|  |
| --- |
|  |

Q/DX

青岛鼎信通讯股份有限公司技术文档

SS-01-0250

|  |
| --- |
|  |

安全评测标准

|  |
| --- |
| 水印 |
|  |

青岛鼎信通讯股份有限公司   发布

目  次

[1 编码安全 1](#_Toc491167505)

[1.1 安全编码原则 1](#_Toc491167506)

[1.1.1 最小功能性 1](#_Toc491167507)

[1.1.2 最小授权原则 1](#_Toc491167508)

[1.1.3 谨慎使用多任务和多进程 1](#_Toc491167509)

[1.1.4 界面输出最小化 1](#_Toc491167510)

[1.1.5 代码应尽量简单 1](#_Toc491167511)

[1.1.6 避免高风险的服务、协议 1](#_Toc491167512)

[1.1.7 最小授权 1](#_Toc491167513)

[1.1.8 设计错误、异常处理机制 1](#_Toc491167514)

[1.1.9 提供备份机制 2](#_Toc491167515)

[1.1.10 其他编码原则 2](#_Toc491167516)

[1.2 输入处理 2](#_Toc491167517)

[1.2.1 验证各种来源的输入 2](#_Toc491167518)

[1.2.2 对输入内容进行规范化处理后再进行验证 2](#_Toc491167519)

[1.2.3 选择合适的数据验证方式 2](#_Toc491167520)

[1.2.4 在服务端进行验证 3](#_Toc491167521)

[1.2.5 建立统一的输入验证接口 3](#_Toc491167522)

[1.2.6 控制写入日志的信息 3](#_Toc491167523)

[1.3 输出处理 3](#_Toc491167524)

[1.3.1 限制返回给客户的信息 3](#_Toc491167525)

[1.3.2 建立错误信息保护机制 3](#_Toc491167526)

[1.4 数据库访问 3](#_Toc491167527)

[1.4.1 合理分配数据库访问权限 3](#_Toc491167528)

[1.4.2 合理存放数据库连接帐号和密码信息 3](#_Toc491167529)

[1.4.3 对数据库操作的返回数据进行验证 3](#_Toc491167530)

[1.4.4 分次提取数据 4](#_Toc491167531)

[1.4.5 通过row（行）级别的访问控制来使用数据库 4](#_Toc491167532)

[1.4.6 确保数据库资源被释放 4](#_Toc491167533)

[1.5 文件操作 4](#_Toc491167534)

[1.5.1 对上传文件进行限制 4](#_Toc491167535)

[1.5.2 把文件名以及文件内容作为不可信的输入对待 4](#_Toc491167536)

[1.5.3 安全的使用文件名 4](#_Toc491167537)

[1.5.4 使用文件系统访问控制 4](#_Toc491167538)

[1.5.5 注意文件访问竞争条件 5](#_Toc491167539)

[1.5.6 禁止下载应用系统自身的配置和数据 5](#_Toc491167540)

[1.6 安全特征 5](#_Toc491167541)

[1.6.1 关注应用的对象重用 5](#_Toc491167542)

[1.6.2 用户访问控制信息的机密性 5](#_Toc491167543)

[1.6.3 不要在客户端存放敏感数据 5](#_Toc491167544)

[1.6.4 避免内存溢出 5](#_Toc491167545)

[1.6.5 可配置数据保护 5](#_Toc491167546)

[1.6.6 禁止在源代码中写入口令 6](#_Toc491167547)

[1.6.7 随机数 6](#_Toc491167548)

[1.6.8 异常管理 6](#_Toc491167549)

[1.7 WEB 开发要求 6](#_Toc491167550)

[1.7.1 使用POST不能使用GET 6](#_Toc491167551)

[1.7.2 创建默认的错误页面 7](#_Toc491167552)

[1.7.3 宜使用通用错误消息 7](#_Toc491167553)

