第9章 软件安全实践

推荐的安全实践

北京邮电大学 徐国胜
guoshengxu@bupt.edu.cn

本讲内容安排

1. 安全生命周期
2. 安全需求
3. 安全设计
4. 安全实现
5. 安全验证

1.1 软件生命周期

- 软件危机：指在软件开发和维护中所产生的一系列严重的问题，包括：
  - 如何开发软件，满足用户对软件的需求，
  - 如何维护数量众多的已有软件。
- 软件危机主要表现：

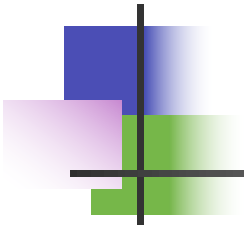
（1）用户需求不明确、变更过多

（2）软件成本日益增长

（3）开发进度难以控制

（4）软件质量差

（5）软件维护困难



- 软件危机产生的原因
  - （1）软件开发无计划性
  - （2）软件需求不充分
  - （3）软件开发过程无规范
  - （4）软件产品无评测手段
- 解决软件危机的途径
  - （1）应该加强软件开发过程的管理
  - （2）推广使用开发软件的成功技术与方法
  - （3）开发和使用好的软件工具

软件工程的产生和发展：为了解决软件危机，人们在软件开发中也不断改进和发展，在50多年中计算机软件开发经历了三个发展阶段：

- 程序设计阶段：约为50至60年代
- 程序系统阶段：约为60至70年代
- 软件工程阶段：约为70年代以后

- 软件工程带来的根本变化：
  - （1）人们改变了对软件的看法。
  - （2）软件的需求是软件发展的动力。
  - （3）软件工作的范围从只考虑程序的编写扩展到涉及整个软件生存周期。

- 软件生命周期：划分为计划、开发、运行与维护三个时期，而每个时期又划分为若干个阶段
  - 计划时期：主要任务是调查和分析。计划时期有问题定义和可行性研究两个阶段。
  - 开发时期：完成设计和实现两大任务。设计任务包括需求分析和软件设计两个阶段；实现任务包括编码和测试。
  - 运行时期：已交付的软件投入正式使用，便进入运行时期。这是软件生存期的最后一个时期，可能要持续若干年甚至几十年。在运行过程中，可能由于多方面的原因，需要对它进行修改。因此，软件人员在这一时期的主要工作，就是做好软件维护。

软件开发时期

- 需求分析：主要解决的问题是“目标系统必须做什么”，也就是要深入描述软件的功能和性能；确定软件设计的限制和软件与其他系统元素的接口；定义软件的其他有效性需求，并用“需求规格说明书”的形式准确地表达出来，提交管理机构评审。
- 软件设计：软件工程的技术核心，主要任务是把已确定了的各项需求转换成一个相应的体系结构，通常细分成总体设计和详细设计两个阶段。
- 编码：主要任务就是按照选定的语言把软件设计转换成计算机可以接受的程序代码， 即写成 “源程序清单”。
- 测试：保证软件质量的重要手段，其主要方式是在设计测试用例的基础上检验软件的各个组成部分

1.2 软件开发模型

- 软件生存周期模型
  - 从软件项目需求定义直至软件经使用后废弃为止，跨越整个生存周期的系统开发、运作和维护所实施的全部过程、活动和任务的结构框架。
  - 有多种软件生存期模型。例如：瀑布模型、演化模型、螺旋模型、智能模型等。它们各有特色，但一般都包含“定义（或计划）”、“开发”和“维护”3类活动。定义活动主要弄清软件“做什么”；开发活动集中解决让软件“怎么做”；维护活动则聚集于软件的“修改”，即“What-How-Change”。

瀑布模型

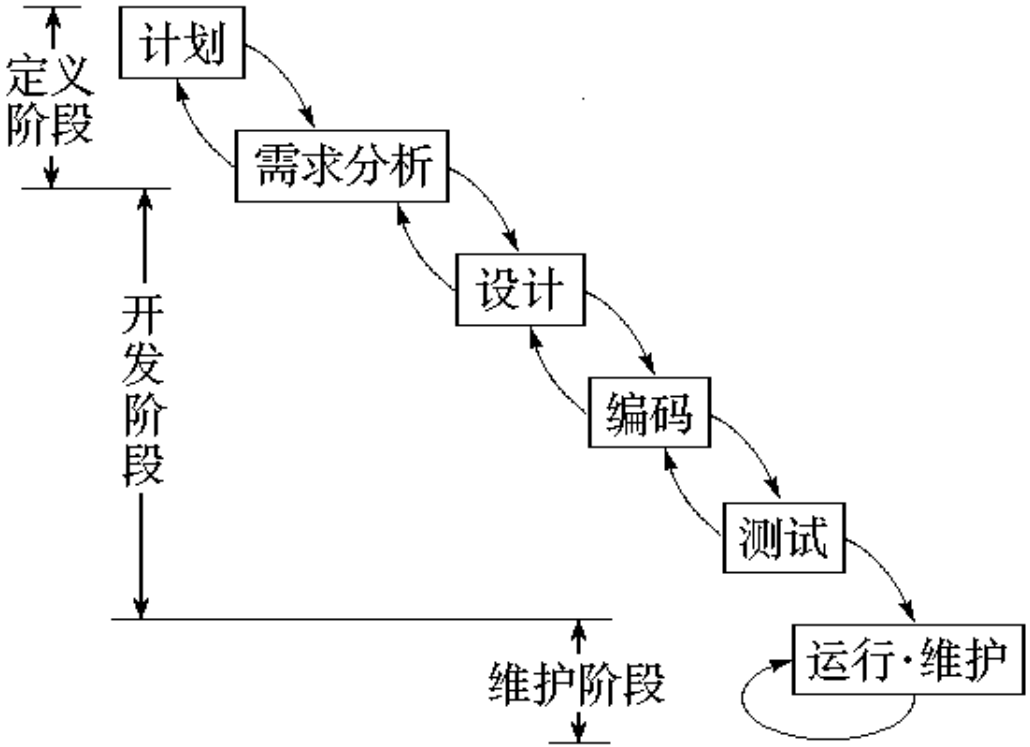
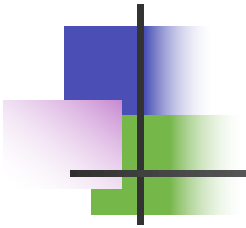


