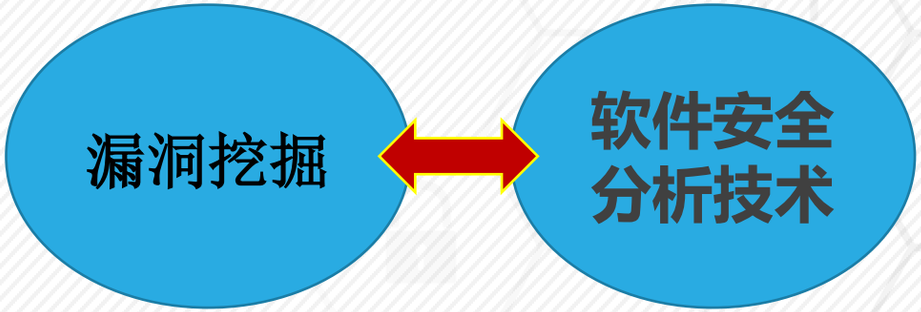
第七章　漏洞挖掘

知识点一：方法概述
知识点二：词法分析
知识点三：数据流分析
知识点四：模糊测试
知识点五：AFL模糊测试框架

知识点一：方法概述

1. 漏洞挖掘方法分类



静态分析技术

方法：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

方法：模糊测试、动态污点分析、动态符号执行等

需要运行程序、准确率非常高，误报率很低

符号执行和污点分析两类技术都分别支持静态分析和动态分析

2. 符号执行

符号执行（Symbolic Execution）的基本思路是使用符号值替代具体值，模拟程序的执行。在模拟程序运行的过程中，符号执行引擎会收集程序中的语义信息，探索程序中的可达路径、分析程序中隐藏的错误。

动态符号执行结合了真实执行和传统符号执行技术的优点，在真实执行的过程中同时进行符号执行，可以在保证测试精度的前提下提升了执行效率。

符号执行基本原理

符号执行三个关键点是变量符号化、程序执行模拟和约束求解。

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**。

符号执行基本原理

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。

符号执行有代价小、效率高的优点，然而由于程序执行的可能路径随着程序规模的增大呈指数级增长，从而导致符号执行技术在分析输入和输出之间关系时，存在一个路径状态空间的路径爆炸问题。由于符号执行技术进行路径敏感的遍历式检测，当程序执行路径的数量超过约束求解工具的求解能力时，符号执行技术将难以分析。

符号执行的应用

符号执行已经广泛应用在软件测试、漏洞挖掘和软件破解等。

- **在软件测试中**，符号执行可以获得程序执行路径的集合、路径的约束条件和输出的符号表达式，可以使用约束求解器求解出满足约束条件的各个路径的输入值，用于创建高覆盖率的测试用例。符号执行与模糊测试的结合也是当前流行的软件测试技术。
- **在漏洞挖掘中**，通过符号执行技术可以获得漏洞监测点的变量符号表达式，结合路径约束条件、变量符号表达式和漏洞分析规则，可以通过约束求解的方法来求解是否存在满足或违反漏洞分析规则的值。
- 符号执行还可以用于搜索特定目标代码的到达路径，进而计算该路径的输入，用在面向特定任务（比如代码破解）的程序分析中。

举例（漏洞挖掘-检测是否数组越界）：

```
int a[10];

scanf("%d", &i);

if (i > 0) {

    if (i > 10)

        i = i % 10;

    a[i] = 1;

}
```

- 在a[i]=1语句处存在可能数组越界的情况

  访问越界的约束条件是x>=10

- 整段代码存在两个if分支，经过符号执行，知道到达a[i]=1语句处有2条路径：

  ① 路径约束条件为x>0∧x<=10，此时变量i的符号表达式为x；

  ② 路径约束条件为x>0∧x>10，此时变量i的符号表达式为x%10。

- 得到两个判定条件：

① (x>0∧x<=10)∧(x>=10) **--有解 x=10 满足越界条件 存在漏洞**

② (x>0∧x>10)∧x%10>=10)

3. 污点分析

污点分析（Taint Analysis）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。

污点分析可以分为污点传播分析（静态分析）和动态污点分析（动态分析）。静态污点分析技术在检测时并不真正运行程序，而是通过模拟程序的执行过程来传播污点标记，而动态污点分析技术需要运行程序，同时实时传播并检测污点标记。

基本思想

首先，确定污点源，即污点分析的目标来源。通常来讲，污点源表示了程序外部数据或者用户所关心的程序内部数据，是需要进行标记分析的输入数据。

然后，标记和分析污点。对污点源在内存中进行标记、计算涉及到污点的执行过程。

```
1. X=get_input_from_file()
2. Y=get_input_from_network()
……
3. S=X[0]
4. T=S-Y[0]
5. S=0
……
6. goto T
```

污点源1：来自文件

污点源2：来自网络

污点传播1：扩散规则

污点传播2：扩散规则

污点传播3：清除规则

污点检测1：控制流劫持

第六行，验证程序的执行流程将被外部数据任意控制，发生了典型的“**控制流劫持**”。

污点分析核心要素

以上实例介绍了污点分析方法的基本原理，可以看出有如下三个核心要素：

- **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

污点分析的核心是分析输入参数和执行路径之间的关系，它**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。**

知识点二：词法分析

1. 基本概念

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。**

主要思想是将代码文本与归纳好的缺陷模式（比如边界条件检查）进行匹配，以此发现漏洞。

优点：算法**简单**，检测**性能较高**

缺点：只能进行**表面的词法检测**，不能进行语义方面的深层次分析，因此可以检测的安全缺陷和漏洞较少，会出现**较高的漏报和误报**，尤其对于高危漏洞无法进行有效检测。