[1.7.4 应使用强的会话ID 7](#_Toc491167554)

[1.7.5 在每次认证后创建一个新的会话 7](#_Toc491167555)

[1.7.6 保护Cookie 7](#_Toc491167556)

[1.7.7 允许用户终止其会话 7](#_Toc491167557)

[1.8 其它编码要求 7](#_Toc491167558)

[2 Java安全编程指南 8](#_Toc491167559)

[2.1 SQL注入防范 8](#_Toc491167560)

[2.2 跨站脚本攻击防范 9](#_Toc491167561)

[2.3 验证输入文件名 10](#_Toc491167562)

[2.4 外部程序调用漏洞 10](#_Toc491167563)

[2.5 整数溢出 11](#_Toc491167564)

[3 SQL注入 11](#_Toc491167565)

[3.1 漏洞描述 11](#_Toc491167566)

[3.2 漏洞危害 12](#_Toc491167567)

[3.3 产生原理描述 12](#_Toc491167568)

[3.4 解决方案 13](#_Toc491167569)

[4 跨站脚本 18](#_Toc491167570)

[4.1 漏洞描述 18](#_Toc491167571)

[4.2 漏洞危害 19](#_Toc491167572)

[4.3 产生原理描述 19](#_Toc491167573)

[4.4 反射型跨站 19](#_Toc491167574)

[4.5 存储型跨站 20](#_Toc491167575)

[4.6 解决方案 20](#_Toc491167576)

[5 失效的身份认证和会话管理 25](#_Toc491167577)

[5.1 漏洞描述 25](#_Toc491167578)

[5.2 漏洞危害 25](#_Toc491167579)

[5.3 产生原理描述 25](#_Toc491167580)

[5.4 解决方案 26](#_Toc491167581)

[6 不安全的直接对象引用 27](#_Toc491167582)

[6.1 漏洞描述 27](#_Toc491167583)

[6.2 漏洞危害 27](#_Toc491167584)

[6.3 产生原理描述 27](#_Toc491167585)

[6.4 解决方案 27](#_Toc491167586)

[7 跨站请求伪造 28](#_Toc491167587)

[7.1 漏洞描述 28](#_Toc491167588)

[7.2 漏洞危害 29](#_Toc491167589)

[7.3 产生原理描述 29](#_Toc491167590)

[7.4 解决方案 30](#_Toc491167591)

[8 安全配置错误 33](#_Toc491167592)

[8.1 漏洞描述 34](#_Toc491167593)

[8.2 漏洞危害 34](#_Toc491167594)

[8.3 产生原理描述 34](#_Toc491167595)

[8.4 解决方案 34](#_Toc491167596)

[9 不安全的加密存储 34](#_Toc491167597)

[9.1 漏洞描述 35](#_Toc491167598)

[9.2 漏洞危害 35](#_Toc491167599)

[9.3 产生原理描述 35](#_Toc491167600)

[9.4 解决方案 35](#_Toc491167601)

[10 越权访问 35](#_Toc491167602)

[10.1 漏洞描述 35](#_Toc491167603)

[10.2 漏洞危害 35](#_Toc491167604)

[10.3 产生原理描述 35](#_Toc491167605)

[10.4 解决方案 36](#_Toc491167606)

[11 传输层保护不足 36](#_Toc491167607)

[11.1 漏洞描述 36](#_Toc491167608)

[11.2 漏洞危害 36](#_Toc491167609)

[11.3 产生原理描述 37](#_Toc491167610)

[11.4 解决方案 37](#_Toc491167611)

[12 未验证的重定向和转发 37](#_Toc491167612)

[12.1 漏洞描述 37](#_Toc491167613)

[12.2 漏洞危害 37](#_Toc491167614)

[12.3 产生原理描述 37](#_Toc491167615)

[12.4 解决方案 37](#_Toc491167616)

[13 上传任意文件 39](#_Toc491167617)