图1.3 软件生存周期的瀑布模型

瀑布模型的特点

- 阶段间的顺序性和依赖性 ：只有等前一阶段的工作完成以后，后一阶段的工作才能开始；前一阶段的输出文档，就是后一阶段的输入文档。只有前一阶段有正确的输出时，后一阶段才可能有正确的结果。
- 推迟实现：过早地考虑程序的实现，常常导致大量返工，有时甚至给开发人员带来灾难性的后果。瀑布模型在编码以前安排了分析阶段和设计阶段，并且明确宣布，这两个阶段都只考虑目标系统的逻辑模型，不涉及软件的物理实现。
- 质量保证：（1）每一阶段都要完成规定的文档。没有完成文档，就认为没有完成该阶段的任务。（2）每一阶段都要对完成的文档进行复审，以便尽早发现问题，消除隐患



- 演化模型（evolutional model）
  - 先做试验开发，其目标只是在于探索可行性，弄清软件需求；然后在此基础上获得较为满意的软件产品。通常把第一次得到的试验性产品称为“原型”。
  - 显然，演化模型在克服瀑布模型缺点、减少由于软件需求不明确而给开发工作带来风险方面，确有显著的效果。

螺旋模型

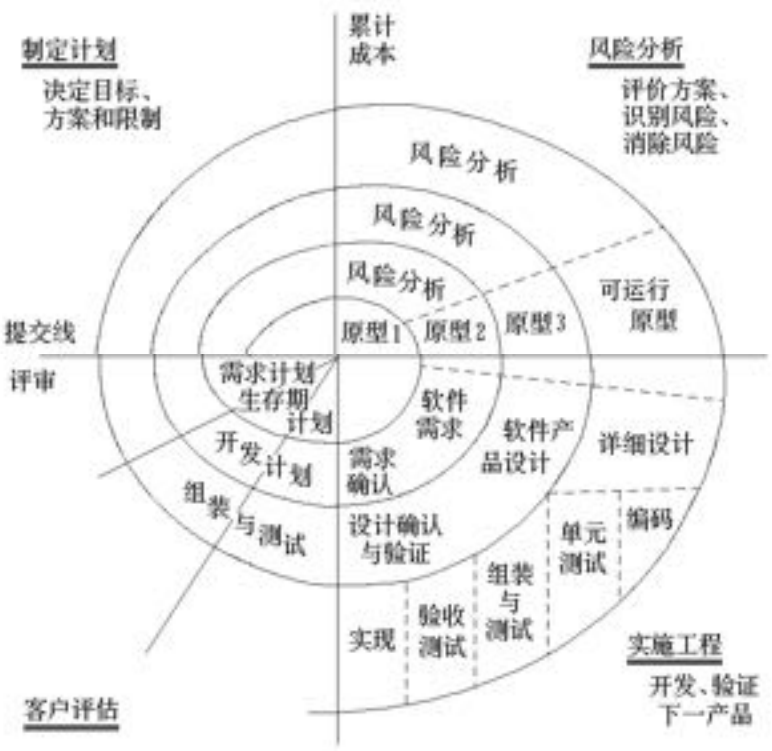
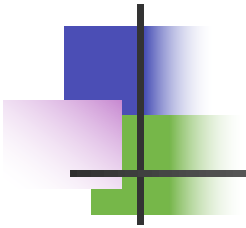


图1.4　螺旋模型



- 螺旋模型特点
  - 沿螺线自内向外每旋转一圈便开发出更为完善的一个新的软件版本。
  - 如果软件开发人员对项目的需求已有较好的理解，则无需开发原型，第一圈就可以直接采用瀑布模型，这在瀑布模型中认为是单圈螺线。
  - 若对项目的需求没有把握，就需要经过多圈螺线，并通过开发一个或多个原型来弄清软件的需求。
  - 对于高风险的大型软件，螺旋模型是一个理想的开发方法。风险分析是螺旋模型的一项重要活动

1.3 软件安全需求

- 安全需求商业上的体现
  - 安全的产品是高质量的产品
  - 媒体(以及竞争对手)将在安全问题上大做文章
    - 现代媒体具备将事实双向放大的作用
  - 用户将避免选择不安全的产品
  - 避免成为攻击者的目标
  - 修补安全漏洞的代价是高昂的

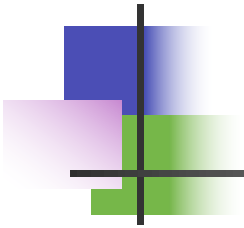
- 修补安全漏洞的代价高昂
  - 配合修补所需的开销，有些人不得列出一个攻击计划以便完整修复
  - 开发人员找出有漏洞的代码所需要的开销
  - 开发人员修补这些代码所需要的开销
  - 测试人员对本次修补进行测试所需的开销
  - 对本次补丁的安装进行测试所需的开销
  - 创建和测试国际化版本补丁所需的开销
  - 将这个补丁在你的网站上发布所需的开销
  - 编写响应的支持文档所需的开销
  - 改善公众关系所需的开销
  - 需要向ISP来付费让其代管补丁程序，还需要网络带宽和下载的费用
  - 浪费生产力所带来的开销
  - 用户应用这个补丁所需的开销
  - 如果客户决定推迟或取消对产品的使用…

1.4 软件安全生命周期

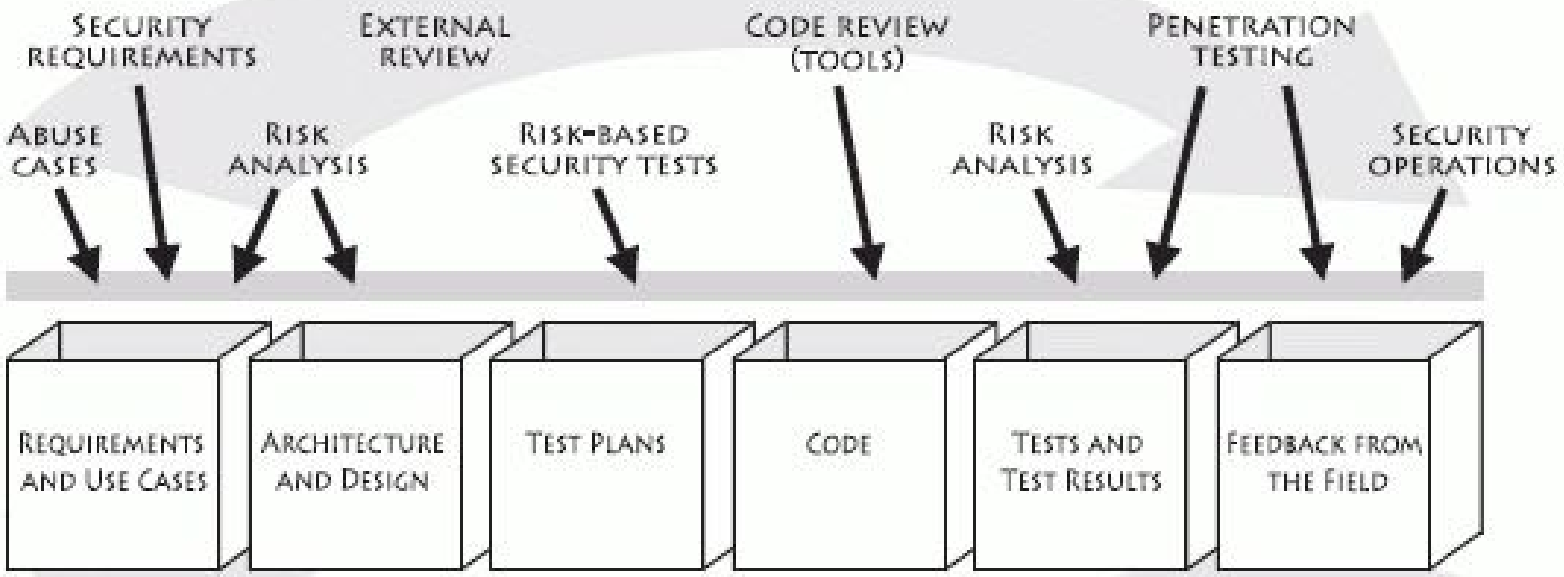
SDL(Security Development Lifecycle，安全开发周期)，微软提出的、从安全角度指导软件开发过程的管理模式

