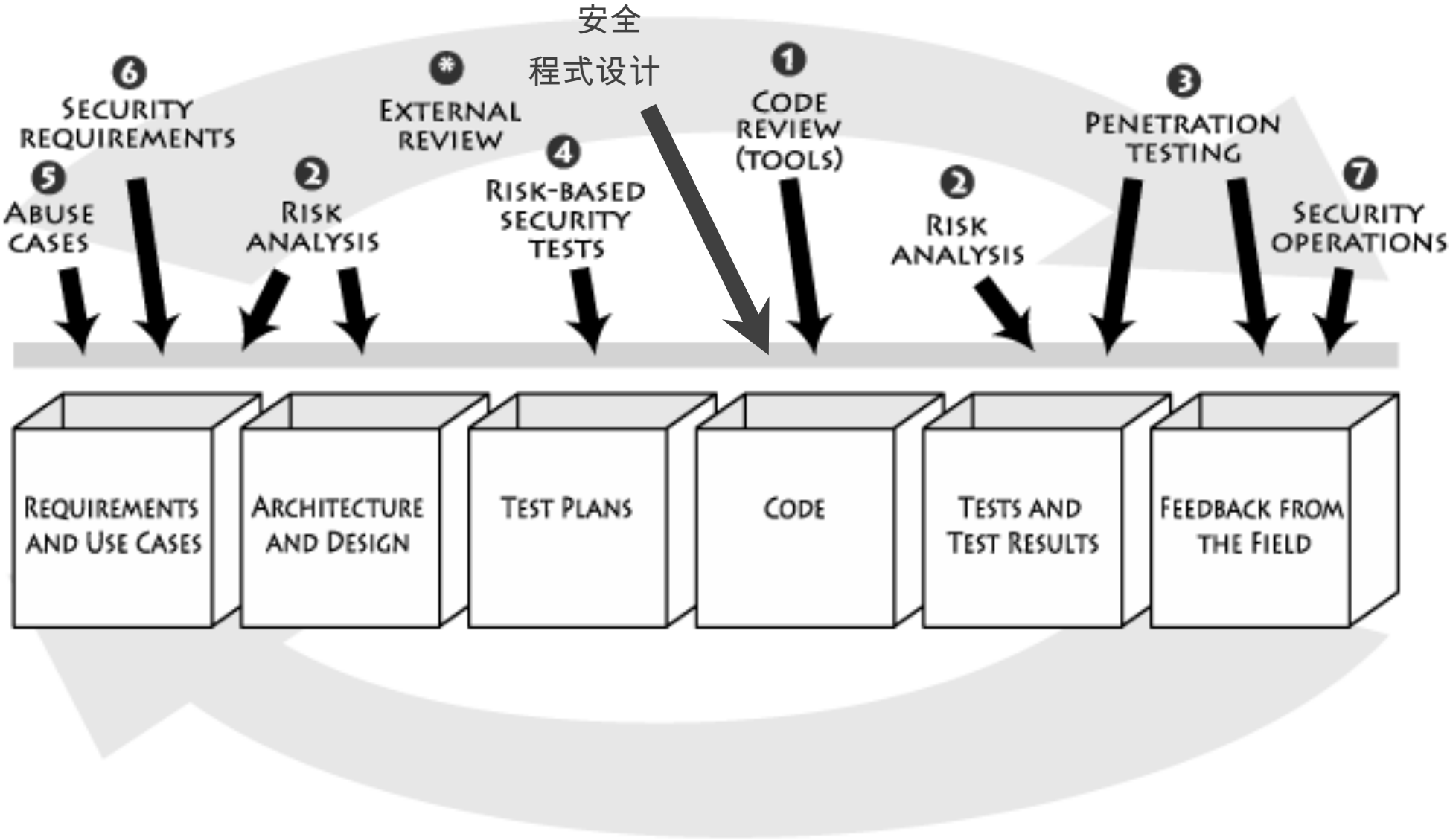
安全软件开发

生命周期

安全SDLC


安全
程式设计
6 SECURITY REQUIREMENTS
5 ABUSE CASES
2 RISK ANALYSIS
* EXTERNAL REVIEW
4 RISK-BASED SECURITY TESTS
1 CODE REVIEW (TOOLS)
2 RISK ANALYSIS
3 PENETRATION TESTING
7 SECURITY OPERATIONS
REQUIREMENTS AND USE CASES
ARCHITECTURE AND DESIGN
TEST PLANS
CODE
TESTS AND TEST RESULTS
FEEDBACK FROM THE FIELD

