大连理工大学
Dalian University Of Technology

软件开发安全设计

大连理工大学　　软件学院

马瑞新

目录

软件开发安全概述

软件开发的特点

- 软件开发具有以下几个特点：
  - ◆脑力密集型
  - ◆实现不具有唯一性
  - ◆隐性成本高
  - ◆细节很容易被放大
  - ◆质量评估很需要专业的高水平

软件安全重要性 – 软件危机

- 第一次“软件危机”- 20世纪60年代
  - ◆根源：日益庞大和复杂的程序对开发管理的要求越来越高
  - ◆解决：软件工程
- 第二次“软件危机”- 20世纪80年代
  - ◆根源：软件规模继续扩大，程序数百万行，数百人同时开发，可维护性难
  - ◆解决：面向对象语言-C++/java/c#
- 第三次“软件危机”- 21世纪头十年
  - ◆根源：软件安全
  - ◆解决：软件安全开发生命周期管理

知识子域：软件安全开发生命周期

- 软件生命周期模型
  - ◆了解典型的软件开发生命周期模型的特点，包括瀑布模型、迭代模型、增量模型、快速原型模型、螺旋模型、净室模型；
- 软件安全保障
  - ◆了解软件安全和软件安全保障的基本概念；
  - ◆理解软件安全开发的必要性；
  - ◆理解软件安全问题增加的原因；
  - ◆了解软件安全问题相关的几个术语的概念。

软件安全

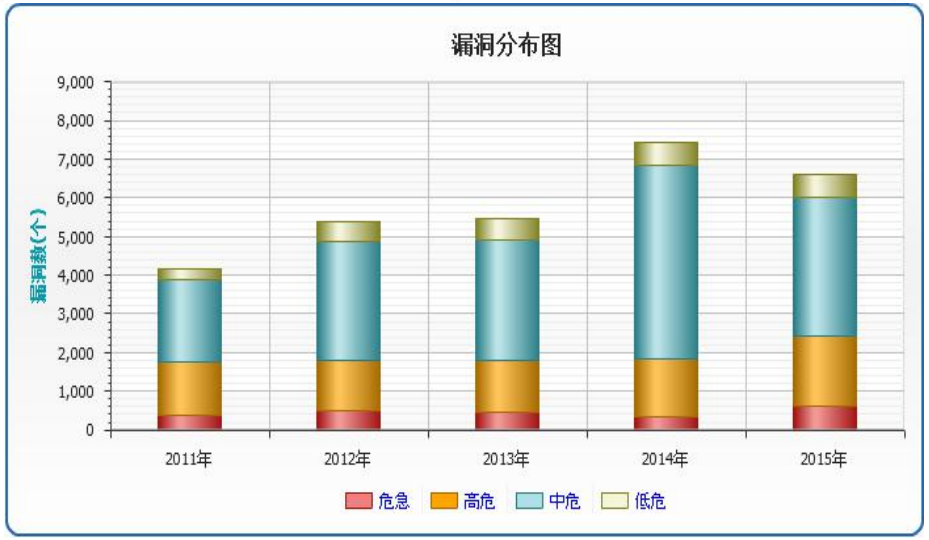
- 安全性：使伤害或损害的风险控制在可接受的水平内----ISO8402
  - ◆软件安全（Software Security）：将开发的软件存在的风险控制在可接受的水平，以保证软件的正常运行。
  - ◆软件存在漏洞--〉容易成为攻击目标
  - ◆安全的软件：即使它的安全性受到蓄意危害时仍然保持安全可靠运行。

软件缺陷普遍存在

- 千行代码缺陷数量
  - ◆普通软件公司：4～40
  - ◆高管理软件公司：2~4
  - ◆美国NASA软件：0.1
- 漏洞数量
  - ◆ 国家漏洞库统计数据，最近5年最高一年入库漏洞7000+

CMMI 分级标准

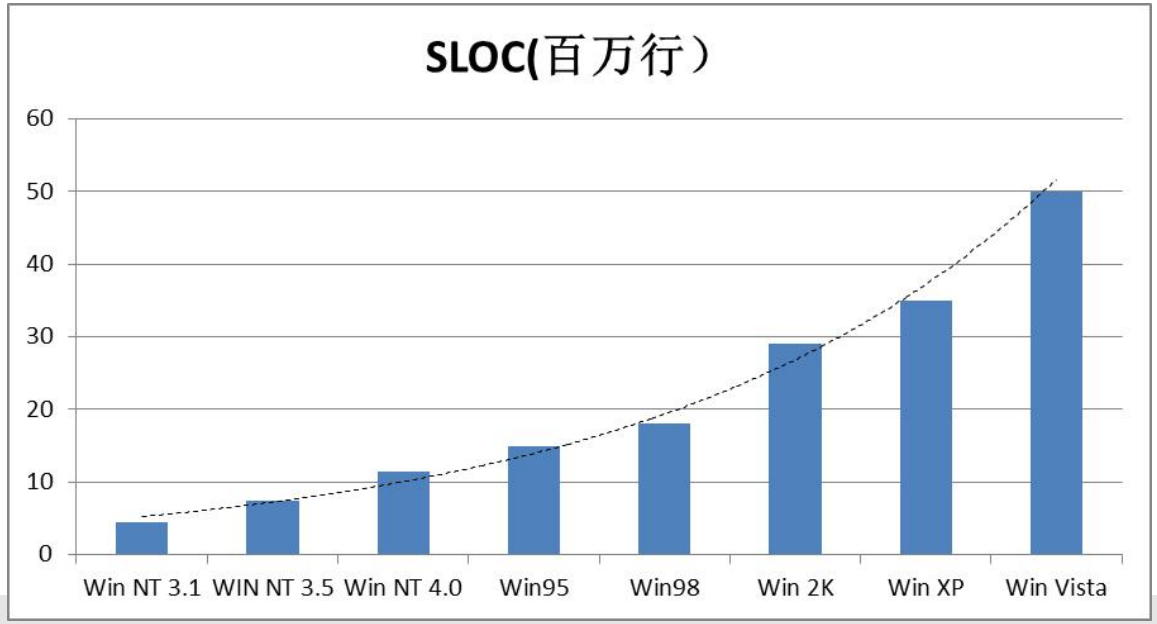
Thousands of lines of code defects	
CMMI1	11.95‰
CMMI2	5.52‰
CMMI3	2.39‰
CMMI4	0.92‰
CMMI5	0.32‰



国家漏洞库漏洞数量统计

软件安全问题产生-内因

- 内因
  - ◆软件规模增大,功能越来越多,越来越复杂
  - ◆软件模块复用，导致安全漏洞延续
  - ◆软件扩展模块带来的安全问题



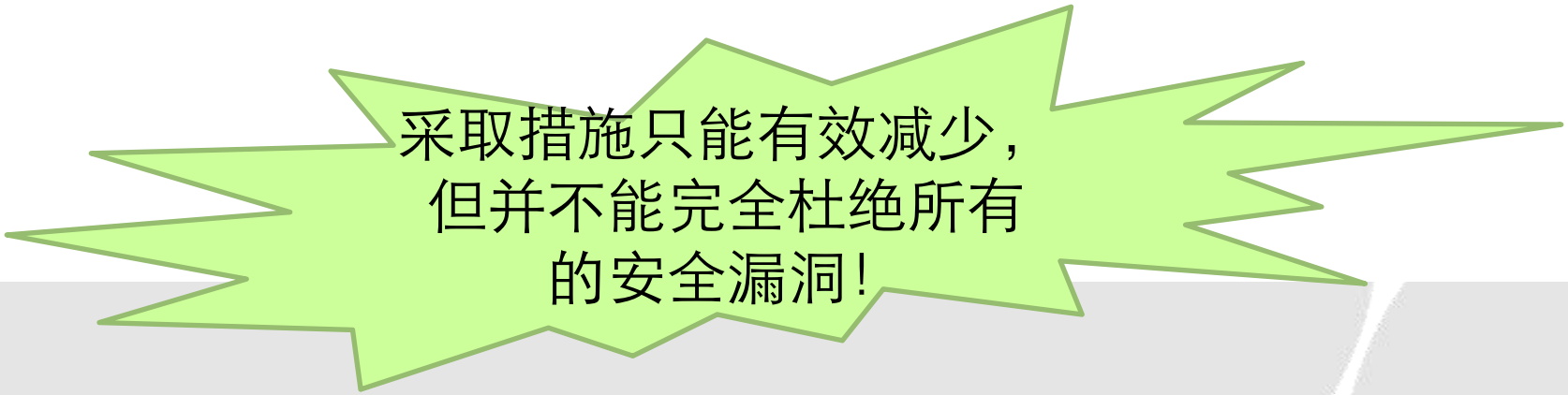
Windows操作系统不同版本源代码数量

软件安全问题产生-外因

- 互联网发展对软件安全的挑战
- 开发环境和开发人员对软件安全的挑战
  - ◆开发者缺乏安全开发的动机
    - 市场和业务要求将交付期和软件功能做主要因素
    - 用户方没有提供安全方面的压力
  - ◆开发者缺乏相关知识
    - 软件复杂性加大，开发者需要学习更多东西
    - 传统软件开发不进行安全教育
  - ◆缺乏安全开发工具
    - 缺乏安全开发配套管理、测试等工具

软件安全保障的含义及思路

- 软件安全保障的概念
  - ◆确保软件能够按照开发者预期、正常地执行任务，提供与威胁相适应的安全能力，从而避免存在可以被利用的安全漏洞，并且能从被入侵和失败的状态中恢复。
- 软件安全保障的思路
  - ◆通过在软件开发生命周期各阶段采取必要的、相适应的安全措施来避免绝大多数的安全漏洞



软件安全保障的目标

- 软件可以规避安全漏洞而按照预期的方式执行其功能
- 在软件开发生命周期中提升软件的安全性
  - ◆可信赖性：无论是恶意而为还是无意疏忽，软件都没有可利用的漏洞存在
  - ◆可预见性：对软件执行时其功能符合开发者的意图的信心。
  - ◆遵循性：将（软件开发）跨学科的活动计划并系统化，以确保软件过程和软件生产

软件安全保障与风险管理

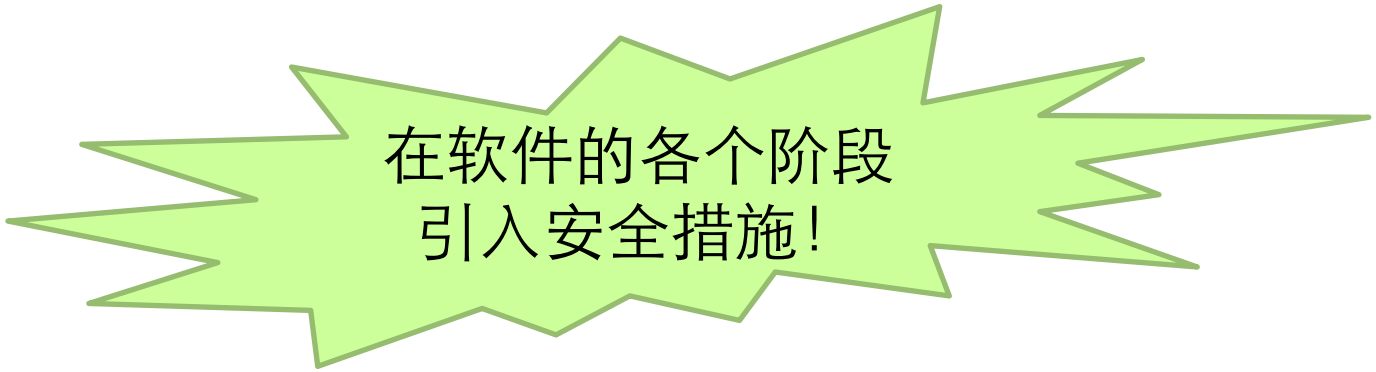
- 在软件安全保障中，需要贯彻风险管理的思想
  - “安全就是风险管理”
- 软件安全是以风险管理为基础
  - 安全不必是完美无缺的，但风险必须是能够管理的
- 最适宜的软件安全策略就是最优的风险管理对策
  - 这是一个在有限资源前提下的最优选择问题
  - 防范不足会造成直接的损失；防范过多又会造成间接的损失