图 9.1 安全开发生命周期（©2010 年微软公司。权利所有。署名－非商业性使用－共同创作共享 3.0 许可）

- SDL适用于所有符合以下标准的软件
  - 广泛应用于企业、商业机构、政府部门或其他机构的软件；
  - 广泛处理个人或敏感信息的软件；
  - 广泛接入互联网的软件（不适用于连入微软互联网服务器进行在线升级代码、数据库的软件）



- Gary McGraw 软件安全过程
  - 代码审查
  - 架构风险分析
  - 渗透性测试
  - 基于风险的安全测试
  - 误用用例
  - 安全需求
  - 安全操作


SECURITY REQUIREMENTS
EXTERNAL REVIEW
CODE REVIEW (TOOLS)
PENETRATION TESTING
ABUSE CASES
RISK ANALYSIS
RISK-BASED SECURITY TESTS
RISK ANALYSIS
SECURITY OPERATIONS
REQUIREMENTS AND USE CASES
ARCHITECTURE AND DESIGN
TEST PLANS
CODE
TESTS AND TEST RESULTS
FEEDBACK FROM THE FIELD

本讲内容安排

1. 安全生命周期
2. 安全需求
3. 安全设计
4. 安全实现
5. 安全验证

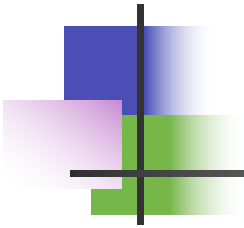
2.1 安全编码标准

- 安全编码
  - 具备良好记录的编码标准并强制执行这个标准。
  - 编码标准鼓励程序员遵循按照项目和组织的要求确定的一套统一规则和指引，而不是按程序员自己的习惯或偏好行事。

- 安全编码标准
  - 使用安全编码标准定义了一组可以对源代码的符合性进行评估的要求。安全编码标准对评估和对比软件的安全性、可靠性和相关属性提供了一种衡量标准。忠实地应用安全编码标准可以防止引人已知的源代码相关的漏洞。
  - CERT已经发布了C[Seacord 2008]和Java[Long 2012] 安全编码标准，并正在准备为C++ [SEI2012b]和Perl[SEI2012c]制定一个标准。
  - 思科和甲骨文等公司已采纳了CERT 的安全编码标准。

2.2 安全质量需求工程

- 早期的一项研究发现，在开发周期的早期引入安全性分析和安全的工程实践的情况下，投资的利润率范围为12% ~ 21%。美国国家标准和技术研究所(NIST) 报告称，安全性和可靠性方面有缺陷的软件每年要花费595亿美元在软件的崩溃和修补上。糟糕的安全性引起的代价表明，在这一领域，即使是很小的改进也会带来极高的回报。



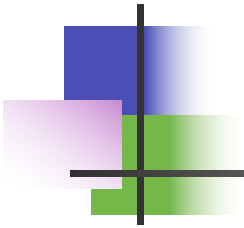
- 安全质量需求工程过程(Security Quality Requirements Engineering Process, SQUARE)用于引出和分析安全需求，它由SEI开发并已经应用于一系列客户案例研究。原始的SQUARE方法由9个步骤构成，但已被扩展，以解决隐私和探测问题。其中1 ~ 4 步是先决步骤。

- （1）统一定义：安全需求工程的先决条件。就一个给定的项目而言，团队成员都趋向于基于他们以前的经验在脑海中给它一个定义，但这些定义并不一定一致。我们没有必要去创造定义。 RFC 4949等资源、提供了一系列定义可供选择或加工。
- （2）识别安全目标：对于组织和即将开发的信息系统而言，必须识别需要保护的资产及与其相关的安全目标，并排定它们的优先次序。不同的用户有着不同的目标。例如，人力资源部门的用户可能会关心人事档案机密性的维护，而财务领域的用户则需确保财务数据不会在未经授权的情况下被访问或修改

- （3）开发文档：操作的概念、简洁陈述的项目目标、文档化的正常用法和威胁场景、误用案例，以及支持需求定义所需要的其他文档。如果缺乏，可能会导致混乱和误传达。
- （4）进行风险评估：有很多风险评估方法可基于组织需要进行选择。从第3步得到的文档为风险评估过程提供了输入。威胁建模也对风险评估提供了大量的支持。风险评估的结果可以帮助识别高优先级的安全问题。

- （5）选择引出技术：当存在若干种类型的用户时，挑选一种引出技术是很重要的。当存在具备不同文化背景的负责人时，比较正式的引出技术(如结构化会谈)可能很有效。在其他情况下，引出技术也许仅仅是与主要负责人坐下来，努力弄明白他的安全需求。在SQUARE的案例研究中，最成功的方法是需求加速法(Accelerated Requirements Method, ARM) 。

- （6）引出安全需求：这一步将用到之前选中的需求引出技术。大多数引出技术都提供了关于如何进行引出的详细指导。
- （7）对需求分类：分门别类使得需求工程师可以区分本质需求、目标(想得到的需求)，以及可能出现的架构约束。这有助于接下来的需求优先次序的排定。
- （8）排定需求的优先次序：需求优先次序的排定可以通过成本/收益分析得到，以决定哪些安全需求具有高性价比。层次分析法（Analytical Hierarchical Process，AHP）是一个排优先级的方法，它采用了对需求两两比较的办法来排优先级。



- （9）需求审查：需求审查可以在各种正式级别上进行，从Fagan审查法到同行评审(peerreview)。案例研究中，最有效的方法是Fagan审查法。一旦需求审查完成，组织就应该拥有一套按优先次序排定的初始安全需求。