[13.1 漏洞描述 39](#_Toc491167618)

[13.2 漏洞危害 39](#_Toc491167619)

[13.3 产生原理描述 39](#_Toc491167620)

[13.4 解决方案 39](#_Toc491167621)

[14 命令注入 40](#_Toc491167622)

[14.1 漏洞描述 40](#_Toc491167623)

[14.2 漏洞危害 40](#_Toc491167624)

[14.3 产生原理描述 40](#_Toc491167625)

[14.4 解决方案 42](#_Toc491167626)

[15 信息泄露和不恰当的错误处理 43](#_Toc491167627)

[15.1 漏洞描述 43](#_Toc491167628)

[15.2 漏洞危害 43](#_Toc491167629)

[15.3 产生原理描述 43](#_Toc491167630)

[15.4 解决方案 43](#_Toc491167631)

[16 目录浏览 44](#_Toc491167632)

[16.1 漏洞描述 44](#_Toc491167633)

[16.2 漏洞危害 44](#_Toc491167634)

[16.3 产生原理描述 44](#_Toc491167635)

[16.4 解决方案 45](#_Toc491167636)

[17 整数溢出 46](#_Toc491167637)

[17.1 漏洞描述 46](#_Toc491167638)

[17.2 漏洞危害 46](#_Toc491167639)

[17.3 产生原理描述 46](#_Toc491167640)

[17.4 解决方案 48](#_Toc491167641)

安全评测标准

# 编码安全

在编码阶段，开发人员的安全意识水平和安全编程能力将直接影响系统自身的安全性和健壮性。规范对开发人员的编码提出统一的安全要求，主要涉及安全编码原则、输入处理、输出处理、 数据访问、文件操作、安全特征、 WEB 开发要求等方面。

* 1. 安全编码原则
     1. 最小功能性

软件应只包含那确实真正需要的功能：

a ） 只运行明确定义的功能。

b ） 系统调用只在确实需要的时候。

c ） 只在确实需要的时候访问数据。

* + 1. 最小授权原则

应采用适当的身份验证和权限分配模式，确保程序功能实现严格遵循“最小权限”和 “需要知道”安全原则。

* + 1. 谨慎使用多任务和多进程

a） 应尽量使用单任务的程序。

b） 如果软件需要使用多任务和多进程，应同步所有的进程和任务以避免冲突。

* + 1. 界面输出最小化

用户界面应只提供必须的功能，确保用户不能通过用户界面直接访问数据或者直接访问被保护对象。

* + 1. 代码应尽量简单

a） 宜使用结构化的编程语言，尽量避免使用递归和Go to声明语句。

b） 宜使用简单的代码，清除不必要的功能，防止采用信息隐藏方式进行数据保护。

* + 1. 避免高风险的服务、协议

应尽量避免使用不加保护或已被证明存在安全漏洞的服务和通信协议传输文件。

* + 1. 最小授权

用户进程的授权宜采用最小授权法，对于软件的普通用户进程，禁止赋予该类进程特权用户权限。

* + 1. 设计错误、异常处理机制

宜建立防止系统死锁的机制，异常情况的处理和恢复机制：具体包括错误和异常检测、数据回滚、安全错误通知、错误和异常记录、断点保护等。

* + 1. 提供备份机制

为保证运行数据的完整性和可用性，宜设计有效的备份策略，根据业务和系统维护需要提供定期或不定期、自动或手动方式的备份机制。

* + 1. 其他编码原则

a）当一个进程对敏感对象（包含秘密信息或包含不可更改信息）使用完后，应立即清除对象的敏感信息，然后再删除对象。任何不需要使用的资源应及时释放。

b）当输入不可信任数据时，要在该数据的内容和格式上同时加以检查。对于整数，应该检查数据大小是否超出了能够表示的范围；对于输入的字符串，要确保长度没有溢出，保证每一个字符都是有效的。