传统软件开发中安全局限性

- 传统的软件生命周期的局限性：
  - ◆软件生命周期包括需求分析、架构设计、代码编写、测试和运行维护五个阶段，缺乏安全相关阶段
- 传统的软件开发教育局限性
  - ◆软件教育包括软件工程、数据结构、编译原理、系统结构、程序语言缺乏安全开发教育
- 开发人员局限性
  - ◆对安全问题没有的足够理解
  - ◆不了解安全设计的基本原理
  - ◆不知道安全漏洞的常见类型
  - ◆不知道如何设计针对安全的测试数据

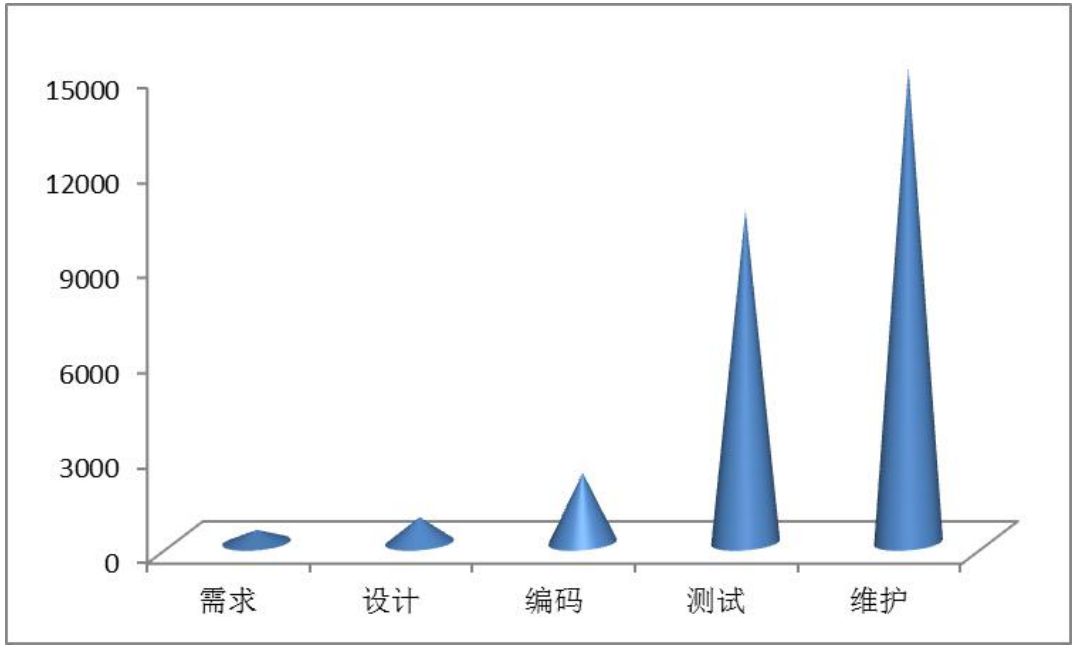
软件安全开发生命周期

- 软件安全开发覆盖软件整个生命周期
  - ◆需求分析阶段考虑软件的安全需求
  - ◆在设计阶段设计符合安全准则的功能
  - ◆编码阶段保证开发的代码符合安全编码规范
  - ◆安全测试和运行维护确保安全需求、安全设计、安全编码各个环节得以正确有效的实施



软件安全问题越早解决成本越低

- 在软件开发生命周期中，后面的阶段改正错误开销比前面的阶段要高出数倍
- NIST：在软件发布以后进行修复的代价是在软件设计和编码阶段即进行修复所花代价的30倍



知识子域：软件安全开发生命周期

- 软件安全开发生命周期模型
  - ◆理解典型的软件安全开发生命周期模型的特点，包括SDL、CLASP、CMMI、SAMM、BSIMM；
  - ◆了解这些软件安全开发生命周期模型的区别和联系。

软件安全概述

◆如何评判一个软件是否足够安全？满足以下属性:

1、机密性。软件必须确保其特性（包括运行环境和用户之间的联系）、资源管理和内容对未授权实体隐藏。

2、完整性。软件及其管理的资源必须能够抵御入侵（覆盖、删除、修改等）并能从中恢复。

3、可用性。软件必须对授权用户（人或进程）开放，即可操作和运行，而对未授权用户关闭。相反，对于未授权的用户应该永远关闭，既不可操作和运行。

4、可追踪性。所有与安全相关的软件行为都必须跟踪与记录，并进行责任归因。

5、抗抵赖性。这一属性是指软件防止用户否认执行了某些行为的功能，以保证可追踪性不受到破坏。

软件安全问题

◆已经成为上到国家下到个人的关键问题

◆软件安全漏洞

- 计算机程序、系统或协议中存在的安全漏洞，它已成为被攻击者用来非法侵入他人系统的主要渠道。
- 对于企业网络的攻击
  - 超过70%来自软件和应用层
  - 92%被发现的漏洞属于应用层

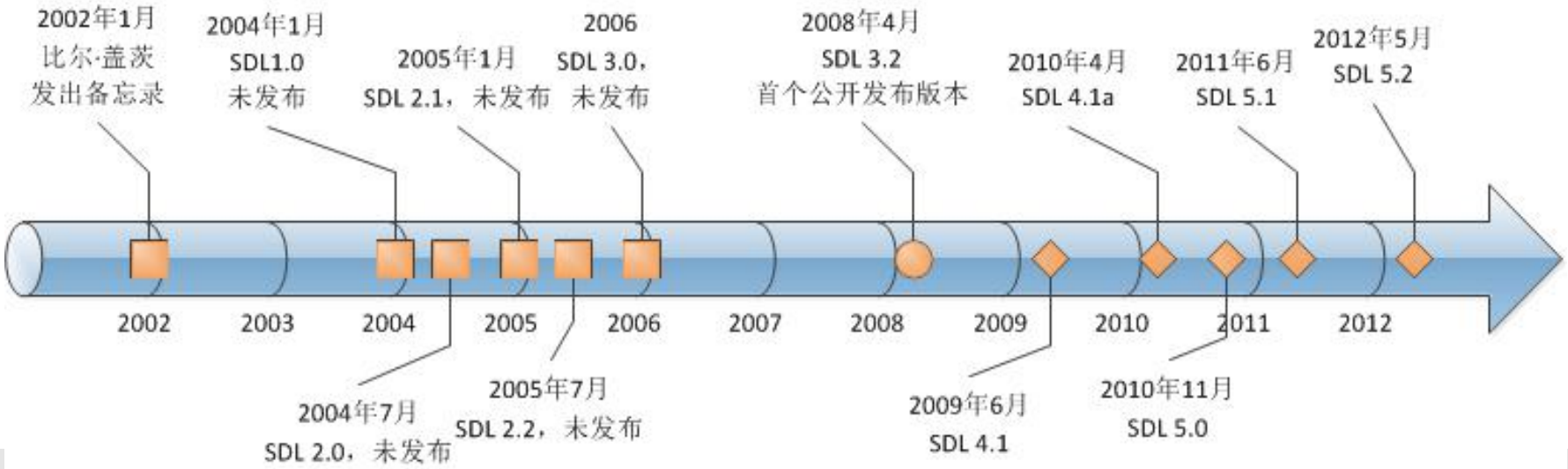
软件开发安全模型

相关模型和研究

- 安全软件开发生命周期
  - 安全设计原则
  - 安全开发方法
  - 最佳实践
  - 安全专家经验
- 多种模型被提出和研究
  - 可信计算安全开发生命周期（微软）
  - CLASP（OWASP）综合的轻量应用安全过程
  - BSI系列模型（Gary McGraw等）
  - SAMM（OWASP）软件保证成熟度模型