2.3 用例与误用例

- 安全误用例(security misuse case)是用例(use case)的一个变种，用于从攻击者的角度描述一种场景。用例已经被证明在描述正常的使用场景方面很有效，同样的道理，误用例在描述入侵者使用场景以及最终识别安全需求方面是有效的
  - 表9.2 展示了安全用例和误用例之间的区别
  - 表9.3 展示了一个针对自动柜员机(ATM)的特定应用的误用例的示例

表 9.2　误用例和安全用例之间的区别

	误用例	安全用例
用法	分析并指定安全威胁	分析并指定安全需求
成功准则	攻击成功	应用程序成功
制造者	安全团队	安全团队
使用者	安全团队	需求团队
外部参与者	攻击者、用户	用户
驱动者	信息资产漏洞分析、威胁分析	误用例

表 9.3　特定应用的误用例

误用例名称：ATM 上的欺骗客户	
概要：误用者成功地使 ATM 相信他是一名合法用户，从而获得对 ATM 客户服务的授权访问	
先决条件：	
1) 误用者拥有一个合法用户的有效身份和认证	
2) 误用者拥有无效的身份和认证，但与有效的身份和认证如此相似，以至于 ATM 无法辨别	
3) ATM 系统发生了故障，接受了正常情况下理应拒绝的身份和认证	
误用者交互	系统交互
请求访问	
	请求身份和认证
误识别和误认证信息	
	授权访问
后验条件：	
1) 误用者可以使用被假冒的合法用户可获得的所有客户服务	
2) 在系统日志中（如果有），只会出现 ATM 被合法用户访问的记录	

本讲内容安排

1. 安全生命周期
2. 安全需求
3. 安全设计
4. 安全实现
5. 安全验证

3.1 安全的软件开发原则

- 机制经济性原则
  - 机制经济性是一个众所周知的原则，适用于系统和软件设计的各个方面，它尤其与安全密切相关。特别是，安全机制应该相对精练、简单，以便于实现和校验(例如安全内核)。
  - 复杂的设计增加了在它们的实现、配置和使用中造成错误的可能性。另外，当达到一定水平的安全保证时，安全机制也会变得更为复杂，所需付出的努力将显著增加。因而，在系统设计中通过付出更多的努力以获得一个问题的简单的解决方案通常更为划算。

- 失败-保险默认原则
  - 访问判定建立在显式授权而不是隐式授权的基础上，这意味着在默认情况下，访问是被拒绝的，保护方案会识别出在什么条件下访问是经过许可的。
  - 如果机制不能授权访问，那么这种情形很容易被侦测到并纠正。然而，如果机制阻止访问失败，则在正常使用时这种失败情况很可能被忽视。

- 完全仲裁原则
  - 图9.6 描述了完全仲裁问题。一个保护系统的主要基础是要求检查对每一个对象的访问是否得到授权。这要求每一个访问请求的来源都通过明确的鉴别，并被授权访问某个资源。

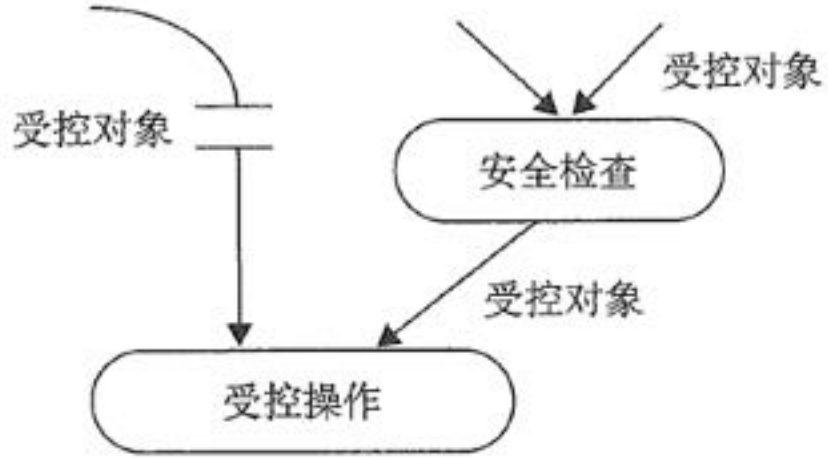


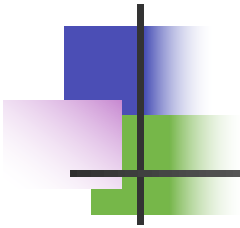
图 9.6 完全仲裁问题

开放式设计原则

- 安全的设计不应该依赖于对潜在攻击者或模糊代码的忽视。比方说，加密系统和访问控制机制应该能够置于开放的检查环境中，且仍保持其安全性。
- 典型的实现方式是降低保护机制与保护钥匙或密码之间的耦合度。这种方式还有一个额外的优点，即能够在不考虑审查者有可能会对保护措施造成危害的前提下，对保护机制进行彻底地检查。
- 之所以有必要采取开放式设计，是因为对于一个通过使用反编译或检查二进制代码技术的攻击者而言，所有代码都是开放的，是可以被窥视的。因此，任何基于模糊代码的保护方案最终都将被揭开真面目。实现一个开放式的设计也允许用户核实，保护方案是否足以应付其特殊的应用需求。

特权分离原则

- 特权分离通过要求一个以上的条件来授予权限，从而消除了单点失败。双因素认证方案就是特权分离原则的应用示例：你拥有的东西和你知道的东西。
- 比方说，安全令牌和基于密码的访问方案具备下列属性(假定能够正确实现)：
  - 用户可能拥有一个脆弱的密码，甚至可能泄漏它，但是如果没有令牌，那么访问方案就不会失败。
  - 用户可能遗失令牌，或者令牌被攻击者窃取，但是如果没有密码，那么访问方案就不会失败。
  - 只有当攻击者同时拥有了令牌和密码的时候，该机制才会失败
- 人们经常把特权分离与由子系统构成的程序的设计混淆起来，后者的子系统基于其所需要的特权。这种方式允许设计者应用一种较细粒度的**最小特权原则**。



最小特权原则

- 当一个有漏洞的程序被攻击者利用的时候，利用代码会以程序当时所拥有的特权运行。在正常的操作过程中大多数系统都需要允许用户或程序执行"需要高级特权才能执行的操作或命令"的一个有限子集。
- 一个常见的最小特权例子是密码修改程序。用户必须能够修改自己的密码，但是却不能准许他们自由地读取或修改包含所有用户密码的数据库。因此，密码修改程序必须正确地接受来自用户的输入，并基于附加的授权检查，以确保用户只能修改自己的密码。如果程序员没有对程序的安全敏感区加以关注，那么诸如这样的程序就可能会引人漏洞