实践1 基于词法分析和逆向分析的可执行代码静态检测

核心思想：根据二进制可执行文件的格式特征，**从二进制文件的头部、符号表以及调试信息中提取安全敏感信息（识别危险函数）**，来分析文件中是否存在安全缺陷。

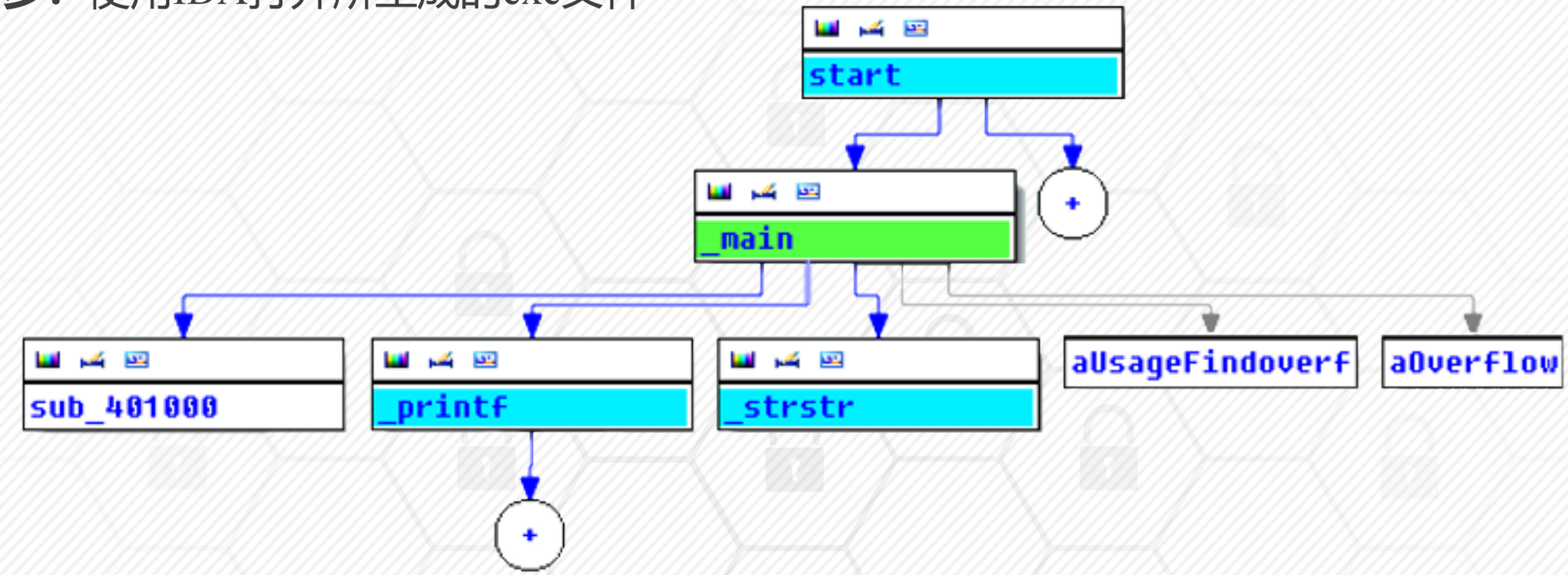
- **第一步** → 找到敏感函数，比如memcpy、strcpy等
- **第二步** → 回溯函数的参数
- **第三步** → 判断栈与操作参数的大小关系，以定位是否发生了溢出漏洞

实验一：基于IDA Pro分析给定的可执行文件是否存在溢出漏洞

对于findoverflow.exe，是通过vc6代码生成的**Release版本**：

```
#include <stdio.h>
#include <string.h>
void makeoverflow(char *b){
        char des[5];
        strcpy(des,b);
}
void main(int argc,char *argv[]){
        if(argc>1)              {
            if(strstr(argv[1],"overflow")!=0)
                makeoverflow(argv[1]);
        }   else
            printf("usage: findoverflow XXXXX\n");
}
```

第一步：使用IDA打开所生成的exe文件



通过该视图，可见，主要有一个main函数，在该函数中可能有跳转，调用了sub_401000函数、_strstr函数和_printf函数。此外，还定义了两个字符串常量，aUsageFindoverf，在其上点右键->Text view，可以看到：

```
.data:00406030                          ; char aUsageFindoverf[]
.data:00406030 75 73 61 67 65 3A 20 66+aUsageFindoverf db 'usage: findoverflow XXXXX',0Ah,0
```

打开main函数汇编代码如下：

```
; int __cdecl main(int argc, const char **argv, const char **envp)
_main proc near

argc= dword ptr  4
argv= dword ptr  8
envp= dword ptr  0Ch

mov     eax, [esp+argc]
push    esi
cmp     eax, 1
jle     short loc_401061
```

[ ]的作用是什么？
是什么方式寻址？

为何要加4？之前为什么要将esi入栈？

```
mov     esi, [esp+4+argv]
push    offset aOverflow ; "overflow"
mov     eax, [esi+4]
push    eax             ; char *
call    _strstr
add     esp, 8
test    eax, eax
jz      short loc_40106E
```

```
loc_401061:                     ; "usage: findoverflow XXXXX\n"
push    offset aUsageFindoverf
call    _printf
add     esp, 4
```

ESI已经是argV的值，指针+4，相当于argV[1]的地址

```
mov     ecx, [esi+4]
push    ecx
call    sub_401000
```

```
loc_40106E:
pop     esi
```

注意

通常在IDA的反汇编中，**arg_x表示函数参数x的位置，var_8表示局部变量的位置；[]是内存寻址，[x+arg_x]通常表达的就是arg_x的地址值。**

由release和debug生成的汇编代码是截然不同的，release版本非常简洁，执行效率优先，debug版本则基本严格按照语法结构，而且增加了很多方便调试的附加信息。

```
sub_401000 proc near

var_8= byte ptr -8
arg_0= dword ptr  4

sub     esp, 8
or      ecx, 0FFFFFFFFh
xor     eax, eax
lea     edx, [esp+8+var_8]
push    esi
push    edi
mov     edi, [esp+10h+arg_0]
repne scasb
not     ecx
sub     edi, ecx
mov     eax, ecx
mov     esi, edi
mov     edi, edx
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
pop     edi
pop     esi
add     esp, 8
retn
sub_401000 endp
```

第二步：定位敏感函数

在主函数中，Printf函数无任何格式化参数存在。因此，存在敏感函数的可能在于sub_401000函数中，打开该函数的代码如左图：

- 一个输入参数arg_0，一个局部变量var_8。
- 通过“lea edx, [esp+8+var_8]”和“mov edi, edx”可知，向目标寄存器存储了目标字符串的地址，为局部变量var_8；
- 通过“mov edi,[esp+10h+arg_0]”以及后面的“mov esi, edi”，可知，将函数的输入参数作为源字符串。
- **那么到底是否发生了溢出呢?**

通过“sub esp 8”可以知道栈大小为8，因此，函数的局部变量var_8的大小最大就是8。这样的话，可以得到sub_401000函数的代码结构大致如下：

```
Sub_401000(arg_0)
{
    Char var_8[8];
    Strcpy(var_8, arg_0);
}
```