SDL

- 什么是SDL
  - ◆SDL（Security Development Lifecycle，安全开发生命周期）
- SDL发展



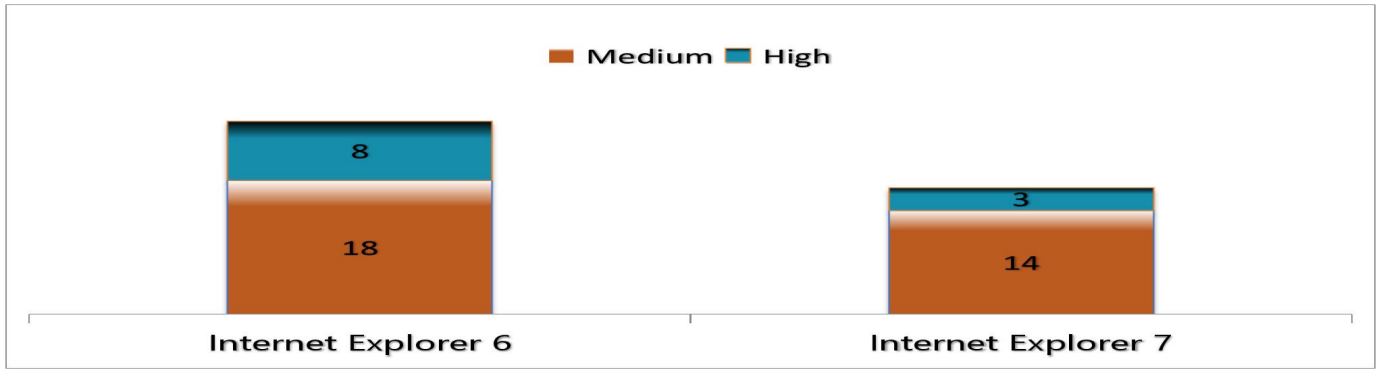
SDL的阶段和安全活动

- 软件安全开发生命阶段
  - ◆5+2个阶段
  - ◆16项必需的安全活动

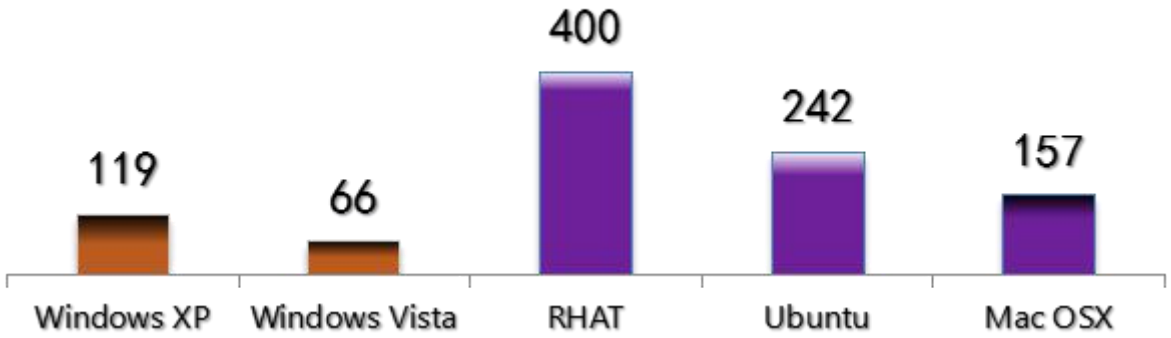
SDL实施效果

- 正式发布软件后12个月内的漏洞对比

◆IE： 漏洞总数下降35%，高危漏洞数下降63%



◆操作系统：漏洞总数降低45%

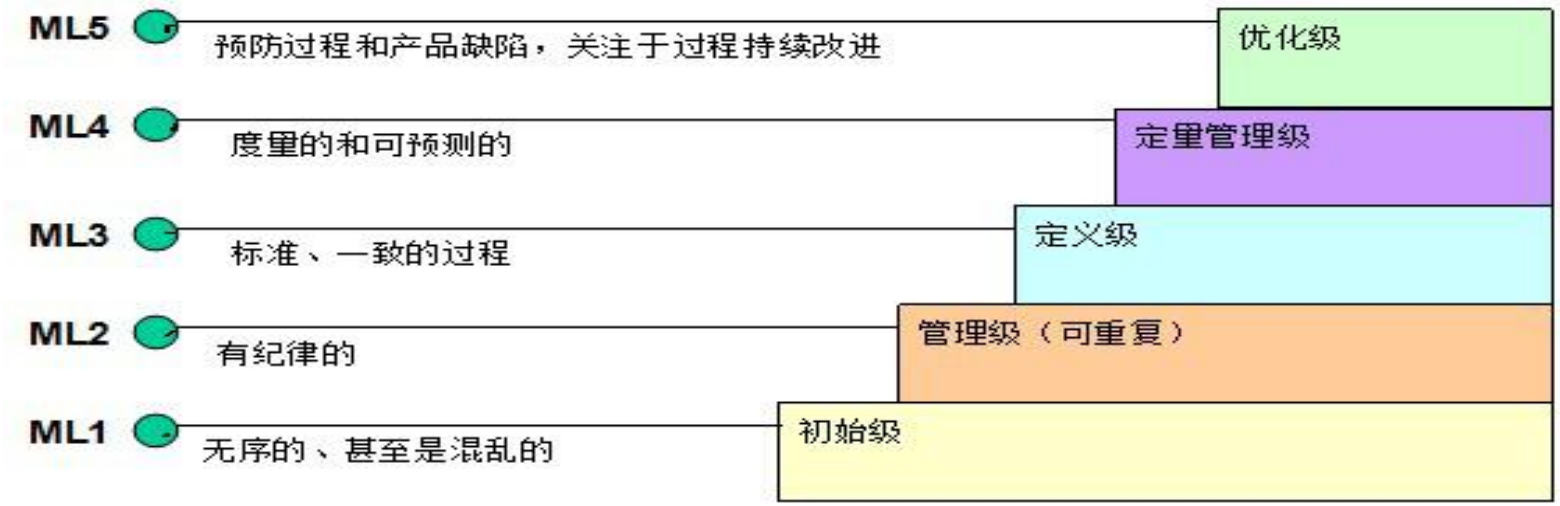


CLASP

- 什么是 CLSAP
  - ◆综合的轻量应用安全过程（Comprehensive, Lightweight Application Security Process，CLASP）
  - ◆用于构建安全软件的轻量级过程
  - ◆包括由30个特定的活动(activities)和辅助资源构成的集合
  - ◆针对这些活动给出了相应的指南、导则和检查列表
- 特点
  - ◆基于角色的安排

CMMI

- 什么是CMMI
  - ◆软件能力成熟度集成模型(Capability Maturity Model Integration)
- 五级



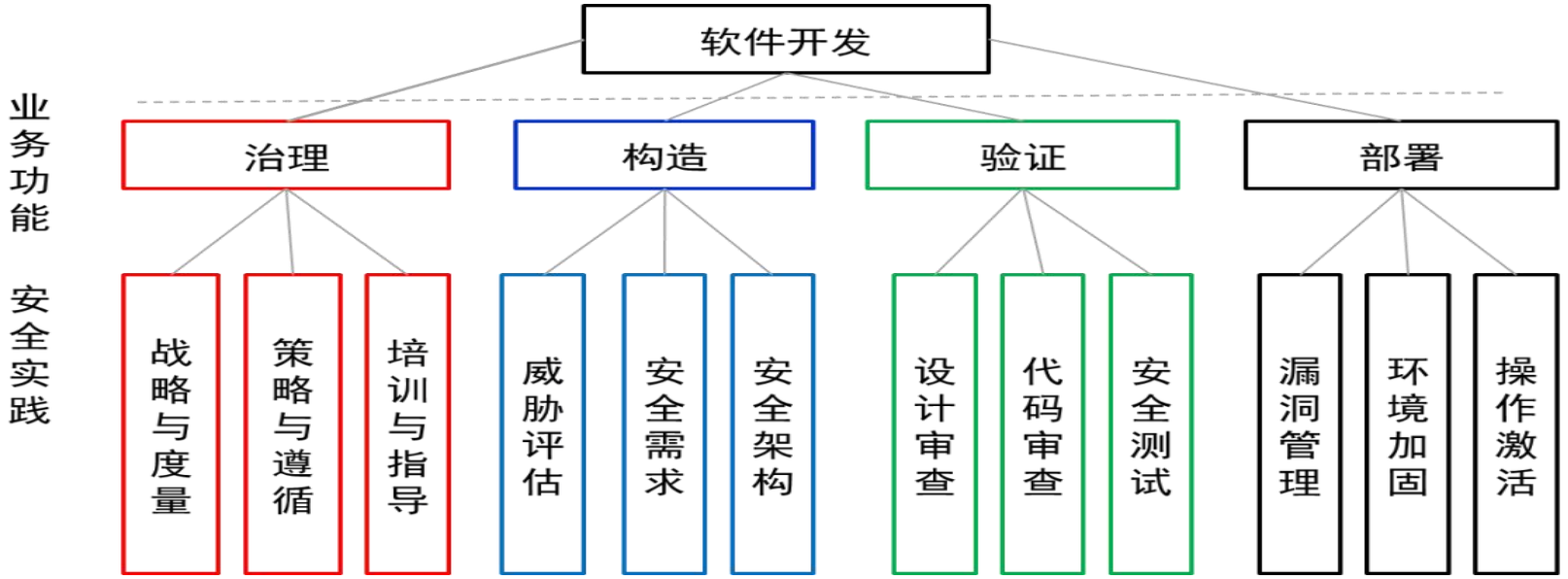
- 过程区域

SAMM

什么是SAMM

◆软件保证成熟度模型（Software Assurance Maturity Mode，SAMM）

◆提供了一个开放的框架，用以帮助软件公司制定并实施所面临来自软件安全的特定风险的策略，



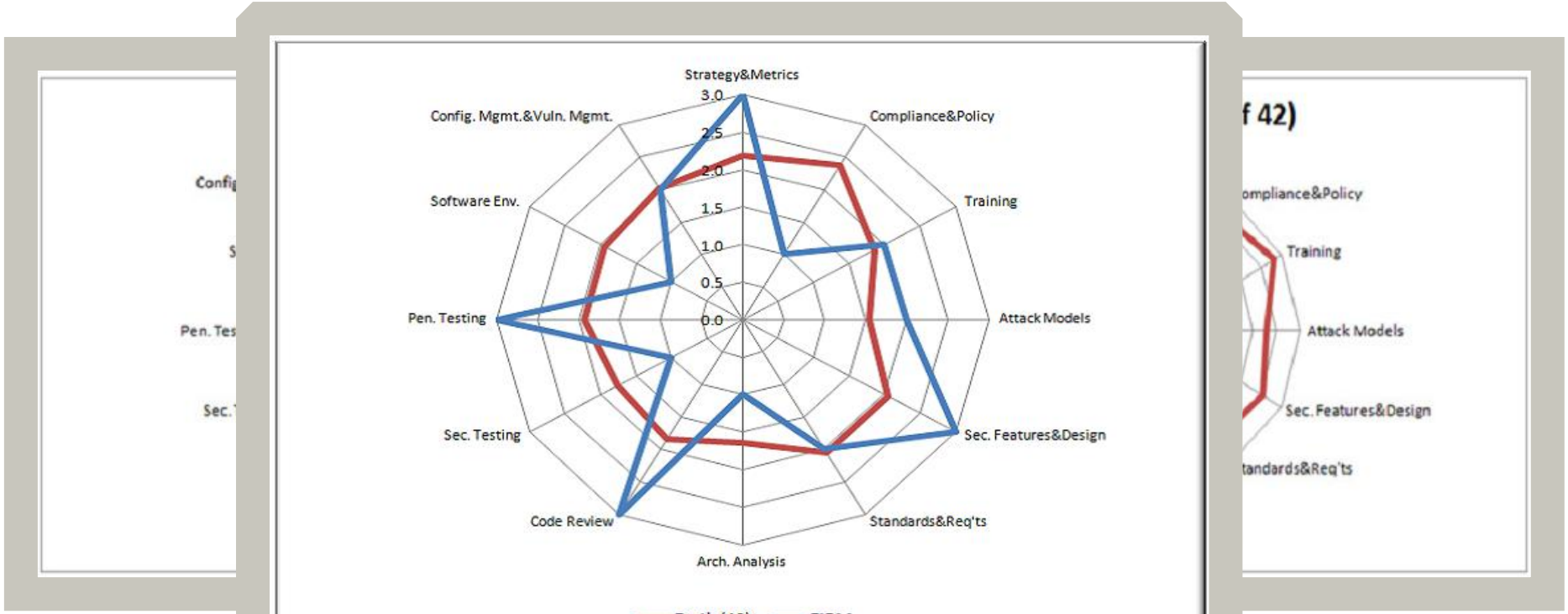
BSI系列模型

- **BSI(Building Security IN)**
  - ◆使安全成为软件开发必须的部分
  - ◆强调应该使用工程化的方法来保证软件安全
- 软件安全的三根支柱
  - ◆风险管理：策略性方法
  - ◆接触点：一套轻量级最优工程化方法，攻击与防御综合考虑
  - ◆安全知识：强调对安全经验和专业技术进行收集汇总，对软件开发人员进行培训，并通过安全接触点实际运用

