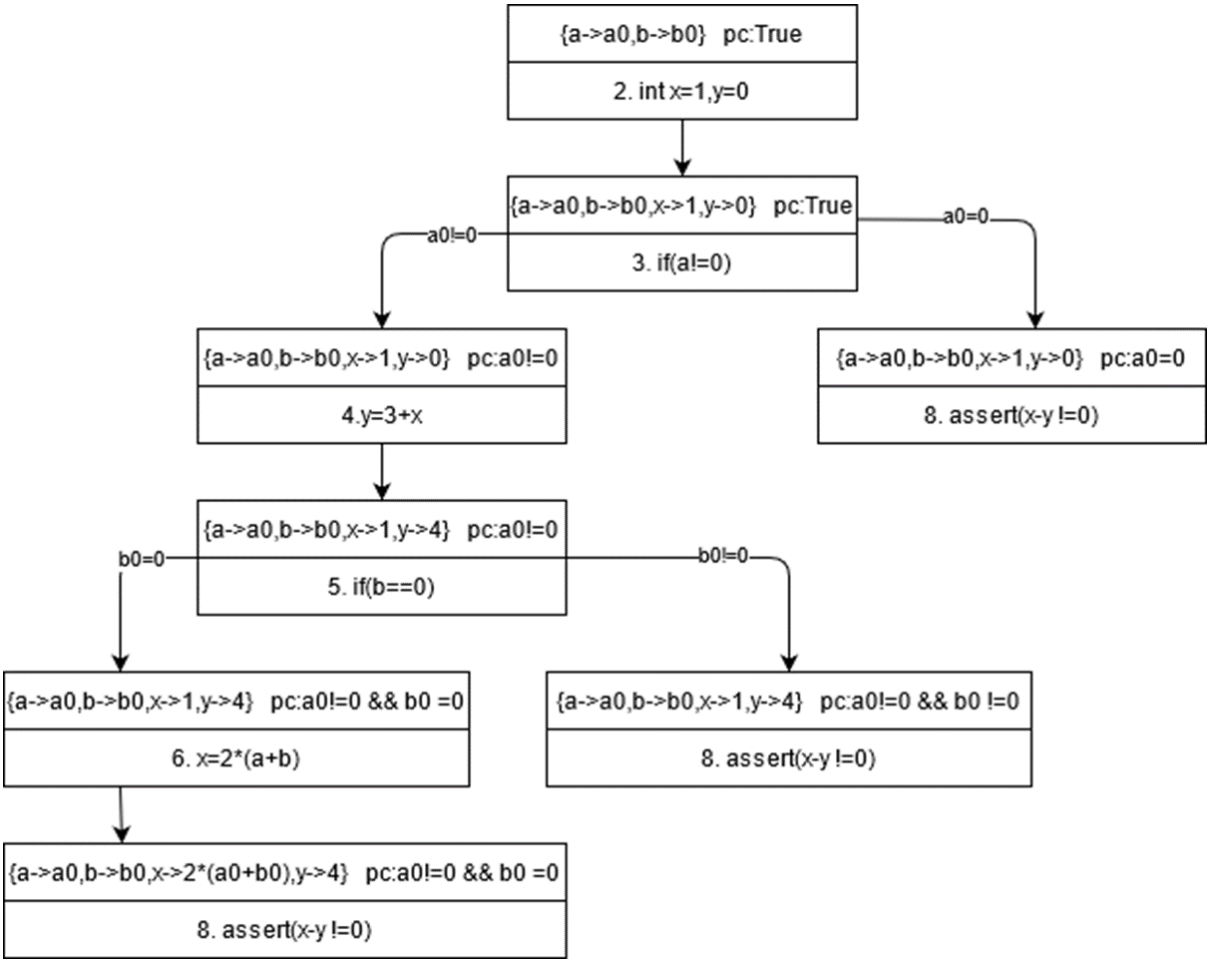
```
int x;
int y, z;
y=x*3;
z=y+5;
```

符号量的内存地址	符号值
add_x	X
add_y	X*3
add_z	X*3 + 5

6.3 符号执行树

程序的所有执行路径可以表示为树，叫做执行树。符号执行过程也是对执行树进行遍历的过程

```
1 void foobar(int a,int b){
2  int x=1,y=0;
3  if(a != 0){
4    y = 3+x;
5    if (b ==0)
6         x = 2*(a+b);
7   }
8  assert(x-y ! =0)
```



结合`assert`的约束`x-y!=0`就可以进行求解出触发约束的输入

6.4 约束求解

类似于解方程，利用我们的符号执行中得到的式子来进行计算

- SAT问题（The Satisfiability Problem，可满足性问题）
- SMT（Satisfiability Module Theories，可满足性模理论）