c）禁止使用通配符的方式进行数据的插入操作。

d）应对面向用户的操作反馈缺省描述进行必要的封装，删除有关后台系统或其它敏感信息。

e）在需要进行文件操作时，应预先设定目前工作路径，使用相对路径表示文件的位置。

f）宜检查所有系统函数调用（System Calls）所传递变量的合法性。

g）应检查所有函数调用的返回代码，尤其是错误代码。

* 1. 输入处理
     1. 验证各种来源的输入

程序默认情况下应对所有的输入信息进行验证，不能通过验证的数据应会被拒绝。宜对如下输入进行验证：

a） HTTP请求消息的全部字段，包括GET数据、POST数据、COOKIE和Header数据等。

b） 不可信来源的文件。

c） 第三方接口数据。

d） 从数据库中检索出的数据。

e） 对来自命令行以及配置文件的输入。

f） 网络服务。

g） 注册表值。

h） 系统性能参数。

i） 临时文件。

* + 1. 对输入内容进行规范化处理后再进行验证

当输入数据包含文件名、路径名、URL 等数据时，应先对输入内容进行规范化处理后再进行验证，如文件路径、URL地址等数据，需要规范化为标准格式后再进行验证。

* + 1. 选择合适的数据验证方式

应根据情况综合采用多种输入验证的方法，包括：

a） 检查数据是否符合期望的类型。

b） 检查数据是否符合期望的长度。

c） 检查数值数据是否符合期望的数值范围。比如检测整数输入的最大值与最小值。

d） 检查数据是否包含特殊字符，如：<、>、 "、 '、%、 (、)、 &、+、\、\'、\"等。

e） 应使用正则表达式进行白名单检查尽量避免使用黑名单法。

* + 1. 在服务端进行验证

仅在客户端进行验证是不安全的，应在客户端验证的同时，在服务器端也进行验证。

* + 1. 建立统一的输入验证接口

应建立统一的输入验证接口，为整个应用系统提供一致的验证方法。

* + 1. 控制写入日志的信息

如日志数据中包含输入数据，应对输入数据进行验证，禁止攻击者能够写任意的数据到日志中。

* 1. 输出处理
     1. 限制返回给客户的信息

编码时应限制返回给客户与业务处理无关的信息，禁止把重点保护数据返回给不信任的用户，避免信息外泄。

* + 1. 建立错误信息保护机制

a） 禁止将详细错误信息直接反馈到客户端，详细错误信息中包含系统信息、文件和目录的绝对径信息。

b） 应对错误信息进行规整和清理后再返回到客户端。

c） 宜只向客户端返回错误码，详细错误信息可记录在后台服务器。

* 1. 数据库访问
     1. 合理分配数据库访问权限

a） 应按照“最小化原则”为应用程序分配数据库访问权限。

b） 避免为应用程序分配过大或不必要的数据库权限。

c） 禁止将数据库DBA权限分配给应用程序。

* + 1. 合理存放数据库连接帐号和密码信息

a） 禁止在应用程序代码和配置文件明文存放数据库连接帐号和密码信息。