BSI成熟度模型

◆对真实的软件安全项目所开展的活动进行量化

◆构建和不断发展软件安全行动的指南

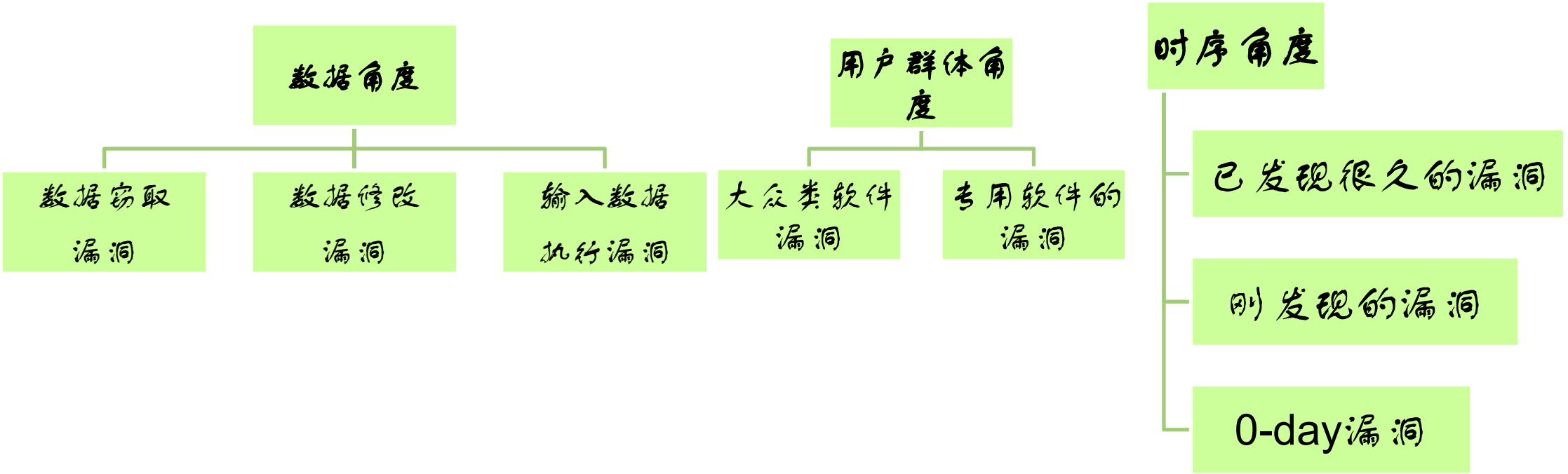


各模型比较

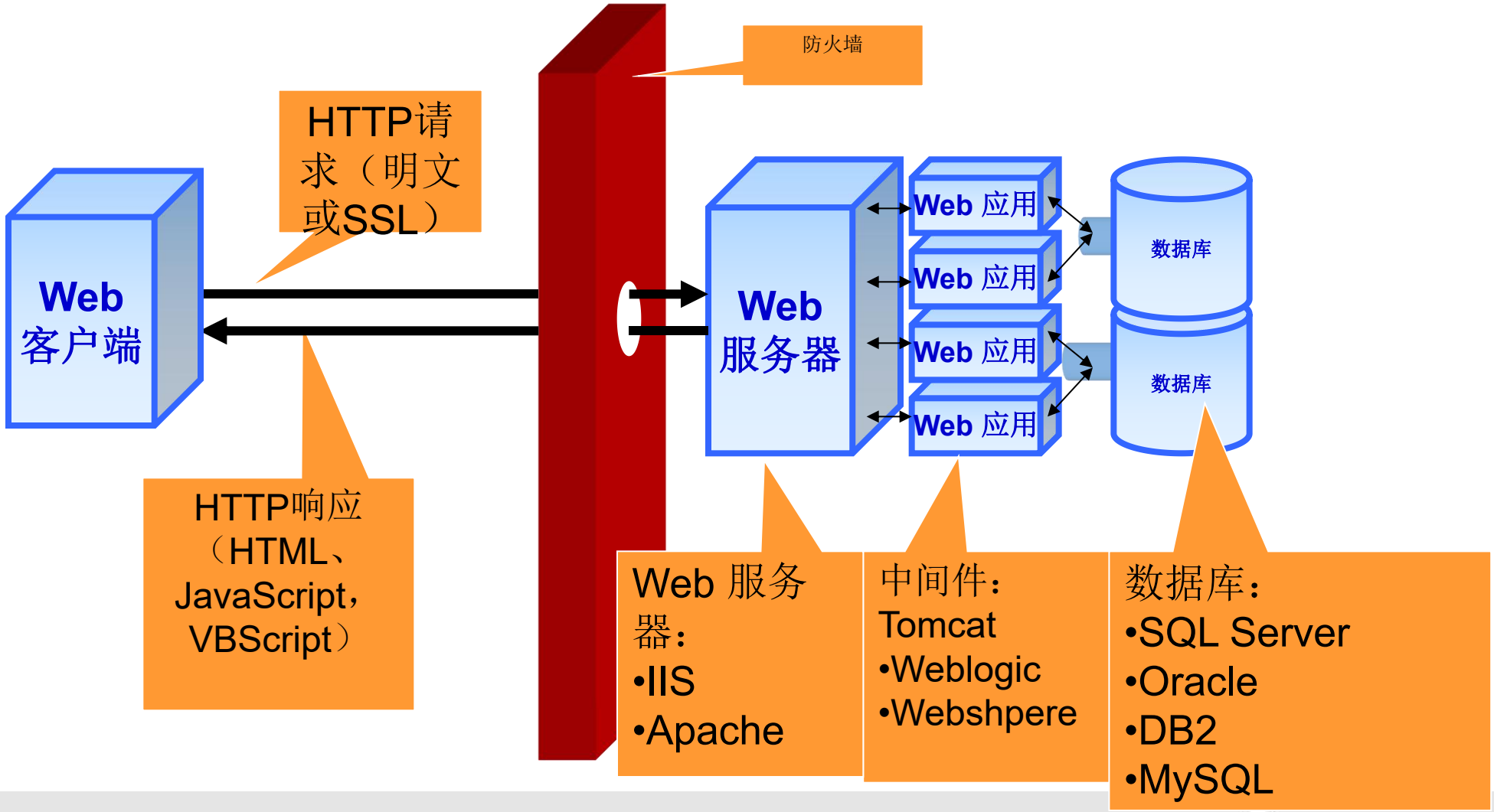
SDL	文档丰富，维护更新及时	较多工具支持	适合大型企业
BSI接触点	强调开发安全重点	注重实用方法	上手容易
BSIMM	最佳实践参考	他山之玉	不强制实践
CLASP	轻量级过程；	以角色及其职责为核心	适合小型企业
SAMM	开放框架	安全知识要求较低	和BSIMM的安全活动能对应
CMMI	自动的、可扩展的框架	集成化框架，消除了各个模型的不一致性	持续改进，就可克服软件开发中困难

软件安全漏洞

- 安全漏洞：在硬件、软件、协议的具体实现过程中存在的缺陷，从而可以使攻击者能够在未授权的情况下访问或破坏系统。



Web应用系统体系架构



网站安全常见问题

- 网络层面
  - 拒绝服务、IP欺骗、ARP欺骗、嗅探
- 系统层面
  - 软件框架漏洞攻击(Struts2、OpenSSL)、配置错误
- 应用层面
  - 代码缺陷（SQL注入、XSS、CSRF ……）
  - 信息泄露
  - 拒绝服务、CC攻击
  - 钓鱼、业务流程缺陷

近期网站安全案例分析 – 携程支付日志泄漏

- 2014.3.22 乌云曝光携程的一个漏洞会导致大量用户银行卡信息泄露，而这些信息可能直接引发盗刷等问题。这一消息很快通过媒体广为流传~
- 其中泄露的信息包括用户的：

持卡人姓名

持卡人身份证

所持银行卡类别

所持银行卡卡号

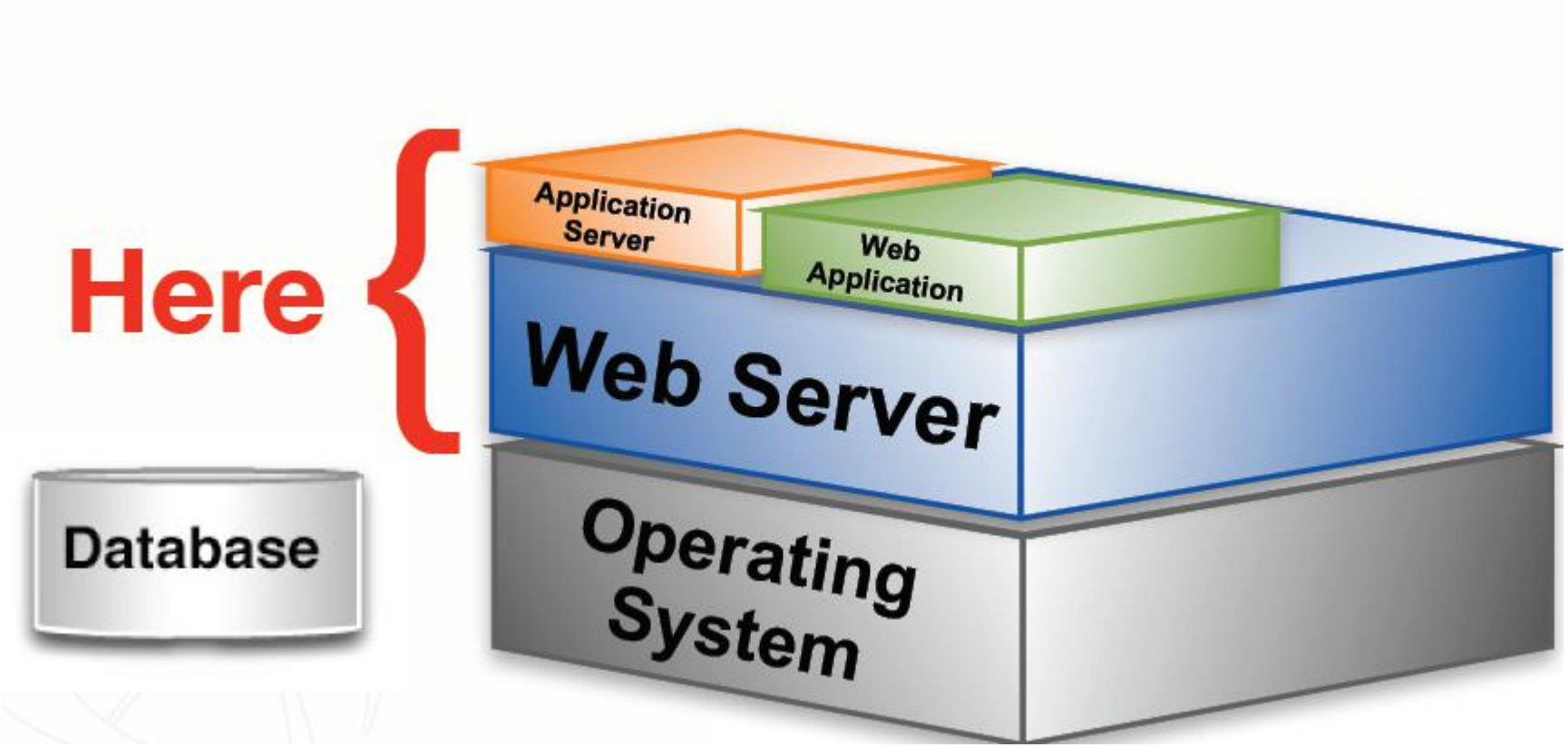
所持银行卡CVV码

所持银行卡6位Bin(用于支付的6位数字)

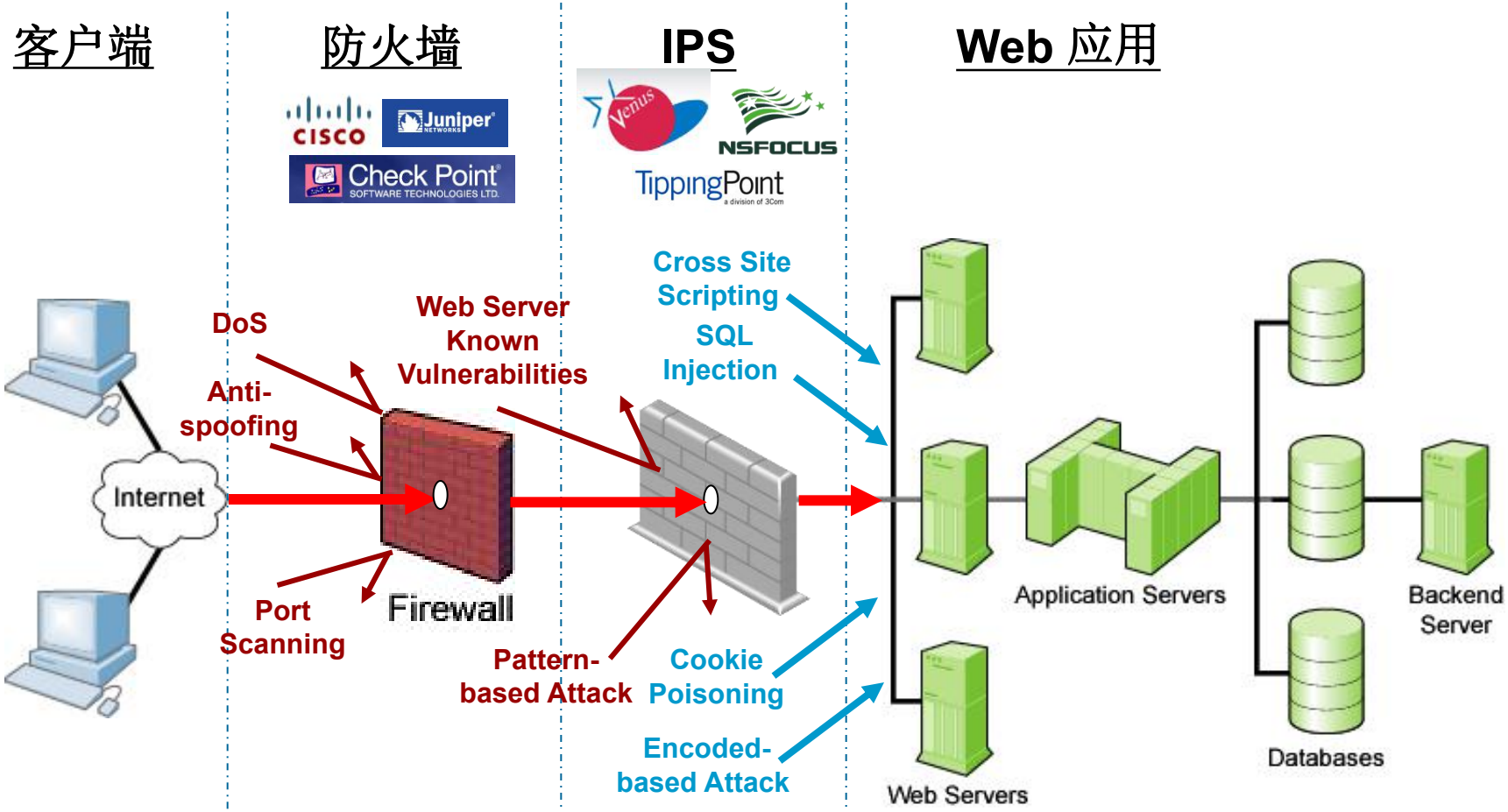
近期网站安全案例分析 – OpenSSL“心脏出血”