- 最少公共机制原则
  - 在某些方面，最少公共机制原则与分布式计算的总体趋势是冲突的。最小公共机制原则规定应该把由一个以上用户所共用的机制的数量减到最小，因为这些机制意味着潜在的安全风险。如果攻击者设法破坏其中某一个共享机制的安全，就有可能通过向依赖于资源的进程中引人恶意代码从而访问或修改其他用户的数据。这个原则看上去与另一种趋势相矛盾一一使用分布式对象为公共数据元素提供共享仓库。
  - 视考虑问题的侧重点不同，为此问题提供的解决方案也许有所不同。然而，如果你正在设计这样一个应用程序，它的每个实例都拥有自己的数据存储，并且在应用程序的多个实例之间或者在分布式对象系统中的多个客户端或多个对象之间，数据是非共享的，那么就要考虑你的系统设计以使最少公共机制执行于程序的进程空间内，且不与其他应用程序共享。

心理可接受性原则

- 和“可用性” 大体一致，它常常需要与安全性进行权衡。
- 可用性也是安全性的一种形式，因为用户错误常常可能导致违反安全问题(例如，当设置或改变访问控制的时候)。US-CERT漏洞数据库中的许多漏洞都可以归结为可用性问题。在这个数据库中，排在缓冲区溢出漏洞之后、第二常见的漏洞就是"安装之后的默认配置是不安全的"。

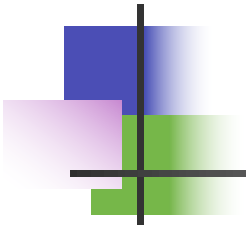
3.2 威胁建模

- 威胁模型
  - 由应用程序的架构定义和针对应用场景的威胁列表构成。
  - 威胁建模包括识别关键资源、应用程序的分解、识别并分类针对每一个信息资产或组件的威胁、基于风险等级评估威胁，然后开发威胁缓解措施，并在设计、编码和测试用例中实现这些缓解措施。这些威胁模型应该作为软件演化中的需求和设计环节进行评审口不够精确的模型可能会导致为确保开发中系统的安全性而付出的努力不足(或者过分)。

- 微软结构化建模过程
  - 1. 识别资产。识别你的系统必须保护的资产。
  - 2. 创建蓝图。规划应用程序的架构，包括子系统、信任边界和数据流。
  - 3. 分解应用程序。分解应用程序的架构，包括底层网络和主机基础设施的设计，从而为应用程序创建一个安全规范。目标是揭露应用程序的设计、实现或部署配置中存在的漏洞。
  - 4. 识别威胁。从攻击者的目标和应用程序的架构及可能存在的漏洞出发，识别出可能影响到应用程序的威胁。
  - 5. 记录威胁。使用为每一种威胁定义的一组捕获那些威胁的模板的常用属性，来记录每一个威胁。
  - 6. 评估威胁。基于攻击的可能性和危害的结果，按优先次序排列呈现最大风险的威胁。比较"威胁带来的风险"和"缓解该威胁的代价" 在此基础上判断是否应该对威胁采取行动。

3.3 分析攻击面

- 系统的攻击面
  - 是对手可以进入系统，并可能导致损害的方法的集合。重点是可能提供进攻机会的系统资源
  - 提高系统安全性的一种方法是减少其攻击面。
    - 这种减少需要分析目标(对手旨在控制或指派以开展攻击的进程或数据资源)、通道(通信信息的手段和规则)和访问权限(与每个资源相关的特权)。
    - 例如，可以通过限制对资源的访问权限的集合来减少攻击面。这是最小权限原则的另一种应用，也就是说，给予特定用户对所需资源最低的访问权限。同样，一旦套接字不再需要就关闭它们，从而减少通信信道，会减少攻击面。
  - 也可以通过加强相对于一个进程的先决条件和后决条件罗以便只允许预期的影响来实现。



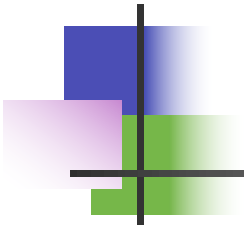
- 测量攻击面
  - 在对相似的系统(如不同的发行版本)作比较时是特别重要的。如果向系统中添加新的特性增加了攻击面，则应该主动地测量攻击面，而不是随着产品演变和不断变化的需求而进行测量

3.4 输入验证

造成漏洞的一个常见原因是没有经过适当验证的用户输入。输入验证需要以下几个步骤。

- 1.必须识别所有的输入源。输入源包括命令行实参、网络接口、环境变量以及用户控制文件等。
- 2. 指定规范并验证数据。所有来自非受信来源的数据必须是完全指定的，而且必须根据指定的规范进行验证。必须将系统设计为能够处理任何范围或组合的有效数据。在此场景中，有效数据是指系统的设计和实现可以预料到的数据，从而保证它们不会导致系统进人不确定的状态。例如，如果系统接受两个整数作为输入，并以两个值做乘法运算，则系统必须要么(a) 验证输入，以确保操作的结果不会导致溢出或发生其他异常状况，要么(b)准备好处理运算的结果。

- 3. 制定的规范必须处理各种限制、最小值和最大值、最小长度和最大长度、有效内容、初始化和重新初始化需求，以及存储和传输的加密需求。
- 4. 确保所有输入都符合规范。使用数据封装(如类)来定义并封装输入。例如，我们不再检查用户名输入中的每个字符，以确定它是有效的字符，而是定义一个类来封装所有与该类型的输入有关的操作。输入必须尽快得到验证。不正确的输入并非总是恶意的——常常只是个意外。尽早报告错误通常有助于纠正问题。当在代码深处发生异常的时候，有时很难发现起因是越界的输入还是无效输入



- 可以将数据字典或类似的机制用作所有程序输入的规范。
- 推荐使用用于处理标准化的并被广泛使用的数据格式，如XML、HTML、JPEG和MPEG类型数据的可靠、高效和方便的工具，都是现成的。

本讲内容安排

1. 安全生命周期
2. 安全需求
3. 安全设计
4. 安全实现
5. 安全验证

4.1 编译器检查

- C和C++编译器对于类型检查通常都不严格，但是你一般可以提升检查级别，从而能够自动侦测到某些错误。开启尽可能多的编译器警告，并修改代码，以便不带警告地通过编译。严格地使用不同头(.h)文件中的ANSI原型，以确保所有函数调用都使用正确的类型。

- 当使用GCC编译C或C++程序的时候，至少使用下面的编译标记(会在主机上发出警告消息)并尝试消除所有警告：
  - gcc -Wall -Wpointer-arith -Wstrict-prototypes -O2
  - 之所以使用-O2标记，是因为某些警告只能被执行于更高优化级别上的数据流分析侦测到。使用-W-pedantic或更专门的标记(如-Wstrict-verflow=3)可诊断可能导致边界检查错误的代数简化

