第 32 卷 第 20 期 计 算 机 工 程 2006 年 10 月
Vol.32 № 20 Computer Engineering October 2006
·安全技术· 文章编号：1000—3428(2006)20—0163—03 文献标识码：A 中图分类号：TP309

软件漏洞起因的分类研究

李 淼[1,2,3]，吴世忠[3]
(1. 长春光学精密与物理研究所，长春 130033；2. 中国科学院研究生院，北京 100089；3. 中国信息安全产品测评认证中心，北京 100089)

摘 要：软件漏洞是发生安全事件的根源，当软件漏洞被利用时，会造成严重的后果。对CERT或者SANS公布的漏洞进行的分析表明：许多漏洞都归因于软件开发人员反复地犯同一类错误。该文对软件漏洞的起因进行分类，应用领域包括：开发人员的使用指南（以避免犯共性的错误），软件工程专业学生的教学素材，以及软件测试人员或审计人员的“项目核查清单”。
关键词：软件漏洞；分类法；完全介入；特征冲突

Research on Taxonomy of the Software Vulnerabilities Origins

LI Miao[1,2,3], WU Shizhong[3]
(1. Changchun Institute of Optics Fine Mechanics and Physics, Changchun 130033; 2. Graduate School of Chinese Academy of Sciences, Beijing 100089; 3. China Information Technology Security Certification Center, Beijing 100089)

【Abstract】 The vulnerabilities of software are the root of most security incident. When these vulnerabilities can be exploited, this has a serious impact. Analysis of vulnerability alerts as distributed by organizations like CERT or SANS shows that many vulnerabilities can be attributed to the same mistakes made by developers. This paper proposes a structured taxonomy of the origins of software vulnerabilities. Such a taxonomy can be used as an aid for developers to avoid common pitfall, as didactical material for students in software engineering or as a “checklist” for software testers or auditors.

【Key words】 Software vulnerability; Taxonomy; Complete mediation; Feature interaction

众所周知，软件漏洞是发生安全事件的重要原因。在计算机安全学中，漏洞指的是存在于一个系统内的弱点或缺陷，系统对一个特定的威胁攻击或危险事件的敏感性，或进行攻击的威胁作用的可能性。漏洞通常是由软件错误（例如未经检查的缓冲区或者竞争条件）引起的。过去几十年里，研究人员已经从不同的角度研究了这个问题。许多研究计划旨在开发检测漏洞或者利用漏洞的工具（扫描器和入侵检测系统）。此外，研究人员已经提出了各种各样的漏洞分类方法。在文献[1]中，Bishop 对已有的漏洞分类方法进行了深入的分析，并且得出的一个重要结论：不存在最好的漏洞分类方法，不同的分类目的就会有不同的最优分类方法。

本文对软件漏洞的起因进行了分类，其主要目的是正确地认识软件工程师犯了哪些错误，从而把漏洞引入到软件中。对软件漏洞起因的分类不可避免地会带有一些武断性。本文提出的分类方法主要从软件开发生命周期（分析、设计、实现、配置或者维护阶段）的角度讨论产生漏洞的起因。

1 相关工作

本文提出的分类方法并不是第一个漏洞分类方法。早在 20 世纪 70 年代，许多研究就对操作系统安全漏洞进行了分类。20 世纪 90 年代，Landwehr等人[2]公布了包含 50 个漏洞的目录，并对它们进行了分类。本文改进了文献[2]的工作，其主要目的是证实漏洞的起因。

为了支持管理员完成管理系统和网络的任务，CVE提供了一个标准的漏洞列表。另外，使用这个列表，用户就可以对不同安全产品的漏洞数据库做出比较。Krsul[3]提出了一个便于在漏洞数据库中组织漏洞的分类方法。

Bishop[1]对现存的漏洞分类方法进行了深入的分析，并且发现它们对于开发漏洞检测工具是不适合的。他提出了一个基于漏洞特征的分类方法。Bishop也认为：不存在最好的漏洞分类方法，不同的分类目的就会有不同的最优分类方法。