- 2014.4.8 OpenSSL曝出名为“Heartbleed（心脏出血）”的漏洞，黑客可利用该漏洞盗取HTTPS服务器的随机64K内存数据，这部分内存中可能包含账号密码cookie等信息，严重威胁网银、支付、网购、邮箱等应用安全。

Web应用安全在哪？


Here
Application Server
Web Application
Web Server
Operating System
Database

Web 应用安全防护



不能通过网络层的保护 (Firewall, IPS, SSL)来阻止和保护应用层的攻击

OWASP

- OWASP（Open Web Application Security Project）是一个开源的、非盈利的全球性安全组织，致力于应用软件的安全研究。其使命是使应用软件更加安全，使企业和组织能够对应用安全风险作出更清晰的决策。
  - http://www.owasp.org
  - 目标是推动安全标准、安全测试工具、安全指导手册等应用安全技术的发展。
  - 全球有140多个分会，近4万名会员。
  - Top Ten
  - WebGoat
  - WebScarab
  - OWASP Testing Guide
  - OWASP Code Review Guide
  - OWASP Development Guide

OWASP 在业界影响

- 国际信用卡数据安全技术PCI标准更将其列为必要组件。
- 美国联邦贸易委员会(FTC)强烈建议所有企业需遵循OWASP所发布的十大Web弱点防护守则。
- 为美国国防信息系统局（DISA)应用安全和开发清单参考。
- 为美国国家安全局/中央安全局，可管理的网络计划提供参考。
- 为欧洲网络与信息安全局（ENISA)，云计算风险评估提供参考。
- OWASP TOP 10为IBM APPSCAN、HPWEBINSPECT等扫描器漏洞参考的主要标准。

网站攻击主要手段介绍
（OWASP Top10 十大应用威胁）

OWASP Top 10 – 2010 (旧版)	OWASP Top 10 – 2013 (新版)
A1 – 注入	A1 – 注入
A3 – 失效的身份认证和会话管理	A2 – 失效的身份认证和会话管理
A2 – 跨站脚本（XSS）	A3 – 跨站脚本（XSS）
A4 – 不安全的直接对象引用	A4 – 不安全的直接对象引用
A6 – 安全配置错误	A5 – 安全配置错误
A7 – 不安全的加密存储—与A9合并成为➔	A6 – 敏感信息泄漏
A8 – 没有限制URL访问—扩展成为➔	A7 – 功能级访问控制缺失
A5 – 跨站请求伪造 （CSRF）	A8 – 跨站请求伪造 （CSRF）
<旧版合并在A6－安全配置错误，新版独立出来>	A9 – 使用已知含有漏洞的组件（新）
A10 – 未验证的重定向和转发	A10 – 未验证的重定向和转发
A9 – 传输层保护不足	与2010年版中的A7合并成为2013年版中的A6

1 注入 - 以 SQL Injection 为例

- 所谓SQL注入，就是通过把SQL命令插入到Web表单递或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令的目的。

· 通过SQL注入，攻击者可以获取数据库信息、篡改数据库、进而控制服务器，是目前非常流行的Web攻击手段。

流行性：常见 危害性：严重

网页中使用如下SQL查询，其中MyUser和MyPassword需要用户输入：

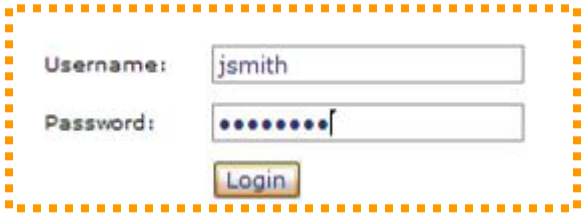
```
SELECT * FROM accounts
WHERE username = 'MyUser' and PASSWORD = 'MyPassword'
```

攻击者可以输入任意用户名，并输入' or '1'='1 作为密码，实现如下查询：

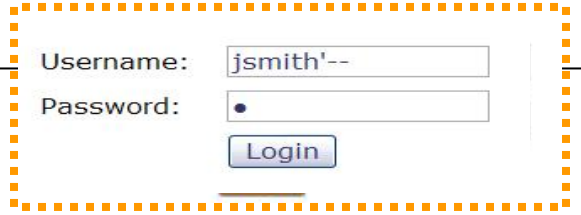
```
SELECT * FROM accounts
WHERE username =  'admin' and PASSWORD = '' or '1'='1'
```

SQL Injection in Code

```
String query = "SELECT * FROM users WHERE name= '" +
   userName + "' AND pwd= '" + pwd + "' ";
```




```
SELECT * FROM users WHERE name='jsmith' AND pwd='test1234'
```



I'm in!

```
SELECT * FROM users WHERE name='jsmith'--' AND pwd='a'
```

SQL注入的后果

- 通常的后果
  - ◆泄露一些重要信息
  - ◆数据库信息
  - ◆表信息
  - ◆列信息
- 数据导出
  - ◆导出库中的数据
- 控制主机
  - ◆控制运行数据库的主机

SQL 注入 主要防范措施

- 严格检查用户输入，注意特殊字符：

  ; < > * | ` & $! # ( ) [ ] : { } ‘ “ ../ ..\ --

  and or script select 等

- 使用“白名单”的规范化的用户的输入
- 转义用户输入内容
- 谨慎使用存储过程(stored procedures)
- 防止错误页面信息泄露
- 使用安全的API，可以参考ESAPI项目

2 失效的验证和会话管理

用户凭证和Session ID是Web应用中最敏感的部分，也是攻击者最想获取的信息。攻击者会采用网络嗅探、暴力破解、社会工程等手段尝试获取这些信息。

流行性：广泛 危害性：严重

1. 某航空票务网站将用户Session ID包含在URL中：

http://example.com/sale/saleitems;sessionid=2P0OC2JDPXM0O

QSNDLPSKHCJUN2JV?dest=Hawaii

一位用户为了让她朋友看到这个促销航班的内容，将上述链接发送给朋友，导致他人可以看到她的会话内容。

2. 一位用户在公用电脑上没有登出他访问的网站，导致下一位使用者可以看到他的网站会话内容。

3. 登录页面没有进行加密，攻击者通过截取网络包，轻易发现用户登录信息。

2 失效的验证和会话管理

主要防范措施：

用户密码强度（普通：6字符以上；重要：8字符以上；极其重要：使用多种验证方式）

不使用简单或可预期的密码恢复问题

登录出错时不给过多提示

密码传输时采用加密形式传输

登录页面需要SSL加密

对多次登录失败的帐号进行短时锁定

设置会话闲置超时

保护Cookie

不在URL中显示Session ID

3 跨站脚本 - Cross-Site Scripting（XSS）

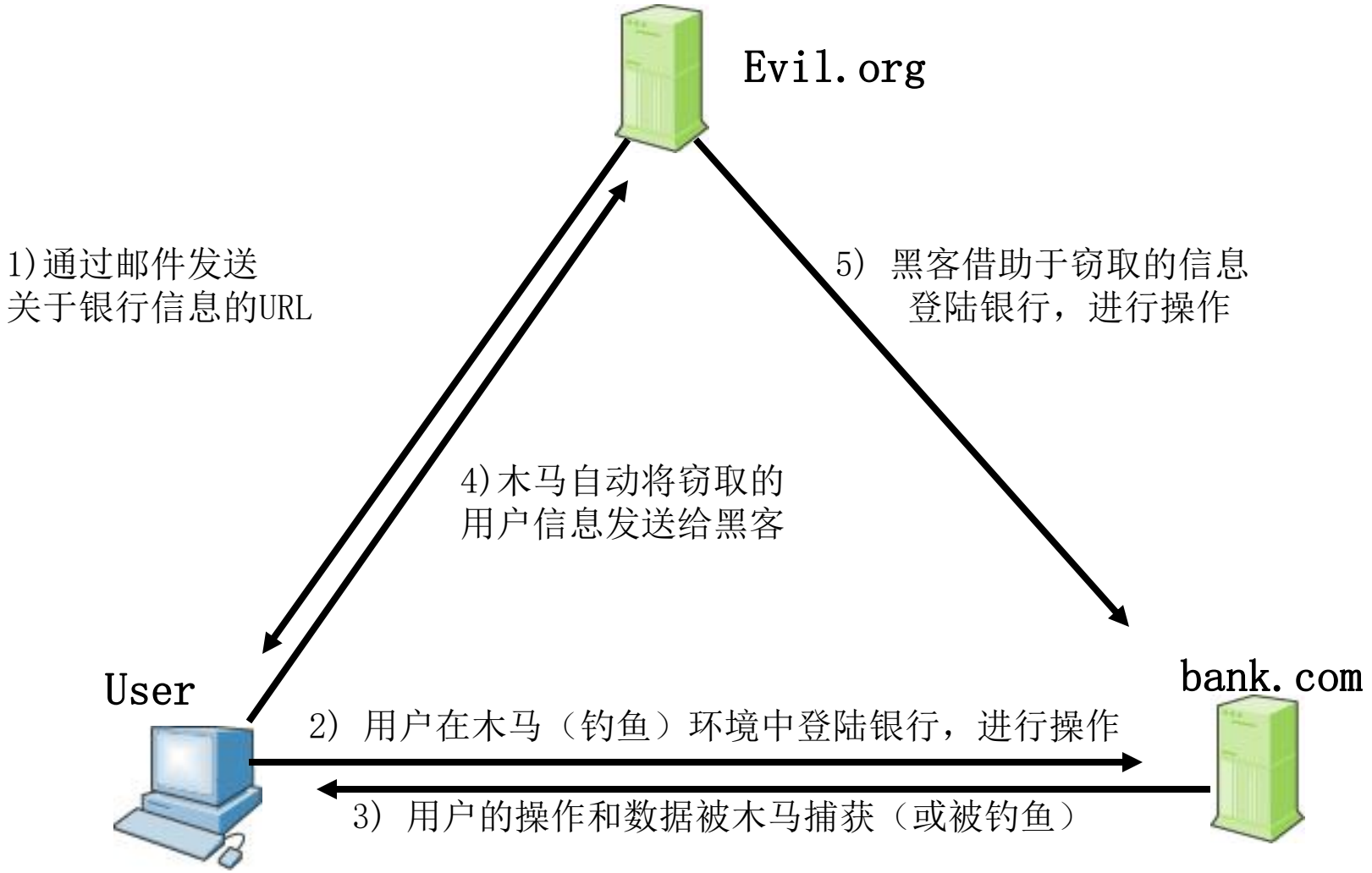
影响面最广的Web安全漏洞。

攻击者通过向Web页面里插入恶意html代码，当用户浏览该页之时，嵌入其中Web里面的html代码会被执行，从而达到恶意用户的特殊目的。

可分为三种类型：反射、存储和DOM

- 流行性：非常广泛 危害性：中等

XSS常见攻击过程


Evil.org
1)通过邮件发送
关于银行信息的URL
5) 黑客借助于窃取的信息
登陆银行，进行操作
4)木马自动将窃取的
用户信息发送给黑客
User
bank.com
2) 用户在木马（钓鱼）环境中登陆银行，进行操作
3) 用户的操作和数据被木马捕获（或被钓鱼）