安全需求工程：两个方法

- 认证安全
  - 最佳做法，安全准则
  - 信息流控制
  - 静态和动态分析

安全编程指南

代码扫描器/静态分析/代码审核

安全测试

通用标准/ EAL

- 设计安全
  - 安全目标
  - 威胁分析

安全架构

安全性

访问控制/密码协议

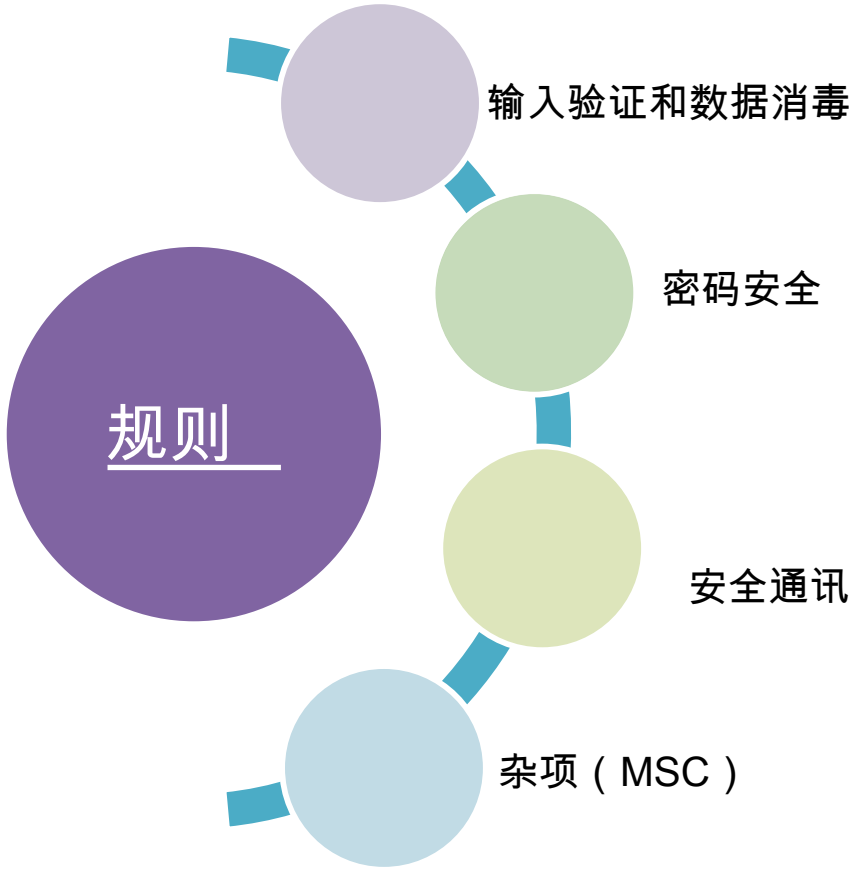
安全需求工程：两个方法

- 认证安全
  - 最佳做法，安全准则
  - 信息流控制
  - 静态和动态分析
- 设计安全
  - 安全目标
  - 威胁分析

安全知识源

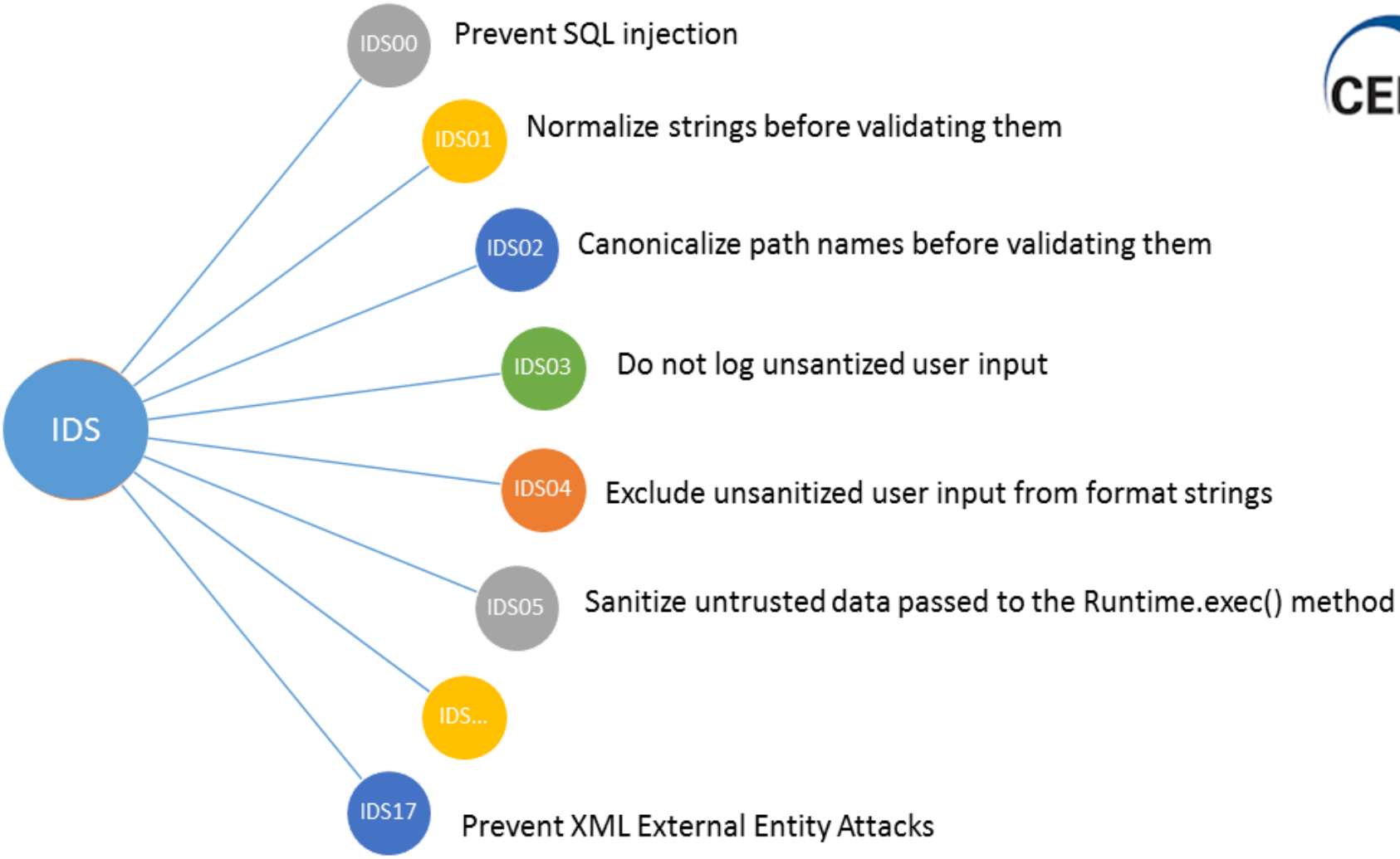
- 教科书
- 指导方针
- 标准品
- 检查清单
- 过去项目的文档安全设计模式
-
- 结构化目录和知识库

最佳做法/安全编程



来源：https://www.securecoding.cert.org/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java

最佳做法/安全编程


IDS
IDS00
Prevent SQL injection
IDS01
Normalize strings before validating them
IDS02
Canonicalize path names before validating them
IDS03
Do not log unsantized user input
IDS04
Exclude unsanitized user input from format strings
IDS05
Sanitize untrusted data passed to the Runtime.exec() method
IDS...
IDS17
Prevent XML External Entity Attacks
CERT

安全指南：目标+反模式

MSC62-J. Store passwords using a hash function

Créée par Matthew Wiethoff, dernière modification par David Svoboda le mai 13, 2016

Programs that store passwords as cleartext (unencrypted text data) risk exposure of those passwords in a variety of ways. Although programs generally receive passwords from users as cleartext, they should ensure that the passwords are not stored as cleartext.