6.5 符号执行方法分类

- 静态符号执行本身不会实际执行程序，通过解析程序和符号值模拟执行，有代价小、效率高的优点，但是存在路径爆炸、误报高的情况
- 动态符号执行也称为混合符号执行，它的基本思想是：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，根据路径搜索策略（深度、广度）对约束表达式进行变形，求解变形后的表达式并生成新的测试用例，不断迭代上面的过程，直到完全遍历程序的所有执行路径。
- 选择性符号执行可以对程序员感兴趣的部分进行符号执行，其它的部分使用真实值执行，在特定任务环境下可以进一步提升执行效率

7 Z3约束求解器

——SMT问题的开源约束求解器，就是自动解方程组

一般使用的方法：

- 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

8 Angr应用示例

见实验

变量符号化——动态符号执行——获取路径约束条件——约束求解

9 污点分析基本原理

- 污点分析标记程序中的数据（外部输入数据或者内部数据）为污点，通过对带污点数据的传播分析来达到保护数据完整性和保密性的目的
- 如果信息从被标记的污点数据传播给未标记的数据,那么需要将未标记的标记为污点数据；如果被标记的污点数据传递到重要数据区域或者信息泄露点，那就意味着信息流策略被违反

污点分析可以抽象成一个三元组（**sources，sinks，sanitizers**）的形式：

- source即污点源，代表直接引入不受信任的数据或者机密数据到系统中；
- sink即污点汇聚点，代表直接产生安全敏感操作(违反数据完整性)或者泄露隐私数据到外界(违反数据保密性)；
- sanitizer即无害处理，代表通过数据加密或者移除危害操作等手段使数据传播不再对软件系统的信息安全产生危害。

污点分析就是分析程序中由污点源引入的数据是否能够不经**无害处理**，而直接传播到污点汇聚点。如果不能，说明系统是信息流安全的；否则，说明系统产生了隐私数据泄露或危险数据操作等安全问题

分为识别污点源和汇聚点、污点传播分析和无害处理三个步骤

识别污点源和污点汇聚点是污点分析的前提

1.识别污点源：

- 使用启发式的策略进行标记，例如把来自程序外部输入的数据统称为“污点”数据，保守地认为这些数据有可能包含恶意的攻击数据；
- 根据具体应用程序调用的API或者重要的数据类型，手工标记源和汇聚；

- 使用统计或机器学习技术自动地识别和标记污点源及汇聚点。

2.污点传播分析：

分类：**显式流分析和隐式流分析**

- 显式流分析：分析污点标记如何随程序中变量之间的数据依赖关系传播。
- 隐式流分析：分析污点标记如何随程序中变量之间的控制依赖关系传播，也就是分析污点标记如何从条件指令传播到其所控制的语句

隐式流的两个问题：

- **欠污染：**由于对隐式流污点传播处理不当导致**本应被标记的变量没有被标记的问题**称为欠污染(under-taint)问题。
- **过污染：**由于污点**标记的数量过多而导致污点变量大量扩散的问题**称为过污染(over-taint)问题。

3.无害处理：

- 常数赋值是最直观的无害处理的方式；
- 加密处理、程序验证等在一定程度上，可以认为是无害处理。

10 污点分析方法

10.1 显式流分析

静态分析：在不运行且不修改代码的前提下，通过分析程序变量间的**数据依赖**关系来检测数据能否从污点源传播到污点汇聚点

常见的显式流污点传播方式包括**直接赋值传播、通过函数(过程)调用传播以及通过别名(指针)**传播。

动态分析：在程序运行过程中，通过实时监控程序的污点数据在系统程序中的传播来检测数据能否从污点源传播到污点汇聚点。

分为三类：

- **基于硬件的污点传播分析**
- **基于软件的污点传播分析**
- **混合型的污点分析**

10.2 隐式流分析

静态隐式流分析：**需要分析每一个分支控制条件是否需要传播污点标记**。路径敏感的数据流分析往往会产生路径爆炸问题，导致开销难以接受

精度与效率不可兼得

动态隐式流分析:

- 如何确定污点控制条件下需要标记的语句的范围？动态执行轨迹并不能反映出被执行的指令之间的控制依赖关系
- 由于部分泄漏导致的漏报如何解决？指污点信息通过动态未执行部分进行传播并泄漏
- 如何选择合适的污点标记分支进行污点传播？鉴于单纯地将所有包含污点标记的分支进行传播会导致过污染的情况