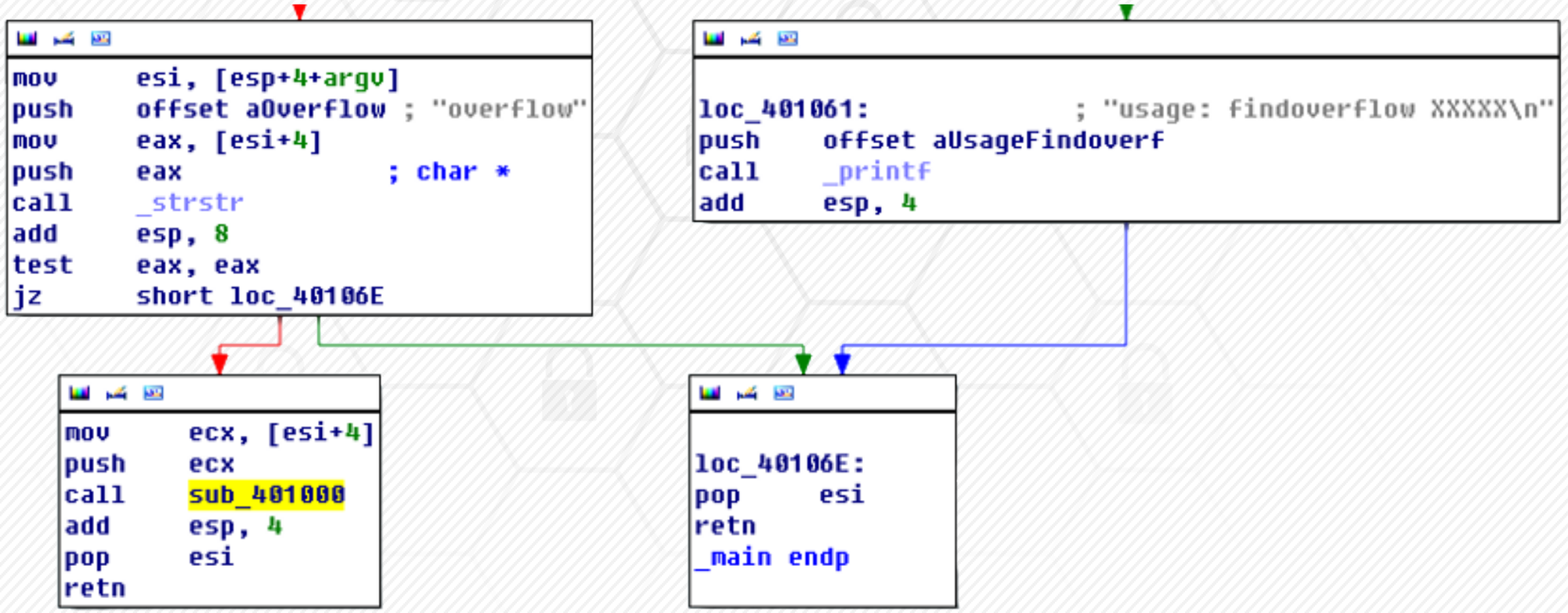
如果输入的字符串的长度大于8，就可能发生溢出了——**需要验证。**

为什么是8，二不是源代码里的5?

打开DOS对话框，运行示例程序，如果不给任何参数的话，会提示：usage: findoverflow XXXXX

如果输入参数，比如：findoverflow sssssssss。却可以运行成功。

这是为什么呢？回顾逆向的反汇编代码，可以知道：



由于程序需要先判断是否包含子串overflow，因此，需要构造的输入需要满足这个条件。

输入：findoverflow overflow，此时出现缓冲区溢出的弹出窗口了。

基于此溢出漏洞，就可以进行漏洞的利用了。

实验二：使用Bugscam脚本来替代手工过程完成漏洞挖掘

Bugscam是一个IDA工具的idc脚本的轻量级的漏洞分析工具，通过检测栈溢出漏洞的诸如strcpy,sprintf危险函数的位置，然后根据这些函数的参数，确定是否有缓冲区漏洞。下载网址：https://sourceforge.net/projects/bugscam/

1、将Bugscam文件解压放到任意地方，然后修改globalvar.idc文件中头行的bugscam_dir为你的bugscam**目录的全路径**（路径不能含有中文）。

```
#include <stdio.h>
#include <windows.h>
void vul(char*bu1){
        char a[200];
        lstrcpy(a,bu1);
            printf("%s",a);
        return;    }
void main(){
        char b[1024];
        memset(b,'l',sizeof(b));
        vul(b);      }
```

实验二：使用Bugscam脚本来替代手工过程完成漏洞挖掘

2、启动ida，**加载任意一个x86程序文件（本例为idc.exe），然后打开脚本文件run_analysis.idc**，运行即可，等待分析完毕，最后的分析报告结果保存在reports目录中的html文件中。

Results for lstrcpyA

The following table summarizes the results of the analysis of calls to the function lstrcpyA.

Address	Severity	Description
401010	8	The maximum possible size of the target buffer (203) is smaller than the minimum possible size of the source buffer (1024). This is VERY likely to be a buffer overrun!
401010	8	The maximum possible size of the target buffer (203) is smaller than the minimum possible size of the source buffer (1024). This is VERY likely to be a buffer overrun!

其中，Severity是威胁等级，越高说明漏洞危险级别越高。本例的程序中，lstrcpyA函数存在溢出漏洞，地址401010处的代码可能将向目标203字节的区域写入1024字节的数据。

知识点三：数据流分析

1. 基本概念

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。

按照分析程序路径的深度，将数据流分析分为**过程内分析**和**过程间分析**

数据流分析方法分类

□过程内分析只针对程序中函数内的代码进行分析，又分为：

- ✓ 流不敏感分析（flow insensitive）：按代码行号从上而下进行分析；
- ✓ 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；
- ✓ 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。

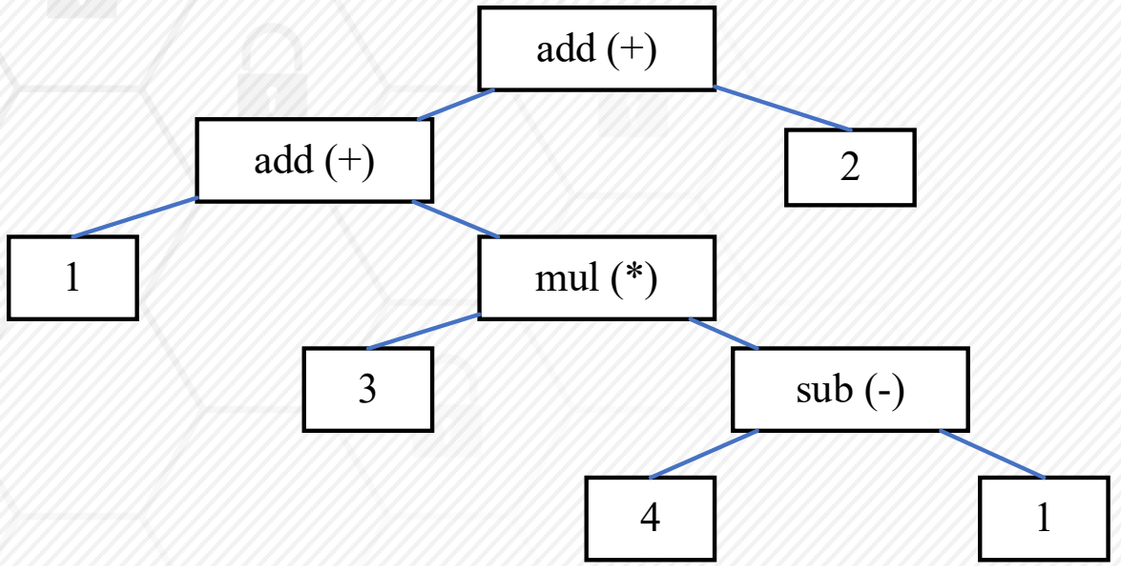
□过程间分析则考虑函数之间的数据流，即需要跟踪分析目标数据在函数之间的传递过程。

- ✓ 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。
- ✓ 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

数据流分析使用的程序代码模型主要包括**程序代码的中间表示**以及一些**关键的数据结构**，利用程序代码的中间表示可以对程序语句的指令语义进行分析。

抽象语法树。是程序抽象语法结构的树状表现形式，其每个内部节点代表一个运算符，该节点的子节点代表这个运算符的运算分量。通过描述控制转移语句的语法结构，抽象语法树在一定程度上也描述了程序的过程内代码的控制流结构。