XSS的危害

- 盗取cookie信息
- 篡改用户浏览的正常内容，伪造页面信息
- 把你重定向到一个恶意站点，网站挂马
- 利用浏览器的漏洞控制你的机器

跨站脚本 XSS 主要防范措施

- 严格检查用户输入。

- 尽量限制在HTML代码中插入不可信的内容（可被用户输入或修改的内容）。

- 对于需要插入的不可信内容必须先进行转义（尤其对特殊字符、语法符合必须转义或重新编码）。

4 不安全的直接对象引用

服务器上具体文件名、路径或数据库关键字等内部资源被暴露在URL或网页中，攻击者可以此来尝试直接访问其他资源。

- 所有Web应用都会受此问题影响。

流行性：常见　危害性：中等

- 示例

某网站的新闻检索功能可搜索指定日期的新闻，但其返回的URL中包含了指定日期新闻页面的文件名：

http://example.com/online/getnews.asp?item=20March2003.html

攻击者可以尝试不同的目录层次来获得系统文件win.ini:

http://example.com/online/getnews.asp?item=../../winnt/win.ini

2000年澳大利亚税务局网站曾经发生一位用户通过修改其URL中ABN ID号而直接访问到17000家公司税务信息的事件。

4 不安全的直接对象引用

主要防范措施：

- 避免在URL或网页中直接引用内部文件名或数据库关键字
- 可使用自定义的映射名称来取代直接对象名

http://example.com/online/getnews.asp?item=11

- 锁定网站服务器上的所有目录和文件夹，设置访问权限
- 验证用户输入和URL请求，拒绝包含./或../的请求

5 安全配置错误

管理员在服务器安全配置上的疏忽，或者采用默认配置，通常会导致攻击者非法获取信息、篡改内容，甚至控制整个系统。

流行性：常见　危害性：中等

示例

服务器没有及时安装补丁

网站没有禁止目录浏览功能

网站允许匿名用户直接上传文件

服务器上文件夹没有设置足够权限要求，允许匿名用户写入文件

Web网站安装并运行并不需要的服务，比如FTP或SMTP

出错页面向用户提供太过具体的错误信息，比如call stack

Web应用直接以SQL SA帐号进行连接，并且SA帐号使用默认密码

SQL服务器没有限制系统存储过程的使用，比如xp_cmdshell

5 安全配置错误

主要防范措施：

安装最新版本的软件及补丁

最小化安装（只安装需要的组件）

Web文件/SQL数据库文件不存放在系统盘上不在Web/SQL服务器上运行

其他服务

严格检查所有与验证和权限有关的设定

权限最小化

不使用默认路径和预设帐号

按照最佳安全实践进行加固配置

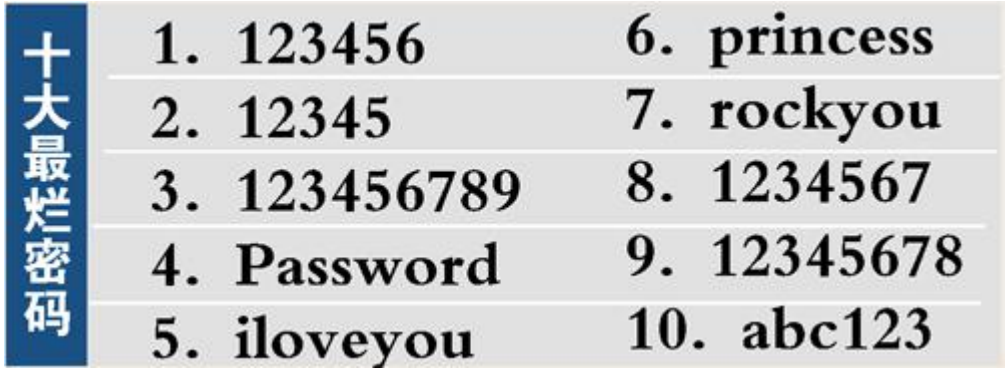
6 敏感信息泄漏

对重要信息不进行加密处理或加密强度不够，或者没有安全的存储加密信息，或者在传输的时候没有采用加密措施，都会导致攻击者获得这些信息。

此风险还涉及Web应用以外的安全管理。

流行性：少见　危害性：严重

对于重要信息，如密码等，直接以明文写入数据库使用自己编写的加密算法进行简单加密,简单地使用MD5, SHA-1等散列算法将加密信息和密钥存放在一起



6 敏感信息泄漏

- 主要防范措施：
  - 对所有重要信息进行加密
  - 使用MD5，SHA-1等Hash算法时候采用加盐存储的方式
  - 产生的密钥不能与加密信息一起存放
  - 严格控制对加密存储的访问

7　功能级访问控制缺失

某些Web应用包含一些“隐藏”的URL，这些URL不显示在网页链接中，但管理员可以直接输入URL访问到这些“隐藏”页面。如果我们不对这些URL做访问限制，攻击者仍然有机会打开它们。

流行性：常见　危害性：中等

示例

1. 某商品网站举行内部促销活动，特定内部员工可以通过访问一个未公开的URL链接登录公司网站，购买特价商品。此URL通过某员工泄露后，导致大量外部用户登录购买。

2. 某公司网站包含一个未公开的内部员工论坛（http://example.com/bbs），攻击者可以进行一些简单尝试就找到这个论坛的入口地址。

7 功能级访问控制缺失

主要防范措施：

对于网站内的所有内容（不论公开的还是未公开的），都要进行访问控制检查

只允许用户访问特定的文件类型，比如.html，.asp，.php等，禁止对其他文件类型的访问

考虑一下管理权利的过程并确保能够容易的进行升级和审计

8 跨站请求伪造 -（CSRF）

攻击者构造恶意URL请求，然后诱骗合法用户访问此URL链接，以达到在Web应用中以此用户权限执行特定操作的目的。

· 和XSS的主要区别是：XSS的目的是在客户端执行脚本；

CSRF的目的是在Web应用中执行操作。

流行性：常见　危害性：中等

· 案例:TP-LINK路由器DNS篡改

Internet Explorer 3.0 到 6.0 版本支持以下 HTTP 或 HTTPS URL 语法：

http(s)://username:password@server/resource.html

1、黑客制作一个国内各种路由的默认IP和默认密码构造一个Http Authentication Url暴力登陆脚本。

2、使用CSRF修改路由的DNS，指向恶意DNS

8 跨站请求伪造 （CSRF）

主要防范措施：

避免在URL中明文显示特定操作的参数内容使用同步令牌（Synchronizer Token），检查客户端请求是否包含令牌及其有效性

检查Referer Header，拒绝来自非本网站的直接URL请求

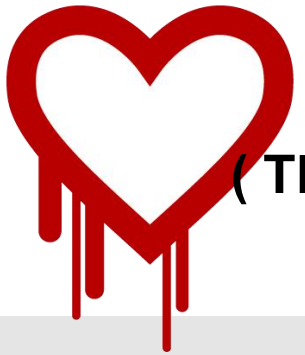
9 使用含有已知漏洞的组件

- 应用程序使用带有已知漏洞的组件会破坏应用程序防御系统，并使一系列可能的攻击和影响成为可能。

流行性：广泛　危害性：中等

案例

（CVE-2013-2251）



CVE-2014-0160
（TLS心跳读远程信息泄露漏洞）

9 使用含有已知漏洞的组件

- 主要防范措施：

  时刻关注组件的安全漏洞信息

  建立组件使用的安全策略，比如需要某些软件开发实践，通过安全性测试和可接受的授权许可

  在适当的情况下，考虑增加对组件的安全封装，去掉不使用的功能和或者组件易受攻击的方面

10 未验证的重定向和跳转

攻击者可能利用未经验证的重定向目标来实现钓鱼欺骗，诱骗用户访问恶意站点。

攻击者可能利用未经验证的跳转目标来绕过网站的访问控制检查。

流行性：少见　危害性：中等

示例

利用重定向的钓鱼链接：

http://example.com/redirect.asp?=http://malicious.com

更为隐蔽的重定向钓鱼链接：

http://example.com/userupload/photo/324237/../../../redirect.asp%3F%3Dhttp%3A//www.malicious.com

利用跳转绕过网站的访问控制检查：

http://example.com/jump.asp?fwd=admin.asp

10 未验证的重定向和跳转

主要防范措施：

尽量不用重定向和跳转

对重定向或跳转的参数内容进行检查，拒绝站外地址或特定站内页面

不在URL中显示目标地址，以映射的代码表示（http://example.com/redirect.asp?=234）

常见编程安全问题

整型数据操作

- 整数已日益成为C程序中被低估的漏洞来源，整数的边界溢出问题通常被有意地忽略了
  - 想当然的认为整数的表示范围已经够用
  - 无法接受对每个算术操作的结果都进行检测所带来的代价。
- 危害：小到程序崩溃以及逻辑错误，大到权限提升以及任意代码执行

预备知识：整型数据存储

- 以32位字长为例：

short型

01	11	11	11	11	11	11	11
10	00	00	00	00	00	00	00

最大:+32767

最小:-32768

unsigned short型

11	11	11	11	11	11	11	11
00	00	00	00	00	00	00	00

最大 65535

最小: 0

int型

01	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

最大:+2147483647

最小:-2147483648

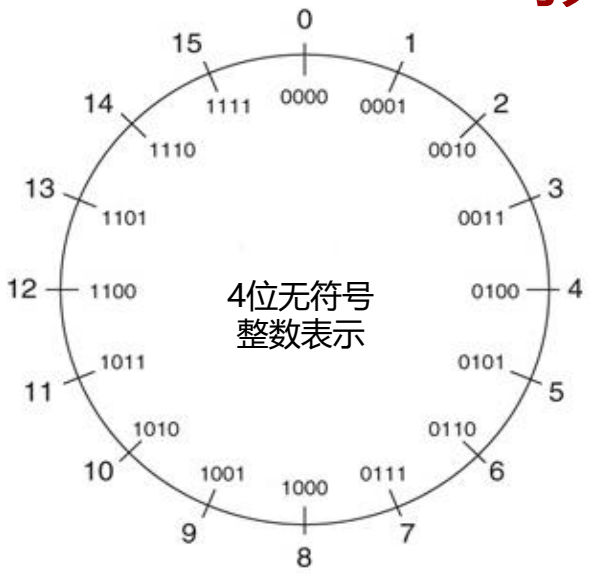
unsignedint型

01	11	11	11	11	11	11	11	11	11	11	11	11	11	11	11
10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

最大：4294967275

最小: 0

预备知识：整数回绕与溢出




```
unsigned int i=1,sum;
if(sum+i>UINT_MAX)
    too_big();                    //不会发生，因为sum+i回绕了
else
    sum +=i;
```

例如：

```
    unsigned int ui;
    ui=UINT_MAX;                  //例如，在x86-32上，4 294
967 295
    ui++;
    printf("ui=%u\n",ui);
    ui=0;
    ui--;
    printf("ui=%u\n",ui);
```