其它分类方法关注安全事件的分类。CC（Common Criteria）为软件开发过程的不同阶段提供了指南，因此可以避免许多漏洞的产生。

2 分类方法

本文提出的分类方法有两个层次结构，如表 1 所示。

表 1 分类方法

分析阶段	缺乏风险分析
	偏颇的风险分析
	未预见的风险
设计阶段	依赖不安全的抽象层
	安全性与易用性或功能的折中
	缺少日志记录
	残留风险
实现阶段	没有检查输入参数
	非原子检查
	访问控制验证
	不安全的异常处理
配置阶段	软件的重用
	复杂的或者不必要的配置
	默认配置的安全性
维护阶段	特征冲突
	向后兼容性

对于每一类，本文给出简要的描述，必要时会举例说明。在提出这个分类方法的过程中，我们参考了一些现有的分类

基金项目：国家部委“929”专项工程基金资助项目
作者简介：李 淼(1978－)，男，博士生，主研方向：信息安全；吴世忠，研究员、博导
收稿日期：2006-03-21 **E-mail：**david_li2001@hotmail.com

方法[1~4]、描述漏洞的书籍[5~8]和CVE的漏洞列表。

2.1 分析阶段

在分析阶段，开发人员应该分析风险，并从较高的抽象层次上提出适当的安全需求。这个阶段产生的缺陷既具有共性又各不相同。这些缺陷分为 3 类：

(1)缺乏风险分析

在开发软件的时候没有考虑到安全问题是一个非常普遍的现象。如果软件工程师在分析阶段没有考虑潜在的风险和恰当的软件安全策略，那么最终的软件产品含有漏洞也就不足为奇了。

(2)偏颇的风险分析

通常情况下，仅有一方对给定的信息系统进行风险分析。因此，提出的安全需求只反映了一方的观点，而没有考虑到另外一方或几方的安全要求。因此在风险分析阶段应该考虑所有的当事人。

基于智能卡的数字现金系统是一个很好的例子。这些系统较好地降低了金融机构的风险，但却没有考虑到智能卡用户的相关要求。

(3)未预见的风险

该类涉及的范围比较广泛。所有开发人员没有意识到的风险都属于这一类。

这类的例子是移动代码（mobile code）带来的风险。操作系统的设计者没有预料到这个风险。如果预料到这个风险，操作系统从一开始就会采用类似沙箱的机制。

2.2 设计阶段

在设计阶段，开发人员根据制定的安全策略选择相应的安全技术。与分类分析阶段所引入的漏洞的起因相比，分类设计阶段所引入的漏洞的起因相对容易。

(1)依赖不安全的抽象层

虽然程序语言或者操作系统提供的许多接口隐藏了程序设计的复杂性，但是它们是不安全的。如果在程序设计时使用了这些抽象层，那么可能导致安全事件的发生。

(2)安全性与易用性或功能的折中

众所周知，设计软件时要在安全和功能之间做出折中。大多数安全措施往往会给用户带来不便，而功能强大的特性通常很容易被滥用。软件开发人员通常强调功能而不是安全性。这个问题经常是一个态度问题：软件开发人员往往花费大量的时间考虑怎样使事情的发生成为可能。从安全性的角度看，考虑怎样使事情不可能发生也是重要的。

这类的典型例子是允许附件中带有可执行文件的电子邮件客户端。用户只要打开电子邮件，附件中的可执行代码就可以自动运行，这给用户带来了方便。可是，它也带来了许多严重的风险（例如，前几年接连发生的电子邮件病毒）。

(3)缺少日志记录

可能会出现这种情况：软件设计人员花费了一些时间来考虑安全预防措施，但是没有采取检测违反安全行为的措施。一个好的日志机制对于跟踪安全事故是非常重要的。在设计阶段，要维持预防和检测之间的平衡。

Java 安全体系结构的设计就属于这类的例子。它为访问控制提出了一种很好的预防方法，但是对日志机制的支持却很不够。

(4)残留风险

不符合以上 3 类的情形都属于这个类别。残余风险指的是在分析阶段证实的，但是在设计阶段没有解决的风险。

2.3 实现阶段

众所周知，检测和修改大规模软件系统中的错误是非常困难的。而检测与安全性相关的错误则更加困难，这是因为安全性测试是一种创造性测试：测试人员必须考虑到攻击者试图入侵系统所采取的各种方法。此外，攻击者经常研究源代码以查找软件系统的漏洞，然而自动测试系统考虑的多是软件系统的常规使用。