An acceptable technique for limiting the exposure of passwords is the use of *hash functions*, which allow programs to indirectly compare an input password to the original password string without storing a cleartext or decryptable version of the password. This approach minimizes the exposure of the password without presenting any practical disadvantages.

Cryptographic Hash Functions

The value produced by a hash function is the *hash value* or *message digest*. Hash functions are computationally feasible functions whose inverses are computationally infeasible. In practice, a password can be encoded to a hash value, but decoding remains infeasible. The equality of passwords can be tested through the equality of their hash values.

A good practice is to always append a *salt* to the password being hashed. A salt is a unique (often sequential) or randomly generated piece of data that is stored along with the hash value. The use of a salt helps prevent brute-force attacks against the hash value, provided that the salt is long enough to generate sufficient entropy (shorter salt values cannot significantly slow down a brute-force attack). Each password should have its own salt associated with it. If a single salt were used for more than on password, two users would be able to see whether their passwords are the same.

The choice of hash function and salt length presents a trade-off between security and performance. Increasing the effort required for effective brute-force attacks by choosing a stronger hash function can also increase the time required to validate a password. Increasing the length of the salt makes brute-force attacks more difficult but requires additional storage space.

Java's `MessageDigest` class provides implementations of various cryptographic hash functions. Avoid defective functions such as the Message-Digest Algorithm (MD5). Hash functions such as Secure Hash Algorithm (SHA)-1 and SHA-2 are maintaine by the National Security Agency and are currently considered safe. In practice, many applications use SHA-256 because this hash function has reasonable performance while still being considered secure.

Noncompliant Code Example

This noncompliant code example encrypts and decrypts the password stored in `password.bin` using a symmetric key algorithm:

```
public final class Password {
  private void setPassword(byte[] pass) throws Exception {
    // Arbitrary encryption scheme
    bytes[] encrypted = encrypt(pass);
    clearArray(pass);
    // Encrypted password to password.bin
    saveBytes(encrypted,"password.bin");
    clearArray(encrypted);
  }

  boolean checkPassword(byte[] pass) throws Exception {
```

安全指南：合规示例

Compliant Solution

This compliant solution addresses the problems from the previous noncompliant code example by using a `byte` array to store the password:

```
import java.security.GeneralSecurityException;
import java.security.SecureRandom;
import java.security.spec.KeySpec;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.PBEKeySpec;

final class Password {
  private SecureRandom random = new SecureRandom();

  /* Set password to new value, zeroing out password */
  void setPassword(char[] pass)
      throws IOException, GeneralSecurityException  {
    byte[] salt = new byte[12];
    random.nextBytes(salt);
    saveBytes(salt, "salt.bin");
    byte[] hashVal = hashPassword( pass, salt);
    saveBytes(hashVal,"password.bin");
    Arrays.fill(hashVal, (byte) 0);
  }
  /* Indicates if given password is correct */
  boolean checkPassword(char[] pass)
      throws IOException, GeneralSecurityException  {
    byte[] salt = loadBytes("salt.bin");
    byte[] hashVal1 = hashPassword( pass, salt);
    // Load the hash value stored in password.bin
    byte[] hashVal2 = loadBytes("password.bin");
    boolean arraysEqual = timingEquals( hashVal1, hashVal2);
    Arrays.fill(hashVal1, (byte) 0);
    Arrays.fill(hashVal2, (byte) 0);
    return arraysEqual;
  }
```

Log Injection

*This is an **Attack**. To view all attacks, please see the Attack Category page.*

Last revision (mm/dd/yy): **06/6/2016**

Description

Writing unvalidated user input to log files can allow an attacker to forge log entries or inject malicious content into the logs.

Log forging vulnerabilities occur when:

1. Data enters an application from an untrusted source.
2. The data is written to an application or system log file.

Applications typically use log files to store a history of events or transactions for later review, statistics gathering, or debugging. Depending on the nature of the application, the task of reviewing log files may be performed manually on an as-needed basis or automated with a tool that automatically culls logs for important events or trending information.

Interpretation of the log files may be hindered or misdirected if an attacker can supply data to the application that is subsequently logged verbatim. In the most benign case, an attacker may be able to insert false entries into the log file by providing the application with input that includes appropriate characters. If the log file is processed automatically, the attacker can render the file unusable by corrupting the format of the file or injecting unexpected characters. A more subtle attack might involve skewing the log file statistics. Forged or otherwise, corrupted log files can be used to cover an attacker's tracks or even to implicate another party in the commission of a malicious act [1]. In the worst case, an attacker may inject code or other commands into the log file and take advantage of a vulnerability in the log processing utility [2].

Examples

The following web application code attempts to read an integer value from a request object. If the value fails to parse as an integer, then the input is logged with an error message indicating what happened.

```
...
String val = request.getParameter("val");
try {
        int value = Integer.parseInt(val);
}
catch (NumberFormatException) {
        log.info("Failed to parse val = " + val);
}
...
```

If a user submits the string "twenty-one" for val, the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

However, if an attacker submits the string "twenty-one%0a%0aINFO:+User+logged+out%3dbadguy", the following entry is logged:

```
INFO: Failed to parse val=twenty-one
```

描述：

“ *写作 未经验证的用户输入*

记录文件可以允许

攻击者伪造日志条目[...]日志伪造漏洞

发生在以下情况：

1）数据从 不可信来源 。[...]。 ”

合规代码

例：

正型

资源：

https://www.owasp.org/index.php / Log_Injection

Input Validation

This is a control. To view all control, please see the Control Category page.

This article is a stub. You can help OWASP by expanding it or discussing it on its Ta

This page contains draft content that has never been finished. Please help OWASP update this content! See FixME.
Last revision (yyyy-mm-dd): 2016-07-27
Comment: Content is poor

Using black and/or white lists which defines valid input data. Such approach is more accurate and provides better risk analysis, when there is need of modification of the lists.

E.g. When we expect digits as an input, then we should perform accurate input data validation.