举例，对于表达式“1+3*(4-1)+2”，其抽象语法树为：

程序代码模型

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。

通常的三地址码指令包括下面几种：

- ✓ 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)

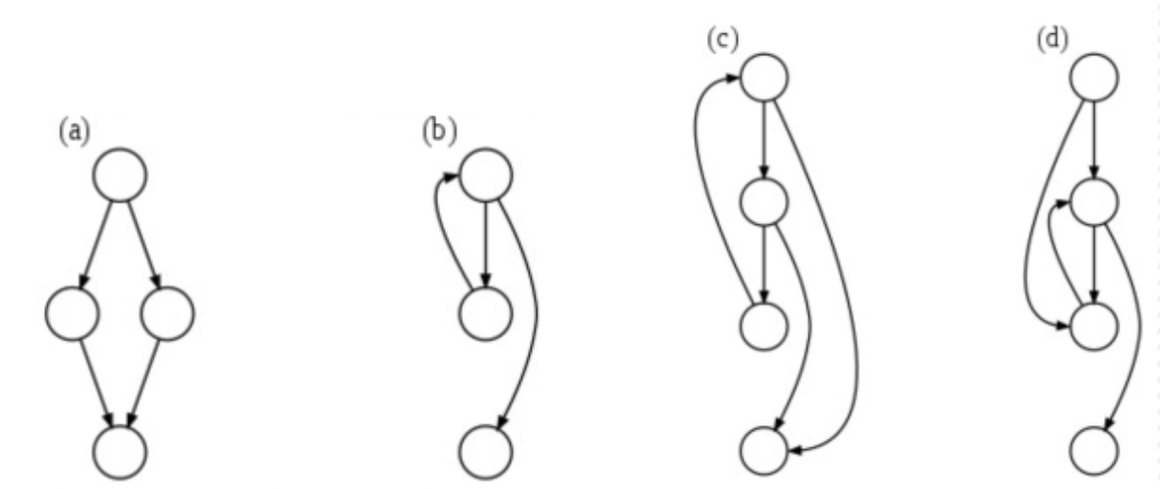
```
for (i = 0; i < 10; ++i) {
    b[i] = i*i;
}
```

```
    t1 := 0          ; initialize i
L1: if t1 >= 10 goto L2  ; conditional jump
    t2 := t1 * t1        ; square of i
    t3 := t1 * 4         ; word-align address
    t4 := b + t3         ; address to store i*i
    *t4 := t2            ; store through pointer
    t1 := t1 + 1         ; increase i
    goto L1              ; repeat loop
L2:
```

程序代码模型

控制流图。控制流图（Control Flow Graph，CFG）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。

(a)有一个if-then-else语句；(b)有一个while循环;(c)有两个出口的自然环路;(d)有两个入口的循环。



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。

2. 基于数据流的漏洞分析流程

基于数据流的漏洞分析技术是通过分析软件代码中变量的取值变化和语句的执行情况，来分析数据处理逻辑和程序的控制流关系，从而分析软件代码的潜在安全缺陷。

基于数据流的漏洞分析的一般流程为：

- 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；
- 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。

基于数据流的漏洞分析非常适合检查因控制流信息非法操作而导致的安全问题，如内存访问越界、常数传播等。由于对于逻辑复杂的软件代码，其数据流复杂，并呈现多样性的特点，因而检测的准确率较低，误报率较高。