对于使用Visual C++的开发人员来说，应该使用/GS选项来启用测试仪(canary)，并执行某种栈重组以防止常见的利用。这种栈重组随着时间的推移不断进化。图9.9展示了Visual C++中的/GS选项的进化过程。

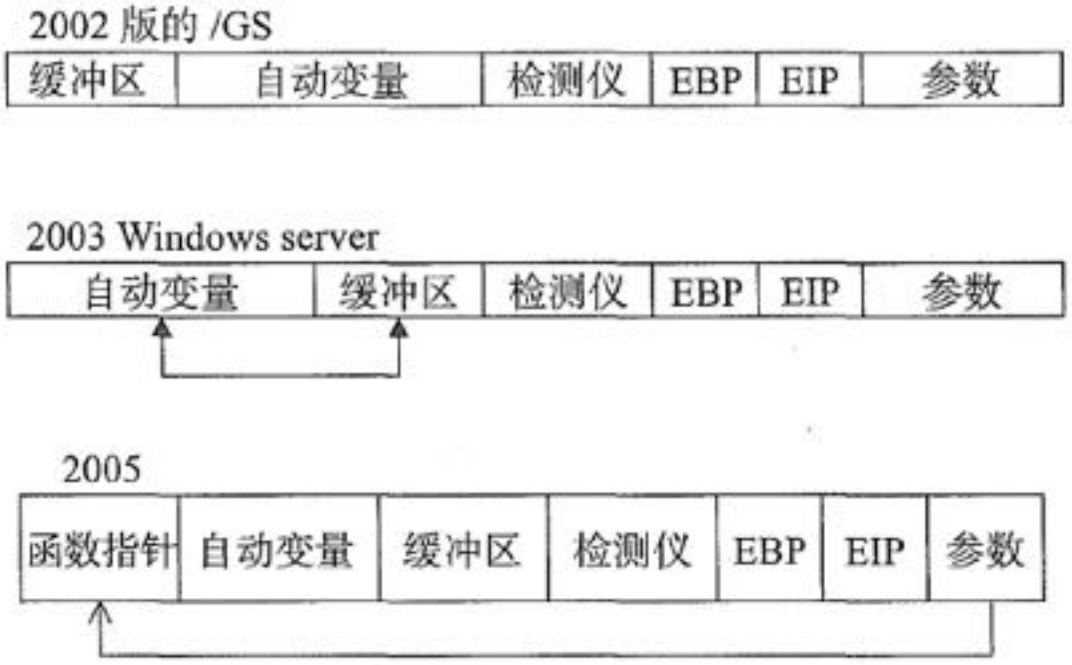


图 9.9 Visual C++ 中的 /GS 标志

4.2 有安全保证的C和C++

CERT正在扩展一个开放源码的编译器来执行有安全保证的C/C++的分析方法，如图9.10所示。

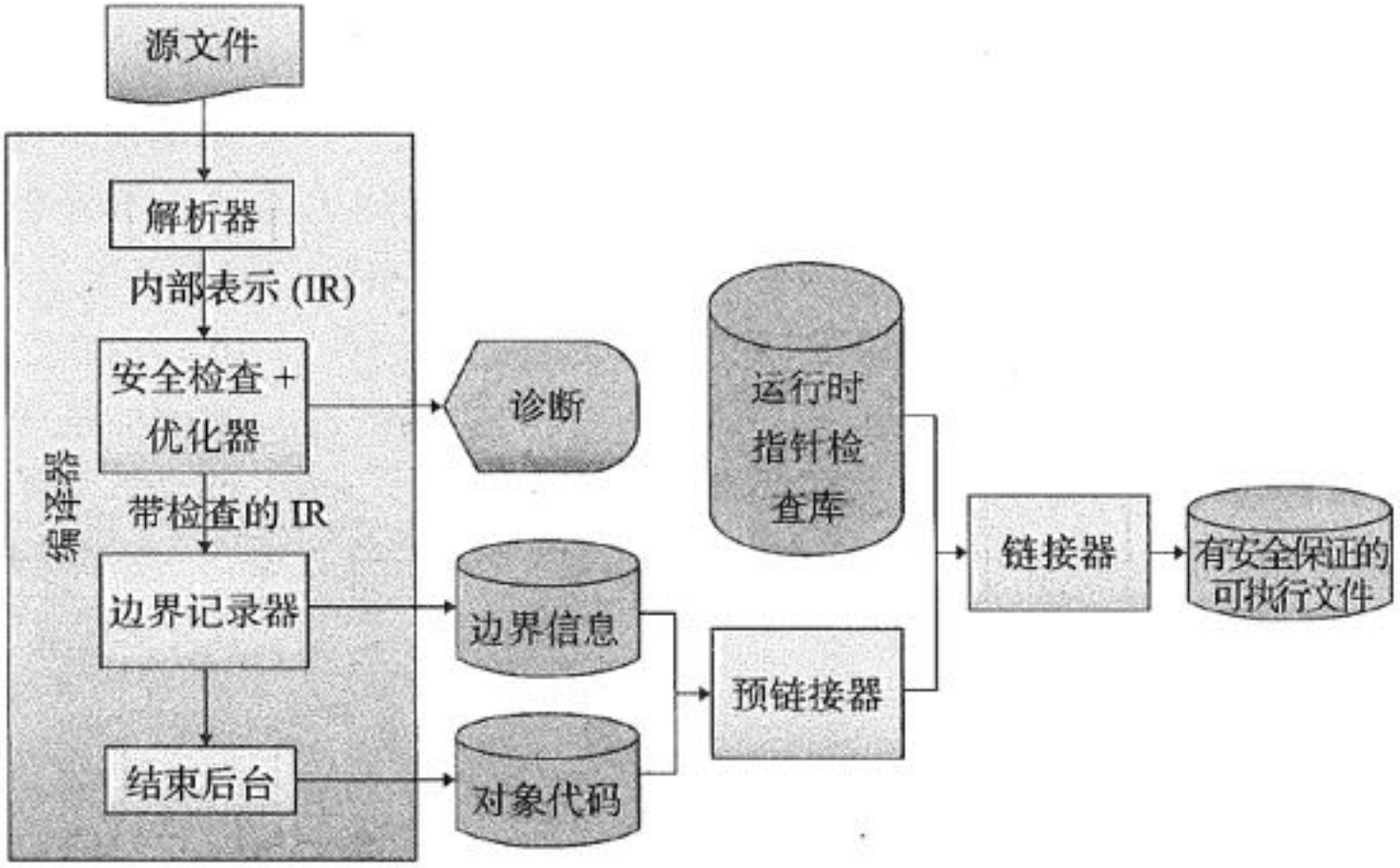


图 9.10 有安全保证的 C/C++ 的分析方法

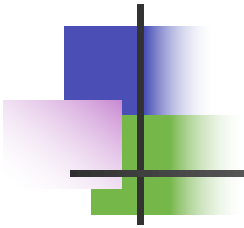
4.3 静态分析

静态分析

- 静态分析器在源代码上操作，生产潜在的错误或意外的运行时行为的诊断警告。
- 静态分析技术虽然有效，但容易出现误报和漏报。为了在最大程度上可行，分析器强制执行的规则都应该是完备和可靠的。如果对于一个特定的规则，分析器不给出漏报的结果，则它被认为是可靠的这意味着它能够找到整个程序中所有违反规则的情况。如果分析器不给出误报的结果，或假警报，那么它被认为是完备的。表9.4中列出了一个给定规则的可能性。