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

int main(int argc, char **argv)
{
        char a[256];
        strncpy(a, argv[1], sizeof(a)-1);

        int b=0;

        for(b=0; b<strlen(a); b++) {
                if(isdigit((int)a[b])) printf("%c", a[b]);
        }

        printf("\n");
        return 0;
}
```

In PHP for input data validation we may use e.g. preg_match() function:

```
<?php
  $clean = array();
  if (preg_match("/^[0-9]+:[X-Z]+$/D", $_GET['var'])) {
```

描述：

“使用黑名单和/或白名单定义有效输入数据。当需要以下方法时，这种方法更加准确，并且可以提供更好的风险分析

修改列表”。

合规代码

例：

正型

资源：

https://www.owasp.org/index.php/Log_Injection

安全准则问题

- 抽象和不精确的描述
- 非正式规范–如何验证合规性？
- 可能会被开发人员误解
- 通常是特定于编程语言的
- 多个重叠的目录（CERT，OWASP等）

- 这些由代码扫描程序支持，这些代码扫描程序通过静态分析实施临时的合规性检查
  - 问题：没有执行验证的描述

防御性编程

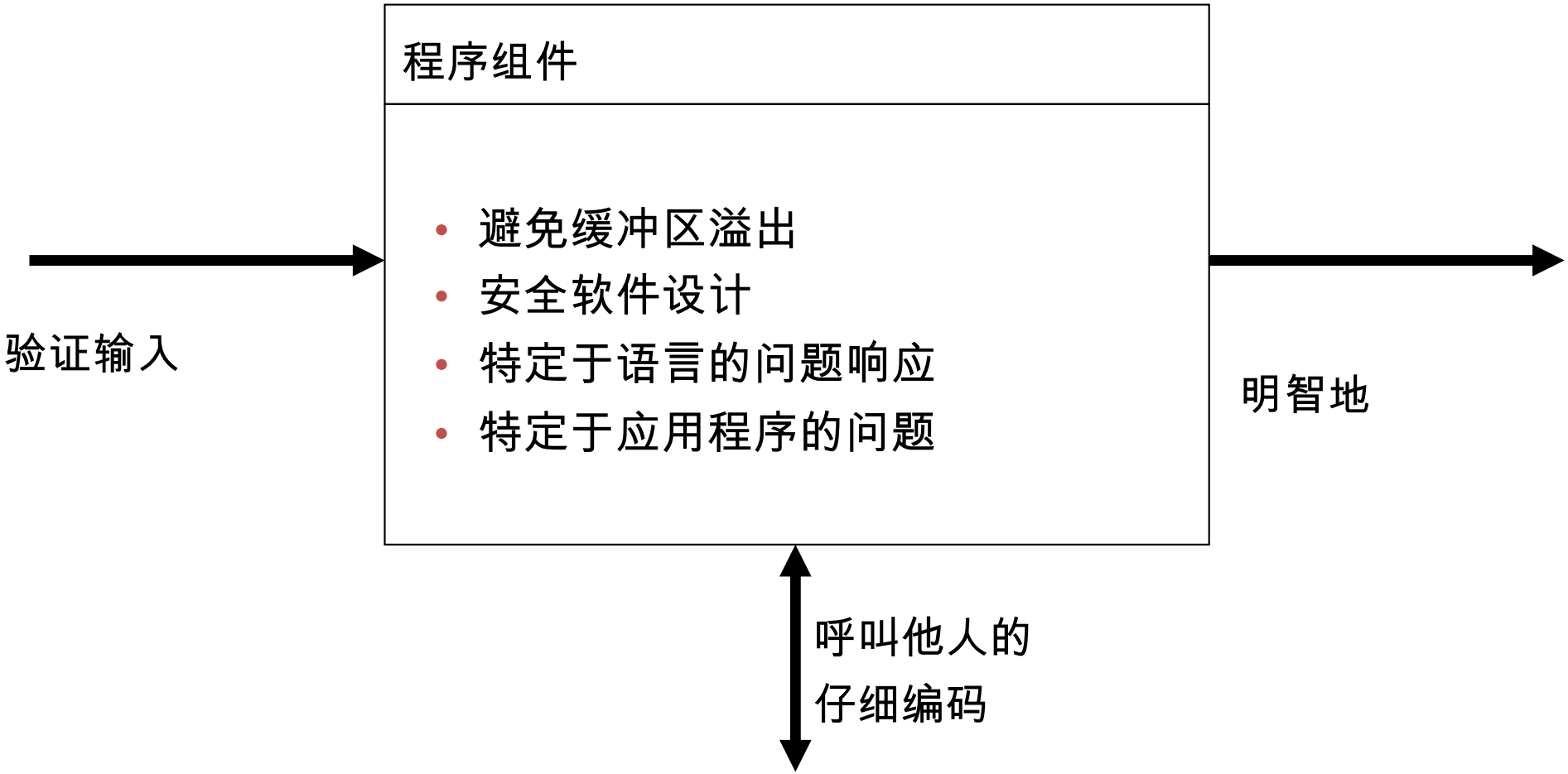
原则 [Viega和McGraw]

- 确保最薄弱的环节
- 深入练习防御
- 安全失败
- 遵循最小特权原则
- 划分
- 把事情简单化
- 促进隐私
- 记住躲藏起来很难
- 不愿信任
- 使用您的社区资源

确保最薄弱的环节

- 考虑可能的攻击
  - 有人将如何尝试对此进行攻击？
  - 他们想完成什么？
- 查找最薄弱的链接
  - 加密库可能还不错
  - 有没有办法解决加密问题？
    - 数据以加密形式存储；哪里 *键* 储存？
- 重点
  - 对整个系统进行安全性分析
  - 在重要时刻花时间

一般类别


程序组件
避免缓冲区溢出
安全软件设计