示例一：检测指针变量的错误使用

```
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
}
```

在检测指针变量的错误使用时，我们关心的是变量的状态。左侧代码可能出现use-after-free漏洞。

漏洞分析规则。下面是用于检测指针变量错误使用的检测规则：

v 被分配空间 ==> v.start

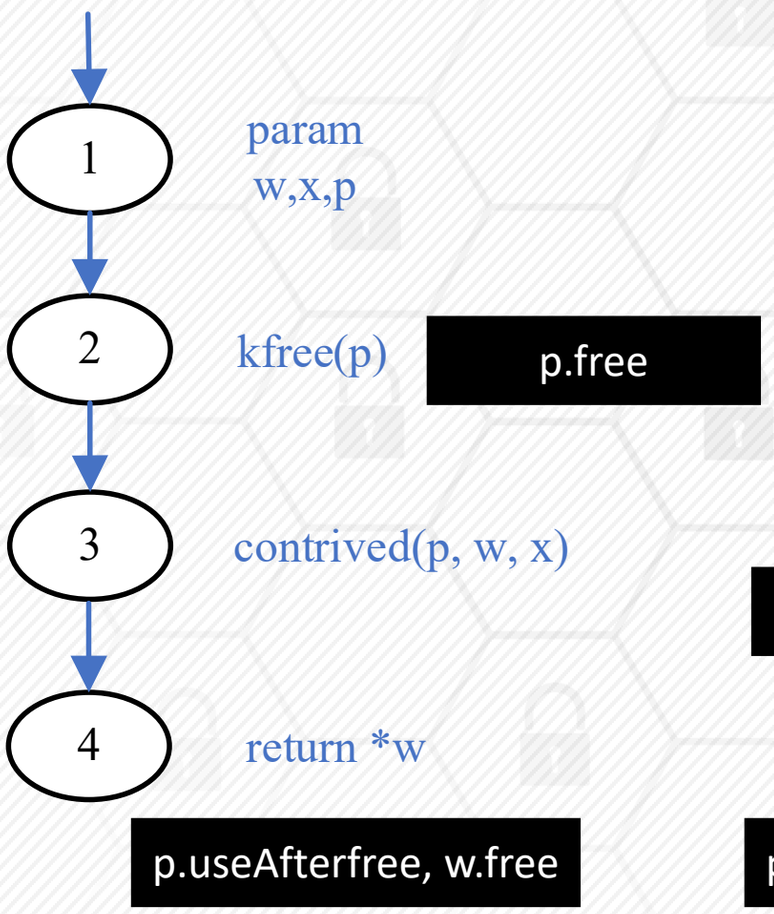
v.start: {kfree(v)} ==> v.free

v.free: {*v} ==> v.useAfterFree

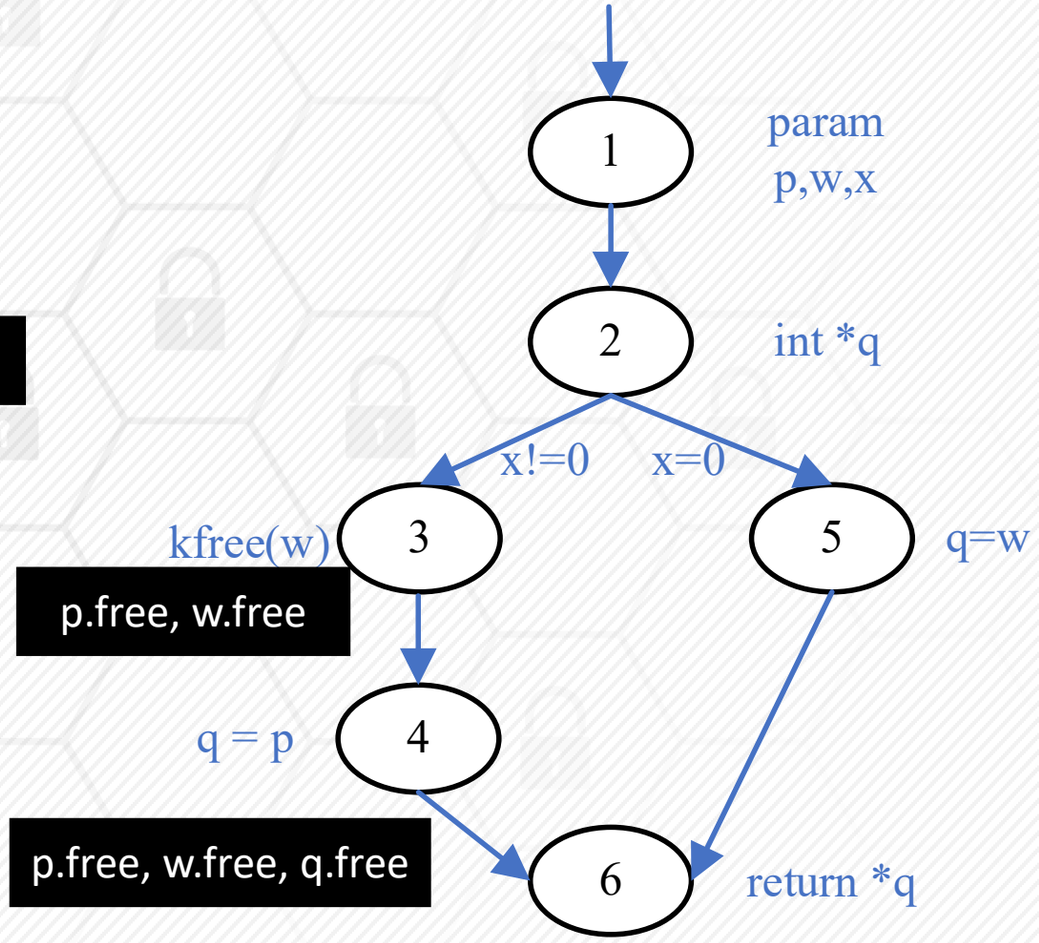
v.free: {kfree(v)} ==> v.doubleFree

示例一：检测指针变量的错误使用

代码建模。这里我们采用路径敏感的数据流分析，控制流图如下



contrived_caller



contrived

示例一：检测指针变量的错误使用

漏洞分析。分析过程从函数contrived_caller的入口点开始，可知调用函数contrived的时候p的状态为p.free。分析函数contrived中的两条路径：

- **1->2->3->4->6**：在进行到6时，6的前置条件是p.free、w.free、q.free，此时语句return *q将触发use-after-free规则并设置q.useAfterFree状态。然后返回到函数contrived_caller的4，其前置条件为p.useAfterFree、w.free，此时语句return *w设置w.useAfterFree。因此，存在use-after-free漏洞。
- **1->2->5->6**：该路径是安全的。

示例二：检测缓冲区溢出

在检测缓冲区溢出时，我们**关心的是变量的取值**，并在一些预定义的敏感操作所在的程序点上，**对变量的取值进行检查**。下面是一些记录变量的取值的规则。

```
char s[n];                  // len(s) = n
strcpy(des, src);           // len(des) > len(src)
strncpy(des, src, n);       // len(des) > min(len(src), n)
s = "foo";                  // len(s) = 4
strcat(s, suffix);          // len(s) = len(s) + len(suffix) - 1
fgets(s, n, ...);           // len(s) > n
```

知识点四：模糊测试

1. 模糊测试

1

模糊测试(Fuzzing)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。**

2

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。

3

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。**

模糊测试分类

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。

模糊测试步骤

（1）确定测试对象和输入数据

由于所有可被利用的漏洞都是由于应用程序接受了用户输入的数据造成的，并且在处理输入数据时没有首先过滤非法数据或者进行校验确认。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。

模糊测试步骤

（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。**

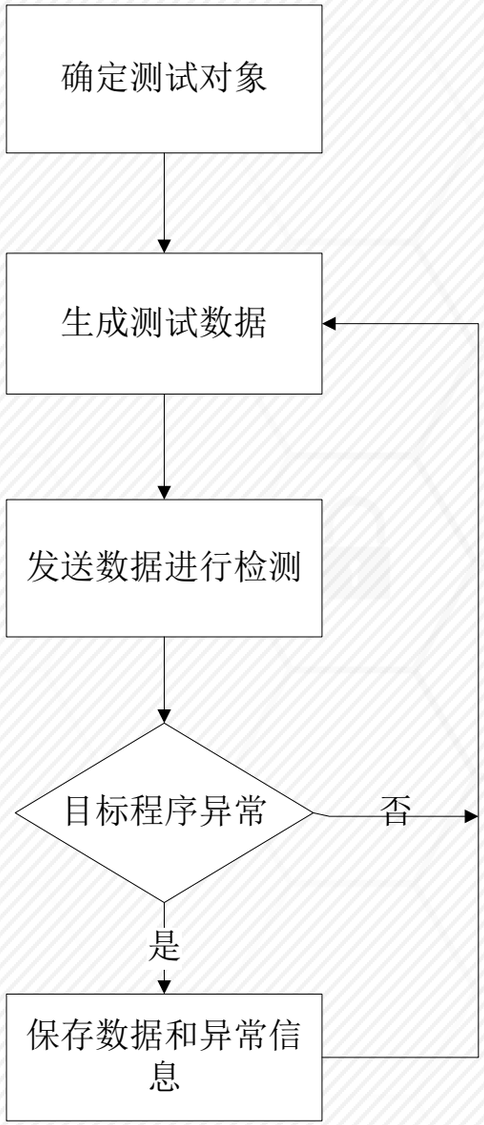
（4）监测程序异常

在模糊测试过程中，一个非常重要但却经常被忽视的步骤是对程序异常的监测。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。

模糊测试步骤

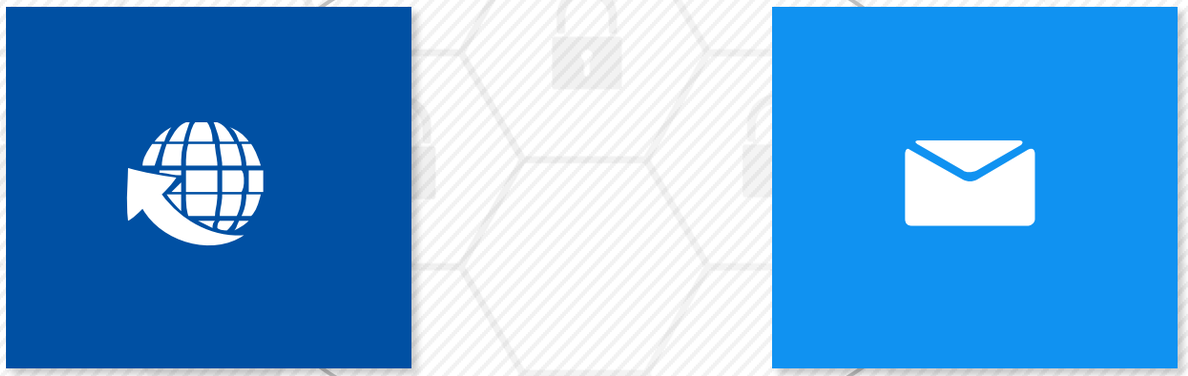
（5）确定可利用性

一旦监测到程序出现的异常，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**。这个步骤不是模糊测试必需的步骤，只是检测这个异常对应的漏洞是否可以被利用。这个步骤**一般由手工完成**，需要分析人员具备深厚的漏洞挖掘和分析经验。