表 9.4　完备性和可靠性

	误报	
漏报	是	否
	否 可靠但有误报	完备且可靠
	是 不可靠还有误报	不可靠



- 静态分析局限性和使用方法
  - 在最大限度地减少误报和漏报上有很多取舍。同时使两者都尽量少显然是更好的。自动化工具，在需要最少的人工输入且大规模的代码库的有效前沿上，在误报和漏报之间往往存在紧张的关系。
  - 可采用各种不同的方式来使用静态分析工具。一个常见的模式是把工具整合到持续构建/集成的进程中。另一个用法是一致性测试。

- 不同静态分析工具对安全问题覆盖的不均衡性
  - 例如，最近的一项研究[Landwehr 2008]发现，所有五个研究的C 和C++ 源代码分析工具都未能诊断出210个测试用例中超过40%的用例，而只有7.2%的测试用例被所有五个工具成功地诊断出，如图9.11 所示。
  - NIST 静态分析工具大全(NIST Static Anaiysis Tool Exposition, SATE)也表明，开发全面的静态分析工具分析标准是有问题的，因为构成正确报告或误报的事物，有很多不同的角度[Okun 2009] 。
  - 为了解决这些问题， WG14 C 标准委员会正在研制ISO/IEC TS 17961 , C安全编码规则[Seacord 2012a]

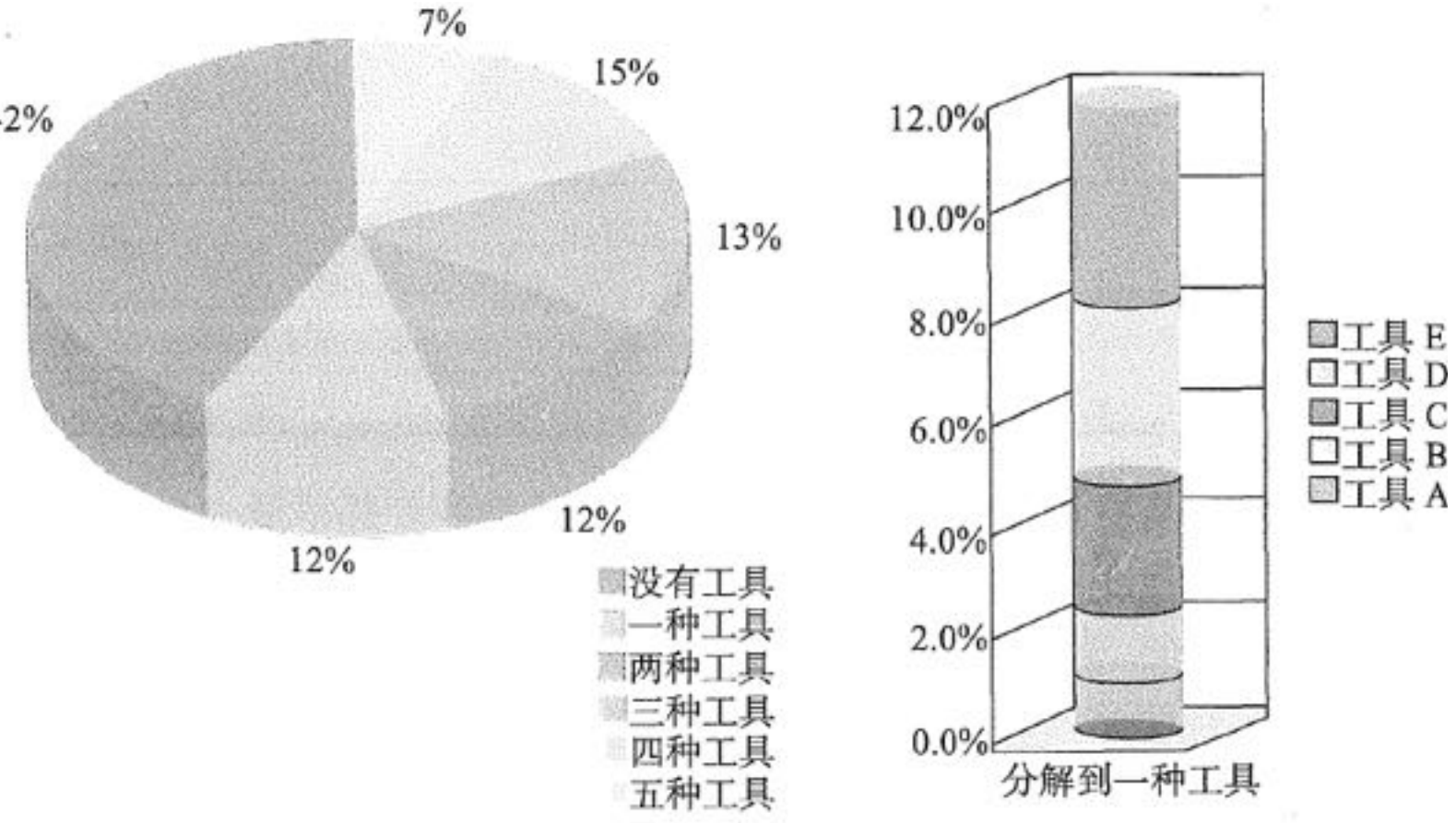


图 9.11　C 和 C++ 的“广度”用例覆盖率（资料来源：[Landwehr 2008]）

4.4 深层防御

- 深层防御背后的思想是使用复合防御策略来管理风险，这样的话当某一层防御失效的时候，另一层防御能够防止安全缺陷演变为可利用的漏洞，并且可以限制得逞的利用所导致的后果。例如，结合使用安全编程技术和安全的运行时环境，可以降低在部署时遗留在代码中的漏洞在操作环境中被利用的可能性。

- 深层防御的替代做法是依赖于单一的策略。例如，有了完备的输入验证，理论上就不再需要其他的防御了。例如，如果所有输入字符串都能经过校验且具备有效的长度，那么就不必使用有界字符串复制操作了，而且可以毫无顾忌地使用strcpy() 、memcpy()及类似的操作。同样也没有必要提供任何类型的运行时保护来防止基于校或堆的溢出，因为根本不存在溢出的机会。