特定于语言的问题响应
特定于应用程序的问题
验证输入
明智地
呼叫他人的
仔细编码

检查安全软件

- 编写安全代码的许多规则
  - “在使用之前对用户输入进行消毒”
  - “执行操作X之前检查权限”
- 如何发现错误？
  - 正式验证
    - \+ 严格
    - – 昂贵，昂贵。*非常*软件罕见
  - 测试：
    - \+ 简单，误报少
    - – 需要运行代码：无法扩展且可能不切实际
  - 人工检查
    - \+ 灵活
    - – 不稳定且缩放比例不佳。

两种选择

- 动态分析
  - 运行代码（可能在检测条件下），以查看是否可能存在问题
- 静态分析
  - 检查代码或运行自动化方法以查找错误或对错误的出现信心满满

静态与动态分析

- 动态
  - 需要选择样本测试输入
  - 可以发现漏洞
  - 无法证明他们的缺席

- 静态的
  - 考虑所有可能的输入（汇总形式）
  - 查找错误和漏洞
  - 在某些情况下可以证明没有错误

正态动力分析

- 在检测执行环境中运行程序
  - 二进制翻译器，静态仪器，仿真器
- 寻找坏东西
  - 使用无效的内存，竞争条件，空指针取消引用等。
- 示例：Purify，Valgrind，Normal OS异常处理程序（崩溃）

回归与模糊

- 回归：在许多正常输入上运行程序，查找缺陷。
  - 防止普通用户遇到错误（例如断言错误）。
- 模糊测试：在许多异常输入上运行程序，查找缺陷。
  - 防止攻击者遇到可利用的错误（例如，经常断言）好）

模糊化基础

- 自动生成测试用例
- 许多轻微异常的测试用例被馈入目标界面
  - API或“协议”
- 监视应用程序是否有错误
  - 监控可能需要仪器输入
-
  - 基于文件（.pdf，.png，.wav，.mpg）
  - 基于网络（http，SNMP，SOAP）
  - 其他（例如与 crashme（ ））

琐碎的例子

- 标准HTTP GET请求
  - GET /index.html HTTP / 1.1
- 异常请求
  - AAAAAA ... AAAA /index.html HTTP / 1.1
  - GET ////////index.html HTTP / 1.1
  - GET%n%n%n%n%n%n.html HTTP / 1.1
  - GET /AAAAAAAAAAAAA.html HTTP / 1.1
  - GET /index.html HTTTTTTTTTTTTTPTP / 1.1