几类在实现阶段产生漏洞的起因如下：

(1)没有检查输入参数

从广义上理解，一个程序的输入可以包括通过文件、网络连接、环境变量、与用户的交互的输入。开发人员通常假设程序的输入参数是正确的。攻击者会利用这个假设进行攻击。典型的例子是缓冲区溢出。如果输入参数有问题，那么无论其内容如何都会“溢出”到 CPU 执行栈的另一位置。如果攻击者选择这个输入，它就有可能用来启动攻击者选定的任意代码。

(2)非原子检查

程序实现过程中经常出现这样的情况：检查变量是否满足条件，如果满足，执行一些动作。通常可以使检查和动作之间的条件无效来进行攻击。

这类的典型例子是所谓的“竞争条件”(race condition)。当由于事件次序异常而造成对同一资源的竞争，从而导致程序无法正常运行时，就会出现竞争条件。注意，竞争条件无须介入同一程序的两个部分之间的竞争；如果一个外部的攻击者可以通过意想不到的方式干扰程序，那么就会出现很多安全问题。例如，如果 Tripwire 2.3.0 确定某个文件不存在，它就会尝试着创建该文件，而不去考虑在进行这两个步骤期间，该文件是否已经被攻击者创建。

(3)访问控制验证

安全系统设计有一个重要的原则——完全介入：访问控制机制应该检查对每个对象的每次访问。可是，访问控制机制有时很容易忽略检查或者错误地实施了访问控制验证逻辑。由于与安全相关的代码分布在功能性的代码之间，所以访问控制验证经常是不完整或者不正确的。

(4)不安全的异常处理

如果执行代码的过程中发生了错误，那么错误会中断或干扰代码的正常流程并创建异常对象，程序将尝试寻找异常处理程序（一段告诉程序如何对错误作出响应的代码），以帮助程序恢复流程。

程序员应该考虑异常处理程序对安全的影响。一个典型的例子是高负载流量下防火墙的行为。如果防火墙不能及时地按照规则检查每个数据包，那么它要么丢弃所有的数据包，要么允许所有的数据包通过。如果防火墙选择后者，那么攻击者就成功地穿透了防火墙。

2.4 配置阶段

在配置阶段也会引入漏洞。如果软件配置的不正确，那么可能会导致产生各种漏洞。

(1)软件的重用

为了在相对友好环境中使用而编写的软件，通常会在不太友好的环境中被重用。由于程序员对程序的运行环境做出了假定，因此环境的变化可能引起安全漏洞。

随着面向组件程序设计方法的出现，软件在不友好的环境中的重用问题在设计和实现阶段也变得十分重要：重用一个组件要求除了知道组件的功能以外还要分析它的安全

属性。

(2)复杂的或者不必要的配置

一些系统的配置机制非常复杂，这使得配置很容易出错。虽然相关文档清楚地描述了配置选项，但是系统管理员通常不会花太多的时间阅读文档。此外，检测不正确的配置并非易事：一个安全漏洞不会影响系统的正常运行，只有在攻击发生时，人们才会发现它们。

所谓的社会工程攻击有时会利用配置方面产生的漏洞。在社会工程攻击中，攻击者试图说服受害者更改配置。缺乏安全经验的用户通常会遵从攻击者的指示，结果打开了安全漏洞。因此改变安全选项要三思而后行。

(3)默认配置的安全性

由于大多数用户和系统管理员都倾向于执行默认安装，因此保证默认配置的安全是十分重要的。然而，这涉及到安全性和易用性之间的折中：安全的默认配置会给用户带来使用上的不便。为了保证大多数用户在使用软件时不受到太多的约束，通用软件的默认配置通常是不安全的。

2.5 维护阶段

最后一类对在维护阶段引入到系统的漏洞进行了分类。从某种意义上说，这类起因包括所有其它的类别：随着软件不断增加特征和功能，在维护阶段开发人员将从事新的分析、设计、实现和配置工作，这与软件开发周期的瀑布模型是相符的，鉴于篇幅，在该类别中对前述部分不予赘述。