整数赋值错误问题

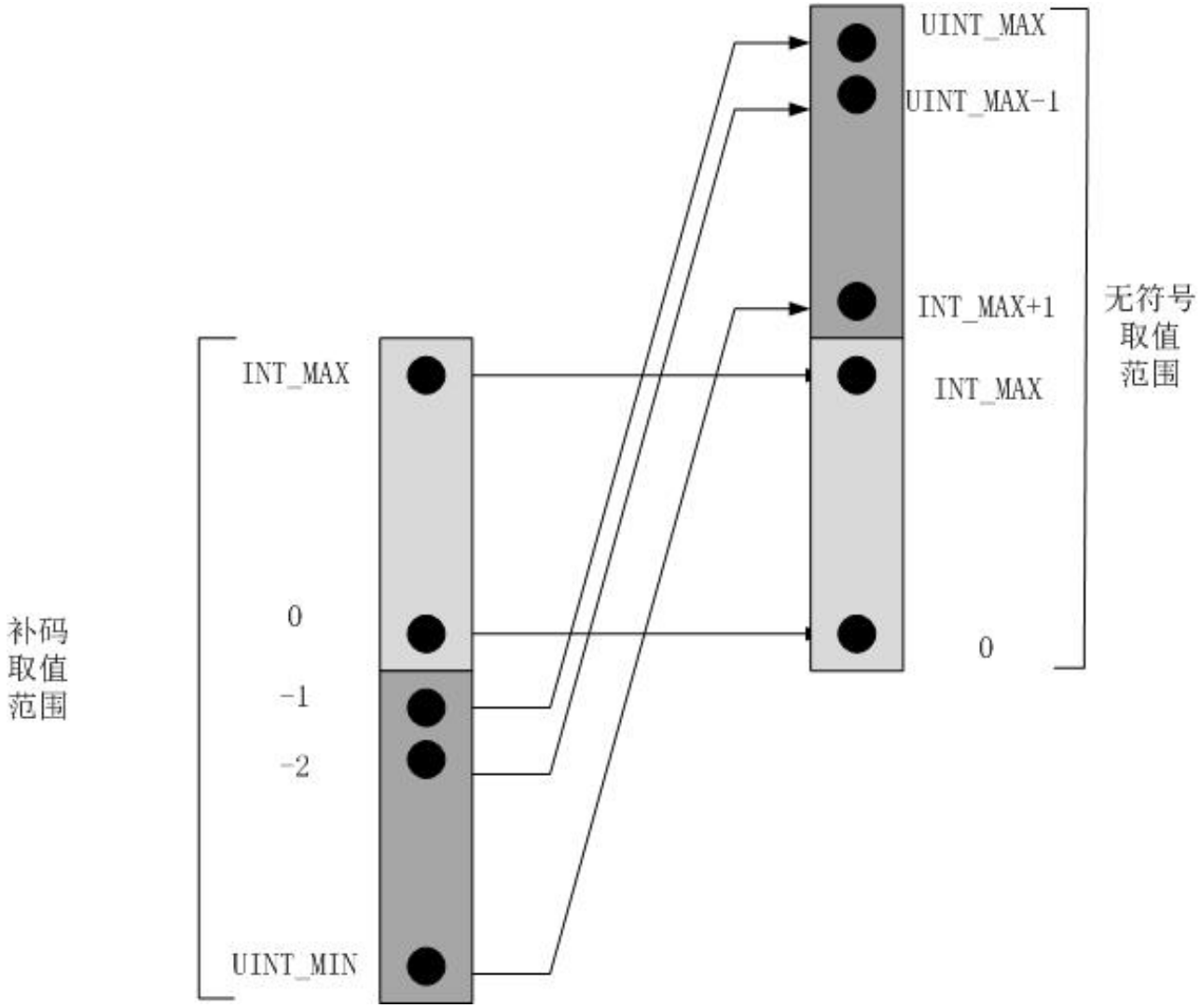
•问题

带符号整型转换成无符号整型时，会发生什么？

例：int i=-3;

unsigned short ui = i;

//ui=65533



■ 结论：

- 有符号=>无符号,必须验证范围

带符号与无符号整型比较问题

- 这段代码有什么问题?

```
unsigned int ui=ULONG_MAX;
char c = -1;
printf("%d",c==ui);
```

- 输出结果:

  1

- 结论:

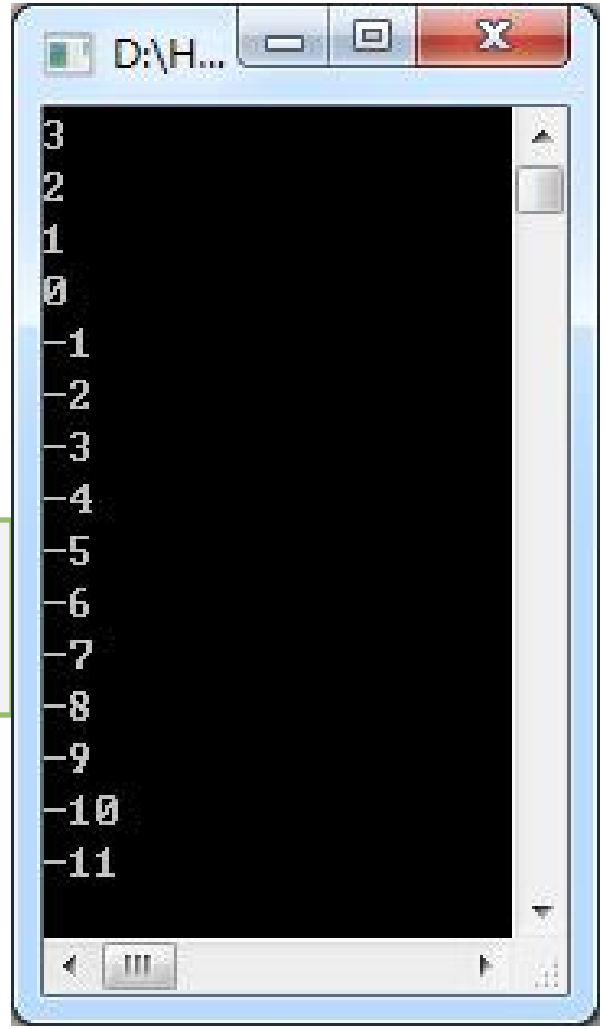
  带符号整型与无符号整型进行比较会发生整型类型隐式转换。带符号整型将转换成相应的无符号整型。

size_t导致的死循环

- 下面这段代码的输出结果是什么？

```
size_t s = sizeof(int);
  while(--s >= 0)
{
    printf("%d",s);
}
```

//纠错建议：
while(--s > 0)



- 总结：
  - size_t为无符号数，仍是无符号转换成带符号数惹的祸！

整型提升导致的内存溢出错误

- 不能长度的整型之间进行运算会发生什么？

例：

```
char c1='a';
char c2='b';
char c3=c1+c2;
```

例：

```
signed char c,c1=100,c2=3,c3=4;
c=c1*c2/c3;
```

■总结：

整型数据参与运算时，如果参与运算的所有整型数据都小于int类型，那么小于int类型的数据都自动提升为int；如果参与运算的所有整数数据不都小于int类型，那么小于int类型的数据都自动提升为unsigned int

临时变量溢出

- 下面的JAVA代码正确吗?

- long c=2*Integer.*MAX_VALUE*;

修改建议:
long c=(long)2*Integer.*MAX_VALUE;*

- 总结:

算术运算操作时，可能会造成存储中间值的临时变量溢出，从而导致运算结果错。

误用short引起缓冲区溢出

- 思考下面的代码运行情况。

```
void test(char *c)
{
   short s = strlen(c);
   char buf[256];
   cout<<"strlen(c)="<<strlen(c)<<endl;
   cout<<"s ="<<s<<endl;
   cout<<"max_buf=256"<<endl;
   if(s < 256)
{
strcpy(buf,c);
}
else
{
      cout<<"overflow!"<<endl;
}
}
void main()
{
   char *c = new char[35000];
   memset(c,'c',35000);
   c[35000-1] = '\0';
   test(c);
}
```

//纠错建议：
size_t s = strlen(c);

输出结果：
strlen(c)=34999
s =-30537
max_buf=256

//纠错建议：
//大于short的最大值

表达式中对同一变量多次写入问题

- 示例

```
int num = 18;

num = ((28 * ++num) * (num = get())) + (num > th[0] ? 0 : -2);
```

```
/* 纠错建议： */
num = ((28 * (num+1)) * anum) + (num > th[0] ? 0 : -2);
```

空字符结尾错误问题

- 示例
  - ◆char s1[5],s2[5],s3[10];
  - ◆strcpy (s1 , "abcde");
  - ◆strcpy (s2 , "abcde");
  - ◆strcpy (s3 , s1);
  - ◆strcat (s3 , s2);

```
/* 纠错建议： */
char s1[6];
char s2[6];
char s3[11];
```

‘a’	‘b’	‘c’	‘d’	‘e’	‘\0’

◆总结：C和C++中字符串常量以"\0"结束。

无界字符串复制问题

```
#include <stdio.h>
#include <stdlib.h>
void get_out_of_bound(void) {
  char source[8];
  puts("Please input a string: ");
  gets(source);
  return ;
}
void main(){
   get_out_of_bound();
}
```

```
/* 纠错建议： */
fgets(source,sizeof(source),stdin);
```

定长字符串越界问题

```
char source1[5];
char source2[ ] = "abcde";
char  *p;
int  i;
strcpy (source1 , source2);
p = (char *)malloc(strlen(source1))
for (i = 1; i<= 6; i++)
{
  p [ i ] = source1 [ i ];
}
p[ i ] = '\0';
printf("p = %s" , p);
```

```
/* 纠错建议： */
char * source1= (char *)malloc(strlen(source2)+1);
```

```
/* 纠错建议： */
p = (char *)malloc(strlen(source1)+1);
```

```
/* 纠错建议： */
for (i = 0; i<5; i++)
```

字符串截断问题

- 下面的代码十分安全?

◆char str[10];

◆cin >>str;

◆cout << str << endl;

/* 纠错建议： 在cin >>str;前添加*/
cin.width(10);

- 结论：

当源字符串内容长度大于目标字符串内容长度时，源字符串向目标字符串拷贝会发生字符串截断错误

与函数无关的字符串错误问题

```
void str_copy( char* pstr)
{
char str[10];
int  index = 0;
while (pstr[index] != '\0')
{
    str[index] = pstr[index];
    index++;
}
str[index] = '\0';
printf("str = %s\n" , str);
}
```

```
/* 纠错建议：在while循环复制之前首先判断源串长度*/
if(strlen(pstr)>= sizeof(str))
{
  /*进行相应操作*/
}
else
{ /*进行while循环操作 */ }
```

字符串比较错误

- 下面代码的输出结果为：

```
char  *str1 = "abcde";
char  str2[6];
 strcpy(str2 , str1);
if (str1 == str2)
   printf("1\n");
else
   printf("0\n");
```

```
/* 纠错建议*/
if (*str1 ==*str2)
```

JAVA例子

```
String  name1 = new String("tom");

String  name2 = new String("tom");

if(name1 == name2)

{

  System.out.println("name1==name2");

}

else

{

  System.out.println("name1 !=name2");

}
```

/* 纠错建议： */
if(name1.equals(name2))

数组越界问题

- 下面代码的输出结果为：

```
int  array[ ]={34 , 44 , 21 , 90};
for(int index = 0 ; index<=array.length ; index++)
{
   System.out.println(array[index]);
}
```

```
/* 纠错建议*/
for(int index = 0 ; index<array.length ; index++)
```

数组定义和值初始化括号形式混淆错误

- 下面代码的输出结果为：

```
int  *array = new  int(10);
for(int  index = 0 ; index <10 ; index++)
{
   array[index] = index * index;
}
delete []array;
```

```
/* 纠错建议*/
int  *array = new  int[10];
```

未正确区分标量和数组问题

- 下面代码的输出结果为：

```
Obj *obj = new Obj[5];
delete obj;
obj = new Obj(5);
delete[] obj;
```

```
/* 纠错建议*/
Obj *obj = new Obj(5);
delete obj;
obj = new Obj[5];
delete[] obj;
```

- 结论：
  - ◆标量的分配和释放，使用new和delete操作符
  - ◆数组的分配和释放，使用new[]和delete[]操作符

二维数组的内存泄露

- 下面代码的输出结果为：

```
int  **array = new  int* [5];
for(int  index = 0 ; index <5 ; index++)
{
   array[index] = new  int[4];
}
delete [ ] array;
```

```
/* 纠错建议*/
for(int  cols = 0 ; cols <4 ; cols++)
{
   delete [ ] array[cols];
}
```

结论：

◆对二维数组的内存空间释放，不仅要对行进行释放，而且也要对列进行释放

释放指针指向的对象引起内存泄漏

```
class  Person{
  public:
   int  age;
   Person() : age(23){ }
};
int  main(){
   vector<Person*>  personVector;
   for( int  i = 0 ; i<5 ; i++){
     personVector.push_back(new Person);
}
for( int  i = 0 ; i< personVector.size() ; i++){
     cout << (personVector[i])->age << endl;
}
personVector.clear();

}
```

```
/* 纠错建议：  */
/*1.编写函数*/
void  deletePersonVector(std::vector
<Person*>& p)
{
for( int  i = 0 ; i<p.size() ; i++)
{
    delete p[i];
}
}
/*2.将personVector.clear();修改为*/
deletePersonVector(personVector);
```