  - GET /index.html HTTP / 1.1.1.1.1.1.1.1

产生输入的不同方法

- 基于变异-“哑巴模糊测试”
- 基于生成的“智能模糊测试”

基于变异的模糊测试

- 很少或根本不了解输入的结构
- 异常被添加到现有有效输入中
- 异常可能是
  - 完全随机
  - 或遵循一些试探法（例如删除终止符，向前移动字符）

变异模糊测试：利弊

- 长处
  - 设置简单，（自动化）简单
  - 几乎不需要“协议”知识
- 弱点
  - 受初始语料库的限制
  - 对于带有校验和的协议，依赖于质询响应的协议等，可能会失败。

基于世代的模糊测试

- 测试用例是根据格式的一些描述生成的
  - RFC，文档等
- 将异常添加到输入中的每个可能位置
- 与随机模糊测试相比，“协议”知识应能提供更好的结果

基于世代的模糊测试：优点和缺点

- 长处
  - 完整性
  - 可以处理复杂的依赖关系（例如校验和）
- 弱点
  - 需要协议规范
    - 通常可以找到适用于现有协议的好的工具
  - 对于复杂的协议，编写生成器可能会很费力
  - 规范不是代码（实现可能与不完整或错误解释的规范相对应）

注入输入

- 最简单的
  - 在模糊文件上运行程序（文件模糊）
  - 重播模糊的数据包跟踪（网络模糊）
- 修改现有程序/客户端
  - 代码在适当的时候调用模糊器
- 使用模糊测试框架
  - 例如Peach自动生成COM接口模糊器

问题检测

- 查看程序是否崩溃
  - 崩溃类型可以说明很多问题（SEGV与断言失败）
- 在动态内存错误检测器（valgrind / purify）下运行程序
  - 捕获更多错误，但每次运行的代价更高。
- 查看程序是否锁定
- 滚动自己的检查器，例如valgrind皮肤

多少绒毛够了？

- 基于变异的模糊器可以生成无数的测试用例...模糊器何时运行足够长的时间？

- 基于生成的模糊器生成有限数量的测试用例。它们全部运行且未发现错误时会发生什么？

代码覆盖率

- 这些问题的一些答案在于
  代码覆盖率
- 代码覆盖率是一个指标，可以用来确定已执行了多少代码。

- 可以使用各种配置文件工具获取数据。例如 gcov

代码覆盖率的类型

- 线路覆盖
  - 测量已执行了多少行源代码。
- 分行覆盖
  - 衡量已在代码中进行了多少分支（有条件的jmp）
- 路径覆盖
  - 衡量已采取的路径

静态分析基础

- 抽象地模拟程序属性，查找问题
- 工具来自程序分析
  - 类型推断，数据流分析，定理证明
- 通常在源代码上，可以在字节码上或反汇编