b） 数据库连接账户和密码信息可加密后再保存在配置文件中。

* + 1. 对数据库操作的返回数据进行验证

a） 应检查SQL请求的返回记录数。

b） 数据库的数据做为输入数据时，宜进行验证。

* + 1. 分次提取数据

对于数据库查询操作，如果查询返回的结果较多时，宜设计成分次提取，应避免一次提取过多数据。

* + 1. 通过row（行）级别的访问控制来使用数据库

a） 不能依赖应用程序访问控制能够保护数据库的数据，应限制每个请求使用户只能访问他们自己的数据。

b） 应限制SQL请求只向当前认证的用户返回结果。

* + 1. 确保数据库资源被释放

a） 应保证数据库访问在不需要使用的时候被释放，例如连接、游标等。

b） 由于资源泄露会导致系统出错而且很难捕捉到，宜建立一个资源管理模块并且完全按照规则进行操作。

c） 宜禁止依赖Java 和.NET 的垃圾回收器来回收资源。

1） 在Java中，应该在finally 块中释放资源来保证资源在任何环境下都会被释放。

2） 在.NET中，应需要使用关键字 using 引入IDisposable 接口来进行资源释放，而不是直接关闭管理资源的对象。

* 1. 文件操作
     1. 对上传文件进行限制

允许用户上传文件时，应对上传文件做如下限制：

a） 上传文件类型限制应遵循最小化原则，通过文件检查仅允许上传必须的文件类型。

b） 上传文件大小限制，限制文件的容量大小范围。

c） 文件保存路径限制，过滤文件名或路径名中的特殊字符（../或..\等）避免文件保存在非预期目录中。

d） 应关闭文件上传目录的执行权限，在 UNIX/LINUX 系统环境里，建议把上传目录挂载成独立的逻辑盘或设置为jail环境。

* + 1. 把文件名以及文件内容作为不可信的输入对待

对来自文件系统的所有值都应进行合适的输入验证，确保从文件系统中读取的数据符合期望。

* + 1. 安全的使用文件名

a） 应避免在传递的参数中直接使用真实的文件名，尽量使用索引值来映射实际的文件路径。如确实需要在参数中出现文件名，则尽量对文件名进行白名单检查。

b） 应禁止在参数中出现../、..\等特殊字串及其变形。

* + 1. 使用文件系统访问控制

a） 文件只应被指定的用户访问，而且该用户应该被限制为最小权限，例如对文件的读写权限都应被限制到最低。

b） 当应用程序使用被其控制的已存在文件时，应首先验证文件的权限和属组，防止文件存在被篡改的许可权限。

* + 1. 注意文件访问竞争条件

多进程或线程对同一文件进行访问时，应采取适当的加锁策略保证文件数据在访问过程中的一致性。 安全使用临时文件 在程序初始化时宜以最严格的权限策略建立一个安全临时文件夹，该文件夹只有该程序具备读写权限，其他用户无法访问。将所有临时文件都存放在该文件夹中，当临时文件使用完毕后应及时清除临时文件。