- 尽管理论上可以开发出一个小程序并使用输入验证确保其安全，但对于实际的程序来说，这是办不到的。大型的实际程序由程序员团队开发，模块一旦完成，很少能保持一成不变。甚至在初次交给用户或最初的部署之前，也可能进行维护。随着时间的流逝，出于各种各样的原因，这些程序可能会被许多程序员修改。鉴于所有这些改变和复杂性，很难保证单凭输入验证就可以提供完备的安全性。
- 多层防御可以用于防止运行时漏洞利用，除此之外，对于识别开发和维护期间可能发生的变化也有帮助。

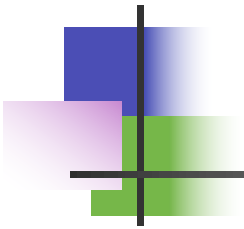
本讲内容安排

1. 安全生命周期
2. 安全需求
3. 安全设计
4. 安全实现
5. 安全验证

5.1 渗透攻击

渗透测试(penetration testing)通常意味着从攻击者的角度探测应用程序、系统或网络，从而搜寻其潜在的漏洞。

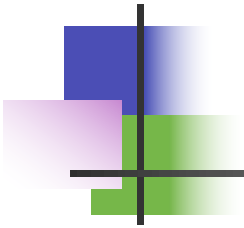
- 渗透测试很有用，尤其是当使用架构风险分析来驱动测试的时候。渗透测试的优点在于能够获得对真实环境中安装的软件的良好理解。
- 然而，任何忽略软件架构的黑盒渗透测试都不能发现软件深层次的风险。
- 未能通过市面上那些过分简单的应用程序安全测试工具进行的笼统的黑盒测试的软件是必然会失败的。这也意味着粗略的渗透测试难以揭示系统的真实安全情况。不过，倘若连一个笼统的渗透测试者都通过不了，那问题可就大了。



- 测试软件以验证软件是否满足安全需求，这是不可或缺的。除了扫描常见的漏洞之外，这种测试还包括严肃地尝试攻击软件并破坏其安全性。正如前面所讨论的，测试用例可以取自威胁模型、攻击模式、滥用例以及规范和设计。白盒测试和黑盒测试都是适用的，因为渗透测试既能测试功能性需求，又能测试非功能性需求。

5.2 模糊测试

- 模糊测试或模糊化是一种查找软件中可靠性问题的方法其中有些问题也许就是安全漏洞，这种方法是通过故意向程序接口输入元效数据或格式有误的数据而实现的。模糊测试通常是暴力破解的方法，并且需要执行大量的测试(包括使用多种变体以及测试多次)，因此模糊测试一般需要自动地进行。
- 模糊测试是通过攻击接口来发现软件缺陷的若干种方式之一。任何应用程序接口(例如网络、文件输入、命令行、Web表单等)都可以进行模糊测试。其他攻击接口的方法包括探查、嗅探和回放、伪造(有效的消息)、淹没(有效或无效的消息)、劫持/中间人、恶意消息和失序消息。



- 模糊测试的目标随着被测试的接口类型而稍有变化。比方说，当测试一个应用程序能否正确地处理某个特定的协议时，目标包括发现导致截断消息的误处理、不正确的长度值，以及可能导致″不稳定的操作″协议实现的非法类型代码。

- 动态随机化输入功能测试，也称作黑盒模糊化(black-box fuzzing)，
  - 20世纪90年代早期以来已被广泛用于找出软件应用程序中的安全漏洞。
  - 例如，Justin Forrester和Barton Miller以模糊测试的方式在Windows NT上测试了超过30个基于GUI的应用程序，具体做法是使这些应用程序接受有效的键盘和鼠标事件流以及随机的Win32 消息流[Forrester 2000]。当应用程序接受了可以通过使用鼠标和键盘产生的随机的有效输入时，有21% 的应用程序在测试中崩溃，一另外有249毛的程序挂起。当应用程序接受了原始的随机Win32 消息时，所有的程序都会崩溃或挂起。从那时起，模糊测试逐渐发展为涵盖范围广泛的软件接口和各种测试方法的技术。

5.3 开发人员准则与检查清单

- 可以通过执行基于检查清单的设计和代码检验来确保软件的设计和实现远离那些已知的问题。
- 尽管检查清单是一个有用的工具，但它们也可能被误用，最常见的情况是在没有完全理解清单中条款本质的情况下就拿来使用。这可能会导致对问题的误解，或者对设计或实现进行不必要的或无正当理由的改动。

- 检查清单有三个目的。
  - 第一作为一种对已知问题的提醒而存在，这样我们就能记得查找它们。
  - 第二，用于记录设计和代码中检查出的问题以及这些检查发生的时间。
  - 第三(或许是最具价值同时也是最容易忽略的目的)，作为开发人员之间交流常见问题的一种手段。
- 检查清单处于不断的发展中。新的问题需要加人，而那些不会再发生的旧问题(可能由于它们的解决方案已经制度化，或者它们已经赶不上技术进化的步伐)应该从检查清单中移除，以免继续不必要地耗费精力。要想决定哪些条款应该保留或者从检查清单中移除，应当取决于欲检测那些条款所需付出的努力，以及已发现的缺陷的实际数量和严重程度。

5.4 独立安全审查

- 独立安全审查(independent security review) 可能根据审查的性质和范围而发生重大变化。安全审查可以放眼整体，也可以只着眼于某一局部。独立审查常常是在开发团队之外发起的。如果干得漂亮，可以(为安全性)作出颇具价值的贡献;如果做得不好，则可能对开发团队造成干扰，使整个团队偏离最住的方向。

- 独立安全审查可以帮助我们实现更安全的系统。外部审查者会带来独立的观点，例如识别和纠正无效的假定。开发大型复杂系统的程序员往往会由于太贴近系统而失去大局观。比方说，安全性要求苛刻的应用程序的开发人员也许会为特定方面的安全问题殉精竭虑，却完全忽略了某些其他有漏洞的部分。经验丰富的审查人员熟悉常见的错误和最佳的解决方案，并能够提供概括性的观点，包括识别过程中的缺陷、架构上的弱点，以及设计和实现中需要额外或特别注意的地方。

- 独立安全审查还可以作为一个有用的管理工具。委托一个独立审查而且按审查结论的做法有助于管理，因为这样可以证明它们已经满足了预期的需求。此外组织与管理机构之间的关系通常随着独立审查的额外保证而得到改善口组织委托独立安全审查的目的常常是将审查结果公之于众，特别是当评测具有权威性且得出正面结果的时候。

谢谢大家！