- 长处
  - 完整的代码覆盖范围（理论上）
  - 潜在地验证是否存在/报告整个类错误的所有实例
  - 捕获与动态分析不同的错误
- 弱点
  - 假阳性率高
  - 许多属性无法轻松建模
  - 建造困难
  - 几乎没有真正系统中的所有源代码（操作系统，共享库，动态加载等）。

两种类型的静态分析

- （而是）简单的代码分析。
  - 查找已知的代码问题：例如，不安全的字符串函数 strncpy（），sprintf（），gets（）
  - 在您的源代码库中寻找不安全的功能
  - 查找重复出现的问题代码（问题界面，错误代码的复制/粘贴等）

- 更深入的分析
  - 需要复杂的代码解析和计算
  - 有些是在诸如Coverity，Fortify，Visual Studio等工具中实现的。

  - 否则必须在解析器（如LLVM）之上开发

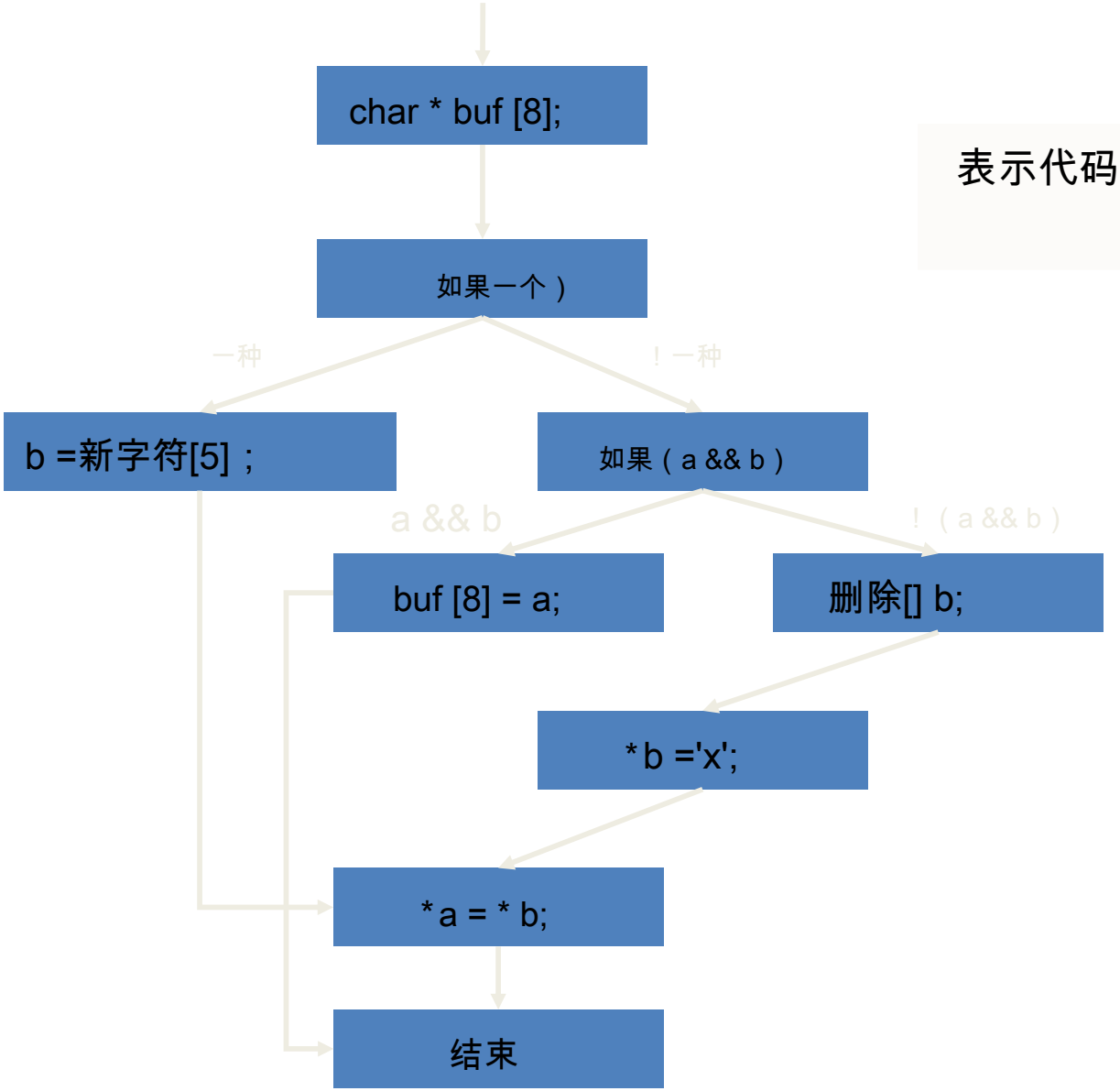
静态分析：健全性，完整性

属性	定义
健全性	“报告正确性的声音”分析说没有错误 ® 没有错误 或同等 有一个错误 ® 分析发现错误
完整性 “报告正确性完整”	没有错误 ® 分析说没有错误

召回：A ® B等于（ ¬ B ）®（ ¬ 一种 ）

	完成	不完整
声音	报告所有错误 无误报 **犹豫不决**	报告所有错误 可能会报告错误警报 **可决定的**
不健全	可能不会报告所有错误报告没有错误警报 **可决定的**	可能不会报告所有错误可能会报告错误警报 **可决定的**

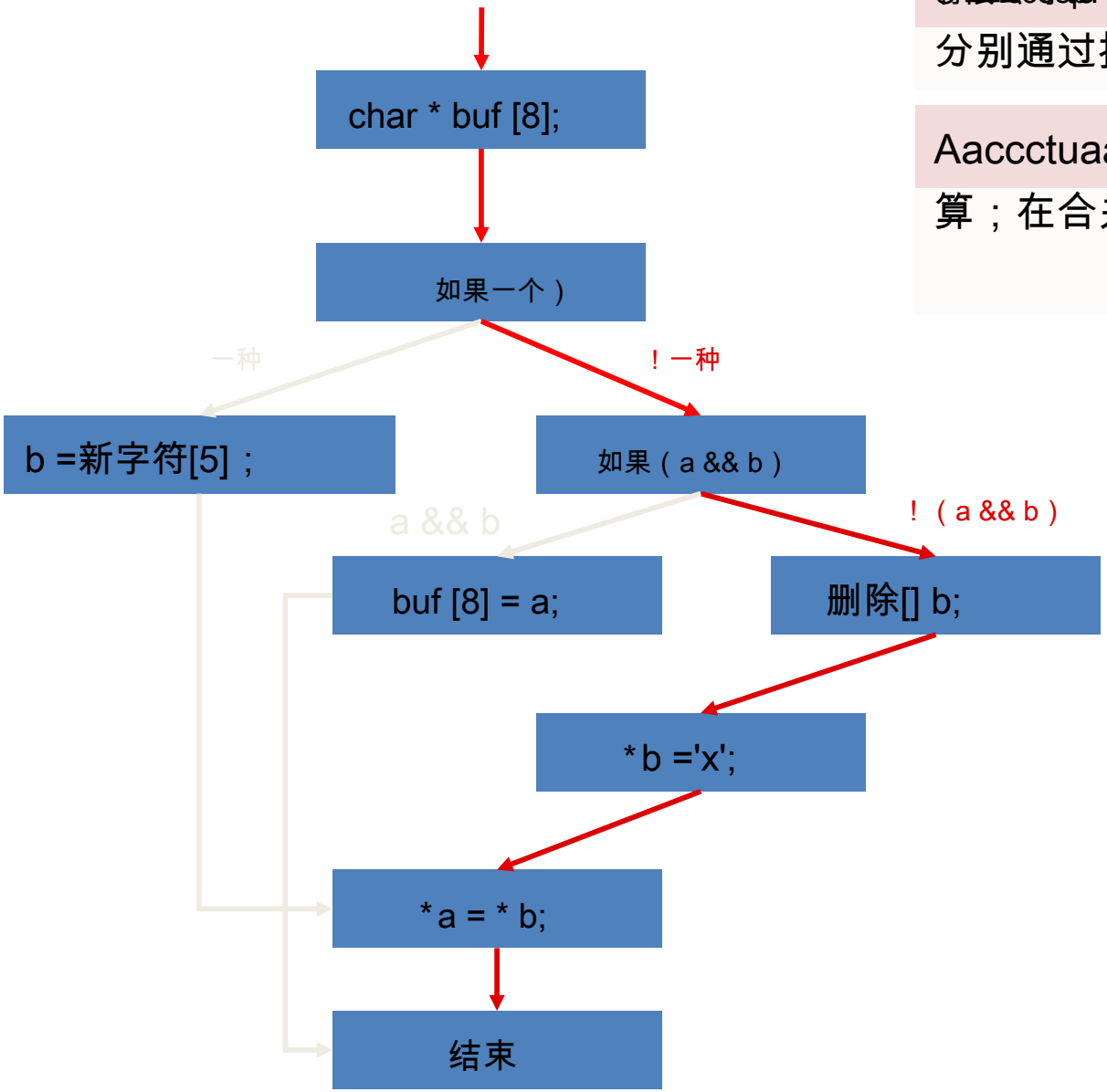
控制流程图



表示代码的逻辑结构

以图形形式

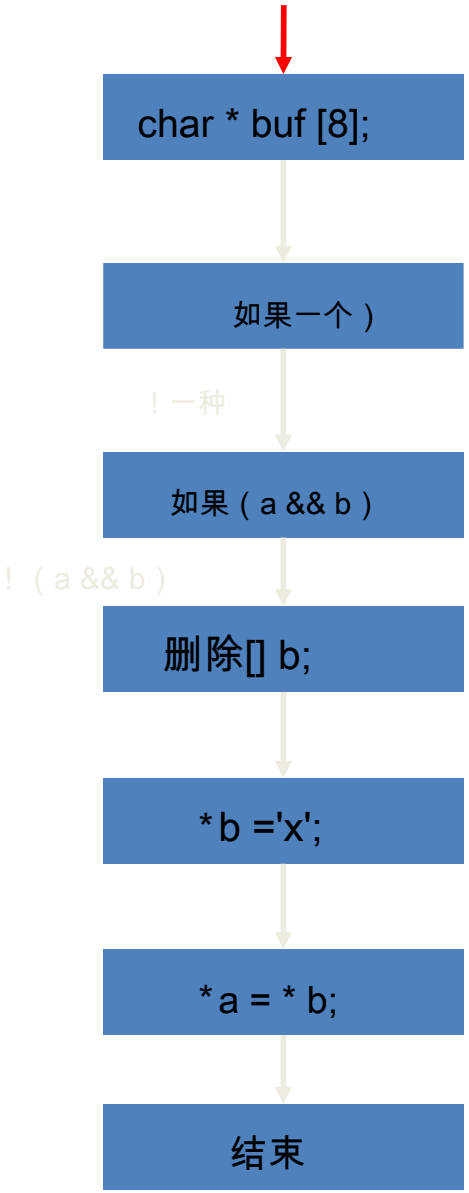
路径遍历



有限公司tu uaalllyy：分析每条路径分别通过控制图

Aaccctuaalllyy每个节点执行一次检查计算；在合并节点处合并路径

申请检查



空指针　　免费使用　　数组超限

查看如何为此路径运行三个检查器

检查器

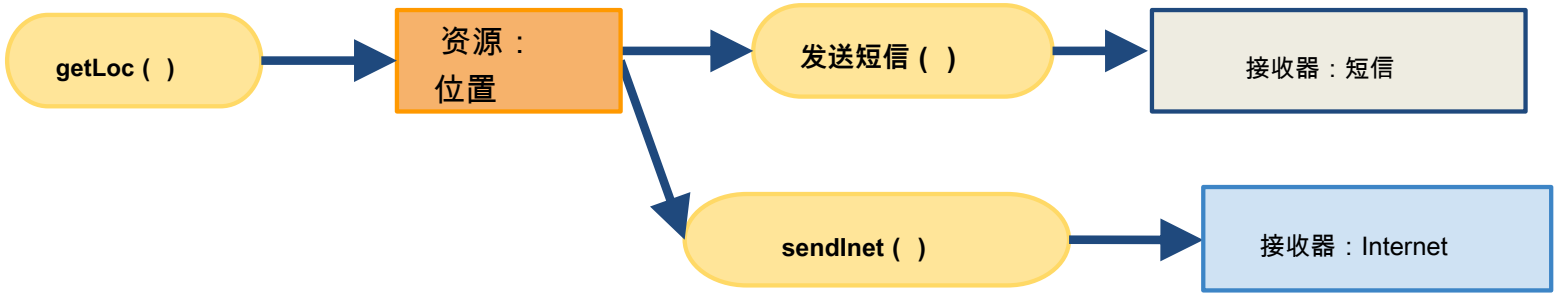
- 由状态图定义，具有状态转换和错误状态

RUn Checker

- 为每个程序变量分配初始状态
- 程序点的状态取决于先前点的状态，程序动作

- 如果达到错误状态则发出错误

数据流分析



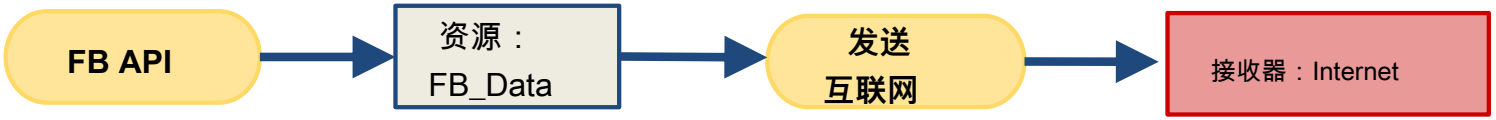
- 小号 Ø 流到汇