(1)特征冲突

随着越来越多的特征增加到软件中，它们之间的交互方式变得不可预见。这个现象被称作特征交互，它是常见的漏洞起因。

这类的典型例子是电话网。在电话网中，新服务的引入将导致许多对安全策略的违反。

(2)向后兼容性

维持向后兼容性和修订安全缺陷经常是相互矛盾的：如果在一个版本的软件中发现了安全缺陷，在软件的下一个版本中修订了缺陷，那么这经常会产生向后兼容性的问题。因此，修订版本有时考虑到向后兼容性问题会采用不安全的工作方式。显然，攻击者可能会利用这个漏洞。

3 结论与展望

本文提出了一个软件漏洞起因的分类方法，主要有两个目的：教育目的（通过熟悉软件漏洞的起因，开发人员能采取适当的措施避免它们）和作为测试和审计工具。

当然，这个分类方法也有不完善之处。我们希望随着它的使用，分类方法能不断地发展；随着新漏洞的发现，这个分类方法将增加新的类别。

参考文献

1 Bishop M. Vulnerability Analysis[C]. Proceedings of Recent Advances in Intrusion Detection, 1999: 125-136.

2 Landwehr C, Bull A, Mcdermott J, et al. A Taxonomy Computer Program Security Flaws, with Example[J]. ACM Computing Surveys, 1994, 26(3): 211-255.

3 Krsul I, Spafford E, Tripunitara M. Computer Vulnerability Analysis [R]. West Lafayette: COAST Laboratory, Purdue University, Technical Report: COAST TR98-07, 1998-05.

4 Aslam T, Krsul I, Spafford E. A Taxonomy of Security Faults[C]. Proceedings of the 19th National Information Systems Security Conference, Baltimore, Maryland, 1996-10.

5 Anderson R. Security Engineering, A Guide Building Dependable Distributed Systems[M]. John Wiley & Sons, 2001.

6 Gollmann D. Computer Security[M]. John Wiley & Sons, 2000.

7 Viega J, McGraw G. Building Secure Software[M]. Addison-Wesley, 2002.

8 Schneier B. Secrets and Lies: Digital Security in a Networked World [M]. John Wiley & Sons, 2000.

(上接第 139 页)

Rest 中的冗余域发出恢复的请求，Rest 中的冗余域收到请求后，就把自己当前的状态变量都发给 Voter，Voter 收到所有 Rest 中冗余域的对象的状态后，通过 Byzantine 协议选举出正确的状态标志传给 ORP。ORP 收到 Voter 传来恢复对象所有的状态标志。通过公用对象平台产生一个目标需要恢复的对象。然后把目标对象发送给请求恢复的冗余域，该冗余域收到新对象后，先去用新对象代替旧的错误对象，同时重新生产所有需要跟其他用户建立会话的会话密钥，然后再把新对象设为可用对象。这时新恢复的对象的状态肯定是正确的。这种恢复是一种向前恢复策略。

3 结束语

本文在研究 SITAR 模型的基础上提出了一种基于 CORBA 中间件分布式对象容忍入侵系统的模型，在该模型基础上给出了异构分布式对象的恢复策略，并通过一个实例说明了本文恢复策略的正确性。本文的进一步研究工作包括：ITDOS 恢复策略效率的度量和效率的提高；对大型对象的传输效率问题；在系统架构内，提出一种非常有效的方法去解决大消息的认证、授权和完整性检查。

参考文献

1 Deswarte Y, Blain L, Fabre J C. Intrusion Tolerance in Distributed Computing Systems[C]. Proc. of the International Symposium on Security and Privacy, Oakland, CA. IEEE Press, 1991.

2 Wang Dazhi, Madan B B, Trivedi K S. Security Analysis of SITAR Intrusion Tolerance System[C]. Proc. of SSRS '03, Fairfax, Virginia, USA, 2003-10-31.

3 OMG. The Common Object Request Broker: Architecture and Specification(Revision 2.5)[Z]. 2001.

4 Castro M, Liskov B. Proactive Recovery in a Byzantine-fault-tolerant System[C]. Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation, 2000-10.

5 Cachin C, Kursawe K, Shoup V. Random Oracles in Constantinople: Practical Asynchronous Byzantine Agreement Using Cryptography[C]. Proceedings of the 19th ACM Symposium on Principles of Distributed Computing, 2000: 123-132.