指针变量的传值和传址混淆问题溢出

```
void  change(int* q)
{
  if(q == NULL){
     q = new int(20);
    }
}
int  main()
{
   int  *p = NULL;
   change(p);
   printf("%d\n" , *p);
}
```

```
/* 纠错建议： */
void  change(int *&q)
```

验证方法参数问题

- 下面的代码是否存在缺陷?

```
private Object state = null;

void setState(Object state)

{

  this.state = state;

}

void useState()

{

    // perform some operate here

}
```

```
/* 纠错建议：赋值之前验证state*/
if(state == null)
{
    // Handle null State
    return;
}
if(!isInvalidState(state))
{
    // Handle Invalid State
    return;
}
```

函数退出时内存未释放问题

- 下面的代码是否存在缺陷?

```
int test(char *src)
{
char  *dest = new char[10];
if(src ==NULL)
{
   return 0;
}
strncpy(dest , src , 10);
delete [ ] dest;
return 1;
}
```

```
/* 纠错建议： return之前*/
   delete [ ] dest;
```

程序异常退出时未关闭已打开文件

```
const char *file1 = "test.txt";
  const char *file2= "test.bat";
  FILE *fp1 = fopen(file1, "r");
  if(!fp1)
{
   fprintf(stderr, "Can not open %s.\n",file1);
   return -1;
}
FILE *fp2 = fopen(file2, "r");
  if(!fp2)
{
   fprintf(stderr, "Can not open %s.\n",file2);

   return -1;
}
fclose(fp1);
fclose(fp2);
```

```
/* 纠错建议：此处添加*/
 fclose(fp1);
```

写文件没有调用fflush

代码：

```
int main(){
  FILE *file = fopen("test.txt","w");
fprintf(file, "%s", "hello world");
fclose(file);
return 0;
}
```

```
/* 纠错建议:在fclose之前添加*/
fflush(file);
```

临时文件未删除问题

```
public class TestTempFile
{
public static void main(String[] args) throws IOException
{
   File tempFile = new File("tempfile.tmp");
   if(tempFile.exists())
{
      System.out.println("The tempfile.tmp already exists. ");
      return;
   }
   FileOutputStream fos = null;
   try
{
      fos = new FileOutputStream(tempFile);
      String str = "data";
      fos.write(str.getBytes());
…
   }
```

```
finally
{
      if(fos != null)
      {
          try
           {
              fos.close();
           }
           catch(IOException e)
           {
                // handle error
           }
       }
   }
}
}
```

```
/* 纠错建议：此处添加*/
  if(tempFile.exists())
        tempFile.delete();
```

敏感信息硬编码问题

```
class IPAddressTest
{
    String ipAddress = "178.18.254.1";
    public static void main(String[] args)
    {
    …
    }
}
```

```
/* 纠错建议:将信息代码保存在属性文件中*/
String ipAddress =
ResourceBundle.getBundle("message").getString("ip");
```

引用未初始化的内存错误问题

```
void  init_func(int  *q , int  size)
{
    int  *p = (int *)malloc(size * sizeof(int));
    for(int i = 0 ; i<size ; i++)
    {
      p[i] += q[i];
    }
    …
}
```

```
/* 纠错建议:*/
int  *p = (int *)calloc(sizeof(q), sizeof(int));
```

- **总结：** malloc函数分配的内存并未进行初始化，未初始化的内存可能导致信息泄漏的潜在风险

引用已释放内存的错误问题

```
for(p = head ; p ! =NULL ; p = p->next：
{
    free(p);
}
```

```
/* 纠错建议:*/
for(p = head ; p ! =NULL ; p = q){
    q = p->next;
   free(p);
}
head = NULL;
```

不匹配的内存管理函数问题

```
int *number = new int(10);
free(number)
number = static_cast<int*> (malloc(sizeof(int)));
delete number;
```

/* 纠错建议:
delete number

/* 纠错建议:
free(number);

- 总结:

malloc/free函数与new/delete操作符必须正确配对使用

在嵌套类中暴露外部类的私有字段问题


```
class Point
{
    private int x;
    private int y;
    public class Operate
    {
      public void getPoint()
    {
        System.out.println("x=" + x + ", " + "y=" + y);
    }
    }
    }
    public class TestPoint
    {
    public static void main(String[] args)
    {
      Point p = new Point();
      Point.Operate po = p.new Operate();
      po.getPoint();
    }
}
```

```
/* 纠错建议:*/
private void getPoint()
```

构造函数中抛出异常引发错误

```
class  Person
{
 private :
    char  *name;
    int  age;
 public:
    Person()
    {
    name = new char[10];
    age = -1;
    if(age<0||age>100)
    {
      //纠错建议：  delete[ ]  name;
    throw  exception("age is error.");
    }
    };
    ~Person()
    {
        delete[ ]  name;
    }
};
```

```
int  main()
{
      try
      {
      Person  per;
      }
      catch(const exception& e)
      {
      cout << "异常捕获成功" <<endl;
      }
}
```

/* 纠错建议:*/
在抛出异常之前，对字符数组name进行释放

字符乱码问题

■Web应用程序中，经常会出现一些乱码问题，下面列出常见的几种乱码情况：

1) 字符串在解码时的字符集与编码字符集不一致时，汉字会变为乱码。

2) 中文在经过不支持中文的ISO-8859-1编码后，其字符会变成"?"

■ 代码：

```
InputStream is = System.in;

System.out.println((char)is.read());
```

```
/* 纠错建议:*/
InputStreamReader is = new
InputStreamReader(System.in);
```

功能级别访问控制缺失问题

- Web UI设计时，功能级别的验证是通过用户的访问权限而定，应用程序需要在每个功能被用户访问时，都要在服务端验证该用户是否拥有其访问控制权限。如果忽视了验证步骤，攻击者就可能伪造请求，在未经授权的情况下访问这些功能。

- 示例

攻击方法：

1. Hacker访问http://testing.com/test/index.jsp页面；
2. Hacker打开index.jsp页面的源代码发现按钮被disabled后，将该页面源代码中的disabled属性去掉后，将新的页面另存在自己的容器中；
3. 启动自己的容器并访问http://hackservice.com/my/index.jsp，打开新页面后，发现删除功能可以使用；
4. Hacker点击删除按钮，完成删除功能操作

...

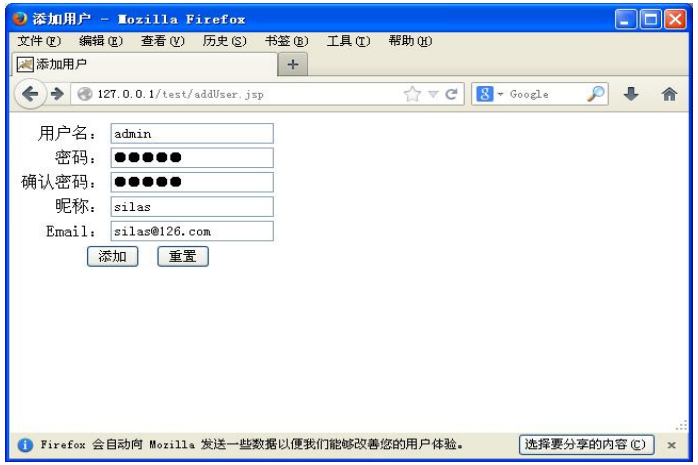
```
<input type="button" value="delete" <% if(power==1) out.write("disabled"); %>>
```

- 纠错建议：

  如果用户没有某个功能访问权限，应将按钮设计为不可见。

表单重复提交问题

■表单重复提交问题主要体现在几个方面：一是提交表单后在当前页面点击浏览器上的"后退"按钮，此时会回到之前的页面，然后再次提交表单造成重复提交；二是用户提交表单后，在当前页面点击浏览器上的"刷新"按钮或者直接按F5功能键完成重复提交；三是通过恶意程序来重复发送恶意请求。



纠错建议是使用令牌（token）机制

Step 1：　初始化token值

```
…
String tokenValue = GenerateToken.getToken();
request.getSession().setAttribute("com.dhee.Token", tokenValue);
…
```

Step 2：　在添加页面中加入token的隐藏域，用于后面步骤判断

```
…
<input type="hidden" name="TokenValue" value="${sessionScope['com.dhee.Token']}">
…
```

Step 3　提交表单后，对token值进行判断，判断过后，需要对token值重新生成并保存

```
…
   String token = request.getParameter("TokenValue");
   String tokenValue = (String) request.getSession().getAttribute("com.dhee.Token");
   request.getSession().setAttribute("com.dhee.Token", GenerateToken.getToken());
   if(token.equals(tokenValue))
   {
                        // 令牌相等时，业务的逻辑处理
   }
   else
   {
                        // 令牌不相等时，具体业务的逻辑处理
   }
…
```

输入验证和数据合法性校验

程序接收的数据可能来源于：

- 未经验证的用户——
- 网络连接——
- 其他不受信任途径——

在构造访问数据的应用程序时，在得到相反的证实之前，应假定所有用户输入都是恶意的。如果不这样做，您的应用程序很容易受到攻击。

输入数据有效性校验

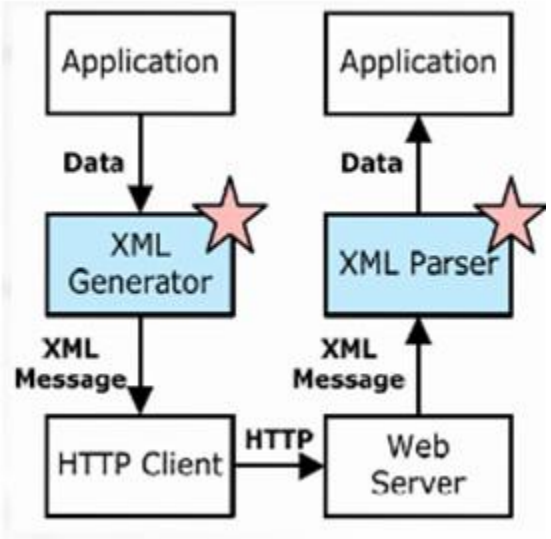
对输入的电子信箱地址和手机号码进行数据有效性校验示例

```
//电子信箱地址格式有效性校验
public static boolean checkEmail(String email) {
  boolean flag = false;
  try {
        String check ="^([a-z0-9A-Z]+[-|\\.]?)+[a-z0-9A-Z]@([a-z0-9A-
  Z]+(-[a-z0-9A-Z]+)?\\.)+[a-zA-Z]{2,}$";
        Pattern regex = Pattern.compile(check);
        Matcher matcher = regex.matcher(email);
        flag = matcher.matches();
  } catch (Exception e) {
        LOG.error("验证邮箱地址错误", e);
        flag = false;
  }
        return flag;
}
```

输入数据有效性校验

```
//手机号码格式有效性校验
public static boolean isMobileNO(String mobiles) {
    boolean flag = false;
    try {
        Pattern p = Pattern.compile("^((13[0-9])|(15[^4,\\D])|(18[0,5-9]))\\d{8}$");
        Matcher m = p.matcher(mobiles);
        flag = m.matches();
    } catch (Exception e) {
        LOG.error("验证手机号码错误", e);
            flag = false;
    }
    return flag;
}
```

避免XML注入



在XML解析的过程中，最常见的有三种漏洞：

- 拒绝服务漏洞
- XML注入
- XML外部实体注入

XML注入


```
1  <?xml version="1.0"?>
2  <order>
3      <quantity>10</quantity>
4      <price>25.00</price>
5      <address>beijing</address>
6  </order>
```

```
1  <?xml version="1.0"?>
2  <order>
3      <quantity><!--</quantity>
4      <price>25.00</price>
5      <address>-->100</quantity><price>0.01</price><address>beijing</address>
6  </order>
```

避免XML注入

对数量quantity进行合法性校验，控制只能传入0-9的数字。

```
if (!Pattern.matches("[0-9]+", quantity)) {
    // Format violation
  }
String xmlString = "<item>\n<description>Widget</description>\n" +
                   "<price>500</price>\n" +
                   "<quantity>" + quantity + "</quantity></item>";
outStream.write(xmlString.getBytes());
outStream.flush();
```