Ø 来源：位置，日历，联系人，设备ID等。

数据流分析的应用

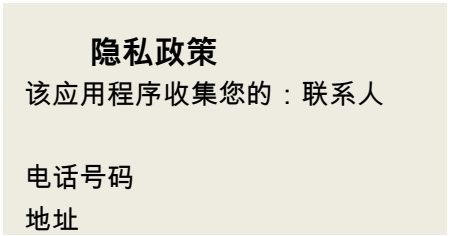
- 漏洞发现
- 中号 Ø 恶意软件/灰色软件分析

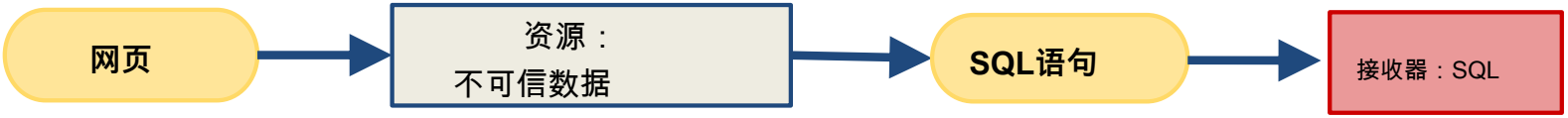
  数据流摘要可启用特定于企业的策略
- API滥用和数据盗窃检测



- 一种 Ø 应用程序隐私策略的自动生成

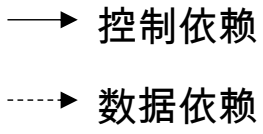
  避免责任，保护消费者隐私

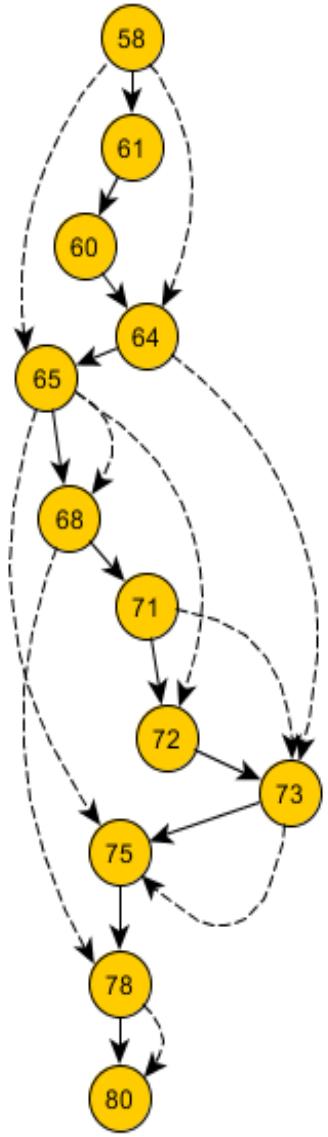




程序依赖图（PDG）

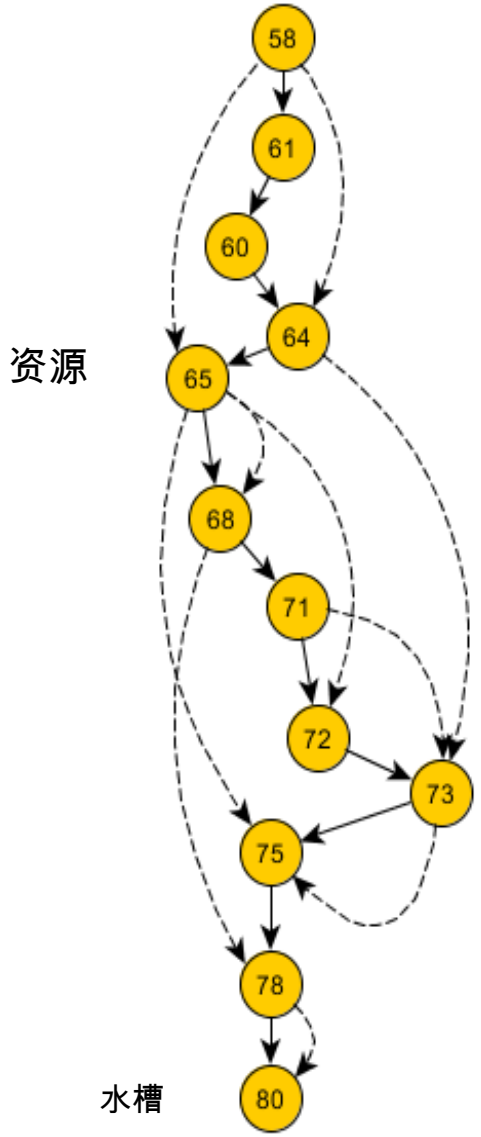
- 控制依赖
- 显式+隐式数据依赖
- 特性：
  - 路径敏感
  - 上下文相关
  - 物体敏感





JOANA IFC工具

- 用于信息流分析
- 注释：SINK / SOURCE
- 无干扰：安全级别（高/低）



好处：自动化代码分析

- 例子
  - C语言：FindBugs，Fortify，Coverity，MS工具（商业），KLEE（学术）
  - Java语言：JOANA，FindBugs，Soot，PMD（学术）

- 目标
  - 在所有人都可以使用的工具中捕获专家已知的推荐做法
  - 还捕获信息流，无干扰

动态分析

- 仪器测试代码
  - 堆内存：净化
  - Perl污染（信息流）
  - Java竞争条件检查
- 黑匣子测试
  - 模糊测试和渗透测试
  - 黑盒Web应用程序安全性分析

净化

- 目标
  - 检测程序以检测运行时内存错误（越界，初始化前使用）和内存泄漏
- 技术
  - 适用于可重定位目标代码
    - 链接到提供跟踪表的已修改malloc
  - 内存访问错误：在每个加载和存储指令之前插入指令序列
  - 内存泄漏：GC算法

持久污染

- Perl代码的运行时检查
  - Perl用于CGI脚本，对安全性敏感
  - 污秽检查阻止了一些潜在的不安全呼叫
- 污染的弦
  - 用户输入，来自用户输入的值
  - 与未受污染的字符串匹配的结果除外
- 禁止通话
  - 打印$ form_data {“ email”}。“ \ n”;
    - 确定，因为可以安全打印（???）
  - system（“ mail”。$ form_data {“ email”}）;
    - 带用户输入作为参数的标记系统调用