所有类型的模糊测试技术

除了最后一步确定可利用性外，所有其它的四个阶段都是必须的。上述典型的fuzzing测试流程如左图所示。



尽管模糊测试对安全缺陷和漏洞的检测能力很强，但**并不是说它对被测软件都能发现所有的错误**，原因就是它测试样本的生成方式具有**随机性。**

2. 智能模糊测试

- 模糊测试方法是应用最普遍的动态安全检测方法，但由于**模糊测试数据的生成具有随机性，缺乏对程序的理解**，测试的性能不高，并且难以保证一定的覆盖率。
- 为了解决这个问题，**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。

智能模糊测试具体的实现步骤如下

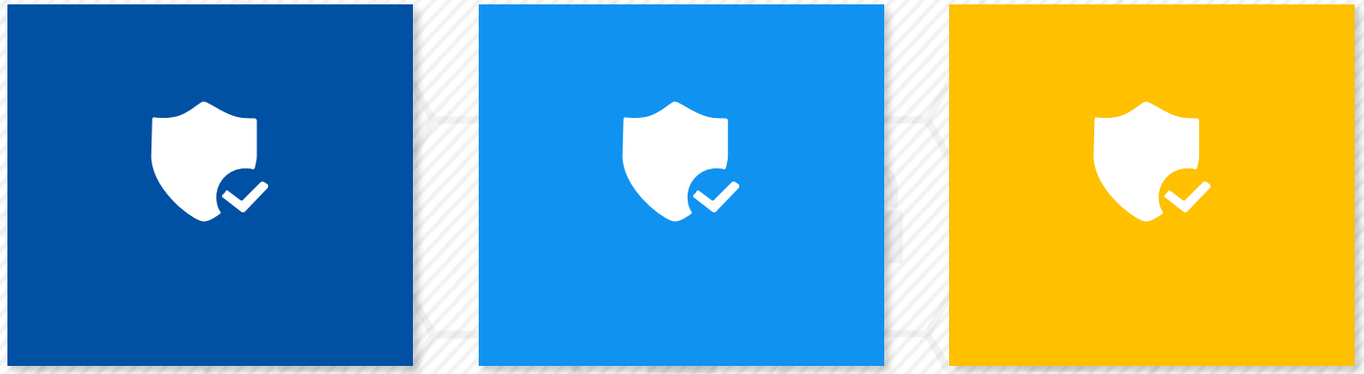
（1）反汇编

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

从汇编代码中直接获取程序运行的内部信息，工作量较大，为此，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



这一步是智能模糊测试的关键，它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。**例如，利用符号执行技术在符号执行过程中记录下输入数据的传播过程和传播后的表达形式，并通过约束求解得到在漏洞触发时执行的路径与原始输入数据之间的联系，从而得到触发执行路径异常的输入数据。

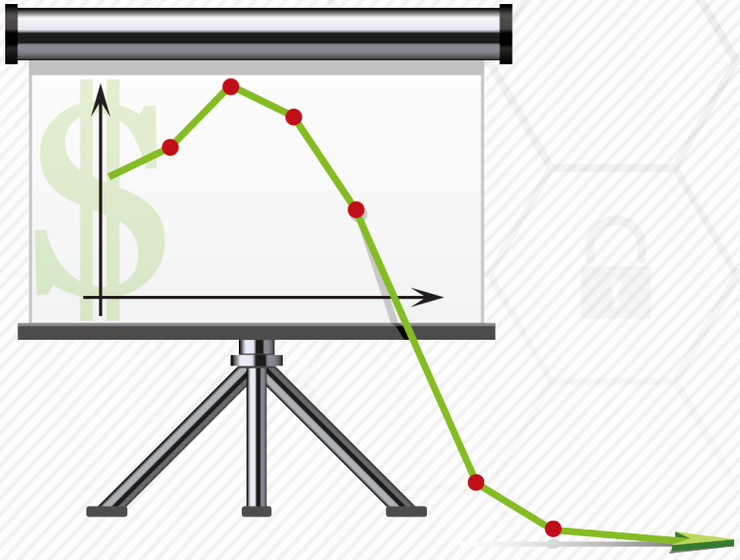
（4）利用分析获得的输入数据集合，对执行路径集合进行测试

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。**与传统的随机模糊测试技术相比，这些**智能模糊测试技术**的应用，由于了解了输入数据和执行路径之间的关系，因而**生成的输入数据更有针对性，减少了大量无关测试数据的生成，提高了测试的效率。**此外，在触发漏洞的同时，智能模糊测试技术包含了对漏洞成因的分析，极大减少了分析人员的工作量。