* + 1. 禁止下载应用系统自身的配置和数据

a） 应将应用系统自身的配置和数据文件保存在WEB空间之外，防止攻击者采用URL探测等手段尝试下载应用系统的配置和数据文件。

b） 应禁止在脚本的动态包含功能中直接使用用户提交的数据和文件。

* 1. 安全特征
     1. 关注应用的对象重用

a） 对于底层系统的对象可重用性来说，宜提供应用软件对敏感数据使用后立即覆盖的能力，这些敏感数据包括口令、安全密钥、会话密钥或者其它的高度敏感的数据。

b） 应提高代码的重复利用率，可创建公共函数库（对象库）供整个软件程序调用。

* + 1. 用户访问控制信息的机密性

应禁止在程序代码中和配置文件中直接明文写入用户名和口令等用户访问控制信息。这些控制信息应加密后存放在配置文件中。

* + 1. 不要在客户端存放敏感数据

由于客户端是不可信任的，应禁止客户端存放敏感数据。特别是在使用 Cookie 时不能把客户重要信息储存在客户端。

* + 1. 避免内存溢出

为防止内存溢出，应注意以下事项：

a） 在对缓存区填充数据时应进行边界检查，应判断是否超出分配的空间。

b） 应保证系统资源及时释放和服务连接的及时关闭, 应显式关闭、释放使用过的资源（如连接对象、文件句柄等），不要依赖垃圾收集器。

c） 软件程序应检查每次内存分配是否失败，并进行处理。

d） 应及时释放内存资源，防止内存泄漏，但要避免重复内存释放。

* + 1. 可配置数据保护

a） 限制非应用软件用户访问可配置数据。

b） 可采用系统访问控制为配置数据文件设置严格的访问权限，仅应用程序可以访问。

* + 1. 禁止在源代码中写入口令

a） 应将加密后的口令存储在配置文件，数据库或者其它外部数据源中。

b） 应禁止将口令存储在代码中，把口令存储在代码中会导致任何人都可以获得到存储在代码中的口令。

c） 应禁止将口令写在发布的软件中，如果需要修改口令，则软件必需通过安装程序补丁等方式替换程序文件，会造成口令难于修改。

* + 1. 随机数

如果应用程序需要随机数，需要找出一种合适的随机数生成方法，要求其生成代价较小并且应满足安全需求。

a） 在Java中，宜使用SecureRandom类生成随机数，不要使用Random类。通常不需要设置种子数给SecureRandom，Java 会自动获取一个信息熵比较平均的值。

b） 在C和C++中，应避免使用标准的随机数函数，如rand(), srand(), srand48(),drand48(), lrand48(), random()和srandom()。

c） 如果操作系统或者应用程序使用的加密库有一个真实的随机源，宜使用该随机源。

* + 1. 异常管理

a） 捕捉并处理异常。

1） 应使用结构化异常处理机制，捕捉并处理异常现象。

2） 宜避免将应用程序置于不协调的状态，该状态可能会导致信息泄漏或拒绝服务攻击。

b） 记录详细的错误信息。

1） 宜在错误日志中记录详细的错误消息。

2） 宜向服务或应用程序的客户发送最少量的信息，确保没有密码或其他敏感数据。如一般性错误消息和自定义错误日志 ID，随后可以将这些信息映射到事件日志中的详细消息。

c） 不要向客户端泄漏信息。

1） 发生故障时，严禁向客户端泄漏信息，禁止暴露的内容包括函数名以及调试内部版本时出问题的详细信息。

2） 宜向客户端返回一般性错误消息。可以将应用程序设置为不向远程用户显示详细错误信息，也可以选择将错误重定向到应用程序页。

3） 宜捕捉所有未处理异常并将它们发送到一般错误页的页级别或应用程序级别上，创建全局错误处理程序。

* 1. WEB 开发要求
     1. 使用POST不能使用GET

使用 Get 方法传递的参数将包含在 URL 里，这些信息会被记录在日志文件中或通过 HTTP 头的Referrer 发送到其它站点上，被存储在浏览器的历史记录中。当参数中包含重要信息时，这些信息也会被记录，这将增大泄密的风险。

* + 1. 创建默认的错误页面

应为HTTP错误创建一个默认的错误页面，丢弃所有的异常，防止攻击者从应用程序的默认出错页面中得到系统信息。

* + 1. 宜使用通用错误消息

应构造错误提示信息来防止诸如用户 id 、网络、应用程序以及服务器环境的细节等重要敏感信息的泄漏。包括：

a） 不区分错误的用户名和错误的密码；

b） 在返回的报告中不能包含主机信息、网络 DNS 信息、软件版本信息、错误代码或者其它错误的详细信息；

c） 不允许把错误的细节放在错误页面的注释里。

* + 1. 应使用强的会话ID

a） 会话id 长度应不能低于64位，建议使用128位长度的会话id。

b） 要求在确定 session 的 id 长度以及生成 id 的随机种子之前，不要相信 web 程序的容器。因为会话id太短很容易被暴力破解。如果攻击者猜测到授权用户的会话id就可以接管用户的会话。