智能模糊测试的核心思想

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。

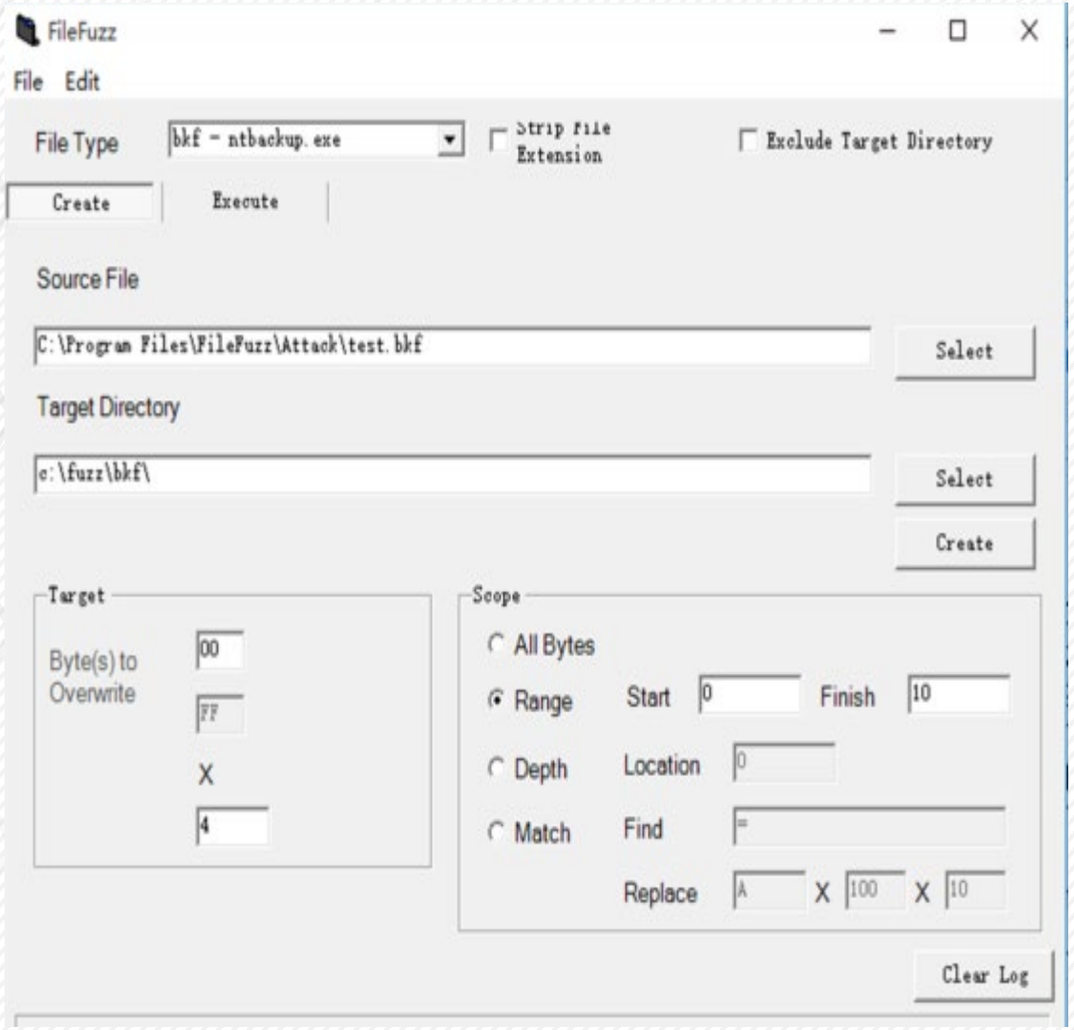
智能模糊测试技术的提出，反映了软件安全性测试**由模糊化测试向精确化测试**转变的趋势。是典型的技术融合的漏洞挖掘测试方法。



3. 模糊测试实践

（1）使用工具

- **用来实现Fuzzing测试的工具叫做Fuzzer。**
- 成品的Fuzzer工具很多，许多是非常优秀的。Fuzzer根据测试类型可以分为很多类，常见的分类包括：**文件型Fuzzer、网络型Fuzzer、接口型Fuzzer**等。
- 右侧工具可以生成多个文件测试用例，发现了Office2003的典型漏洞。



（2）自己动手写Fuzzer

使用模糊测试工具在很多时候不能解决所有问题

比如：被测试的目标程序对测试数据有一定的要求，而实际的Fuzzer不能灵活调整发送的测试数据；被测试的目标程序过于简单或者难，而现有的Fuzzer程序不能提供适合的测试。

作为漏洞发掘者我们最好能学会编写一个Fuzzer，这样就可以随时随地的进行安全测试。而事实上，**目前的多数漏洞挖掘过程，是需要自己手动编写Fuzzer来完成。**

对于目标的可执行文件overflow.exe文件，是由如下程序生成的exe程序：

```
#include <stdio.h>
#include <string.h>
void overflow(char *b){
        char des[50];
        strcpy(des,b);
}
void main(int argc,char *argv[]){
        if(argc>1)      {
            overflow(argv[1]);
        }   else
                printf("usage: overflow XXXXX\n");
}
```

书写Fuzzer

在明确了输入的要求和暴力测试的循环条件后，可以写出如下的代码：

```
void main(int argc,char *argv[]){
        char *testbuf=" ";       char buf[1024];
        memset(buf,0,1024);
        if(argc>1) {
                for(int i=20;i<50;i=i+2) {
                 testbuf=new char[i];
                 memset(testbuf,'c',i);
                 memcpy(buf,testbuf,i);
        ShellExecute(NULL,"open",argv[1],buf,NULL,SW_NORMAL);
                delete testbuf;}
        }
        else printf("Fuzzing X \n其中X为被测试目标程序所在路径");
    }
```

以上代码

通过一个for循环（循环次数根据实际情况去设计）构造不同的字符串作为输入，
通过“ShellExecute(NULL,"open",argv[1],buf,NULL,SW_NORMAL); “
来实现对目标程序的模糊测试。
上述Fuzzer的调用格式为：Fuzzing X 。X表示目标程序。
请完成上述实验并进行结果验证。

知识点五：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

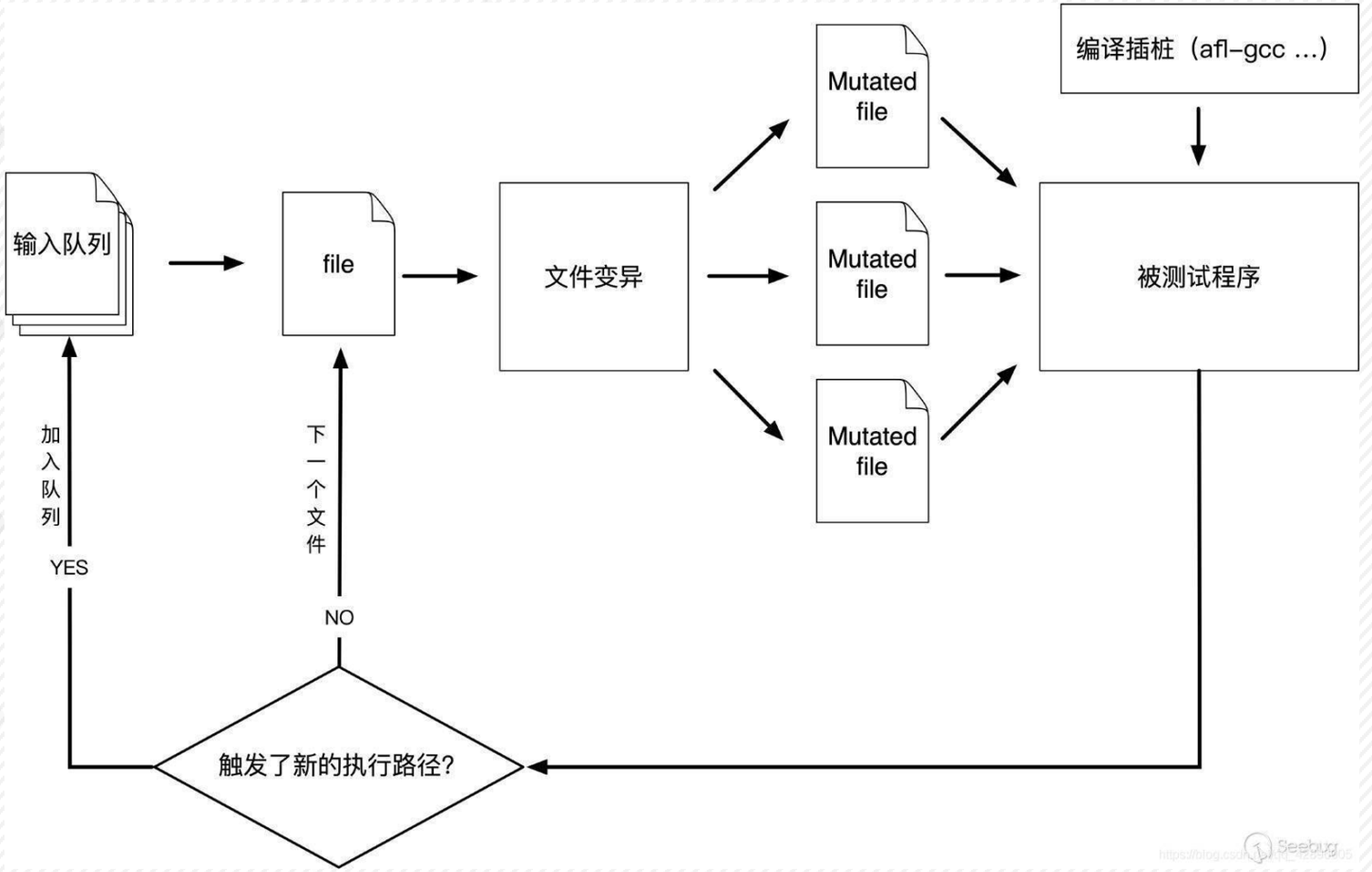
AFL**主要用于C/C++程序的测试**，被测程序**有无程序源码均可**，有源码时可以对源码进行编译时插桩，无源码可以借助QEMU的User-Mode模式进行二进制插装。

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（异常结果）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: 0000000000001628     0 NOTYPE  LOCAL  DEFAULT   14 __afl_maybe_log
    37: 00000000000040b0     8 OBJECT  LOCAL  DEFAULT   25 __afl_area_ptr
    38: 0000000000001660     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup
    39: 0000000000001638     0 NOTYPE  LOCAL  DEFAULT   14 __afl_store
    40: 00000000000040b8     8 OBJECT  LOCAL  DEFAULT   25 __afl_prev_loc
    41: 0000000000001655     0 NOTYPE  LOCAL  DEFAULT   14 __afl_return
    42: 00000000000040c8     1 OBJECT  LOCAL  DEFAULT   25 __afl_setup_failure
    43: 0000000000001681     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup_first
    45: 0000000000001949     0 NOTYPE  LOCAL  DEFAULT   14 __afl_setup_abort
    46: 000000000000179e     0 NOTYPE  LOCAL  DEFAULT   14 __afl_forkserver
    47: 00000000000040c4     4 OBJECT  LOCAL  DEFAULT   25 __afl_temp
    48: 000000000000185c     0 NOTYPE  LOCAL  DEFAULT   14 __afl_fork_resume
    49: 00000000000017c4     0 NOTYPE  LOCAL  DEFAULT   14 __afl_fork_wait_loop
    50: 0000000000001941     0 NOTYPE  LOCAL  DEFAULT   14 __afl_die
    51: 00000000000040c0     4 OBJECT  LOCAL  DEFAULT   25 __afl_fork_pid
    98: 00000000000040d0     8 OBJECT  GLOBAL DEFAULT   25 __afl_global_area_ptr
```

（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

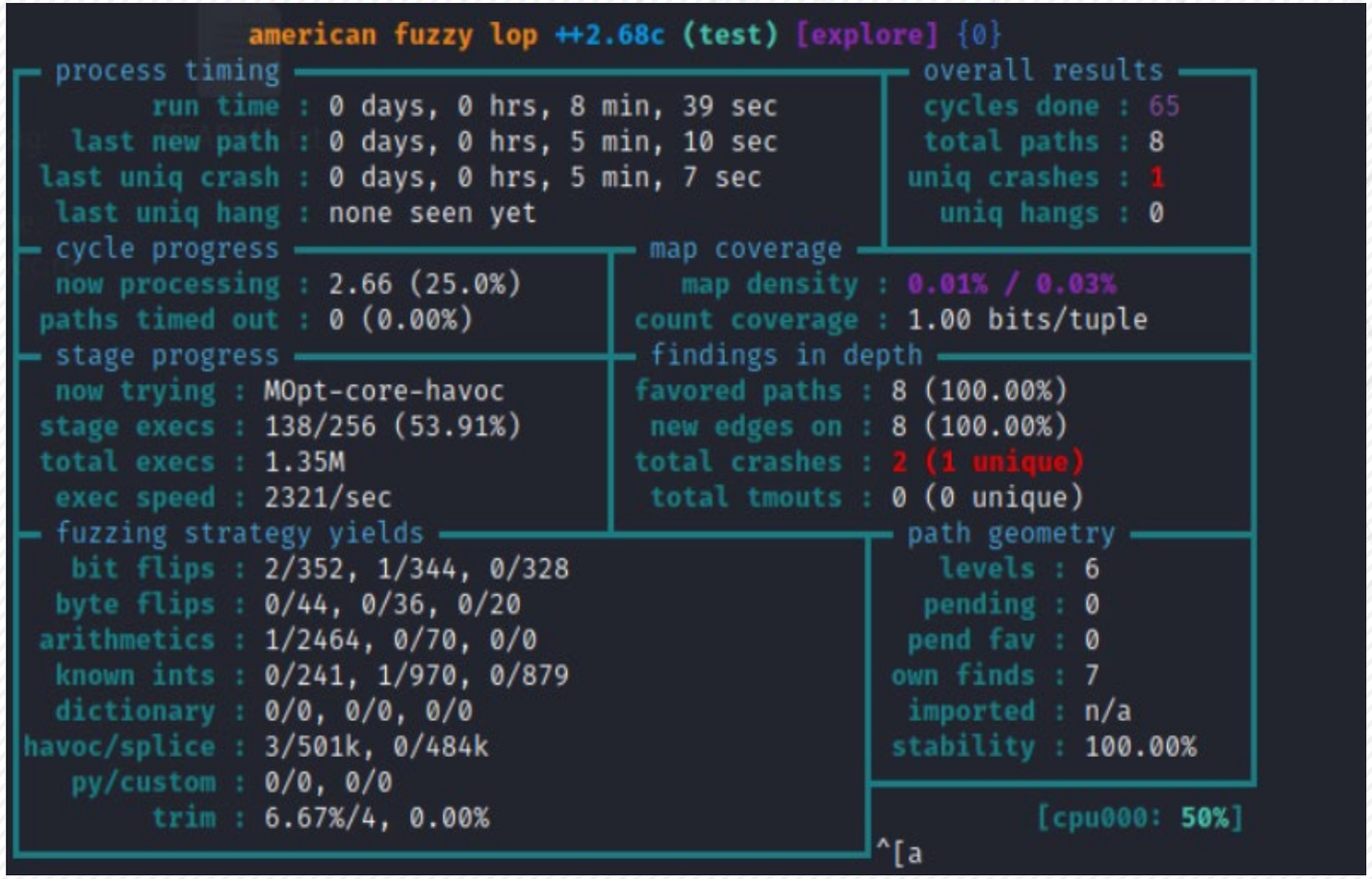
运行如下命令，开始启动模糊测试（**@@表示目标程序需要从文件读取输入**）：

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。



- 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。