* + 1. 在每次认证后创建一个新的会话

在用户认证成功后应先将之前的用户会话销毁，然后重新为用户生成新的会话，保证认证成功前后，用户会话ID不一样。

* + 1. 保护Cookie

a） 要求在建立Session时用户以SSL方式连接，并且将该会话的安全标识设置为cookie 只能通过安全连接进行传输。

b） 不能只依赖HttpOnly cookie带来的任何安全保障，因为其只限于在IE下使用，攻击者也可以很方便的窃取到它。

* + 1. 允许用户终止其会话

a） 宜允许用户通过注销来保护自己的帐号。

b） 应在会话终止时立即销毁所有会话数据，否则该数据会被攻击者利用。

* 1. 其它编码要求

a） 应把数据与程序放置在不同的目录中。

b） 变量在使用前应进行初始化。

c） 应留意字节大小差异、精度、符号数/非符号数的区别、舍位截断问题、不同类型数据的转化、非数值运算、极值处理等问题，避免意外的变量使用和运算导致流程的改变。

d） 调用函数时应对返回值进行检查。

e） 应使用可靠的随机数生成算法，并设置合适的随机数种子。

f） 应使用成熟的、高强度的加密算法，禁止使用自创加密算法。

g） 宜避免使用HTTP GET方法。

h） 宜禁止使用与业务功能无关的第三方代码或库文件，应检查已有第三方代码或库文件的版本和安全性，避免引入安全漏洞。

i） 应及时释放内存资源，防止内存泄漏，但要避免重复内存释放。

j） 应显式关闭、释放使用过的资源（如连接对象、文件句柄等），不要依赖垃圾收集器。

# Java安全编程指南

* 1. SQL注入防范

SQL注入指利用现有应用程序，将(恶意)的SQL命令注入到后台数据库引擎执行的攻击方法。Java程序存在SQL注入问题，能通过使用PreparedStatement类来阻止SQL注入。

一段可能产生SQL注入的代码如下：

|  |
| --- |
| String user = request.getParameter("username");  String pass = request.getParameter("password");  String query = "SELECT id FROM users WHERE username="+user+" AND  password="+pass;  Statement stmt = con.createStatement(query);  ResultSet rs = con.executeQuery(query);  if (rs.next())  {  // 登录成功  int id = rs.getInt(1);  ...  }  else  {  // 登录失败  ...  } |

正确的做法如下：

|  |
| --- |
| String user = request.getParameter("username");  String pass = request.getParameter("password");  String query = "SELECT id FROM users WHERE username=? AND password=?";  PreparedStatement stmt = con.prepareStatement(query);  stmt.setString(1, user);  stmt.setString(2, pass);  ResultSet rs = stmt.executeQuery();  if (rs.next())  {  // 登录成功  int id = rs.getInt(1);  ...  }  else  {  // 登录失败  ...  } |

* 1. 跨站脚本攻击防范

跨站脚本漏洞本质是 Html 的注入问题，恶意用户的输入没有经过严格的控制并最终显示给来访的用户，导致能在来访用户的浏览器里以浏览用户的身份执行Html代码，数据流程如下：

恶意用户的Html输入—>web程序—>数据存储—>web程序—>用户浏览器。

Html代码的示例：

|  |
| --- |
| <img src="http://www.loveshell.jpg" width=100 onerror=alert("载入图片错误!")> |

<>是浏览器的一个 Html 标记，img 是<>标记的名称，src 是<>的第一个属性，=后面是 src 的值，width 是第二个属性，onerror 是标记的事件属性。个 Html 标记包括很多元素，并不是输入<>才会存在HTML注入，其实只要你的输入处在Html标签内，产生了新的元素或者属性，就产生跨站脚本攻击！

对输出做做HTML关键字过滤，能防止跨站脚本。

一段可能产生跨站脚本攻击漏洞的代码如下：

|  |
| --- |
| String name = request.getParameter(“name”);  out.println(“hello, welcome <b>” + name + “</b> !”); |

正确的做法如下，应该至少过滤’（单引号）、”（双引号）、<、>、TAB 键、&、#、script。

|  |
| --- |
| String name = request.getParameter(“name”);  out.println(“hello, welcome <b>” + htmlEncode(name) + “</b> !”);  ….  public String htmlEncode(String sText) {  sText = sText.replaceAll(“&”, “&amp;”);  sText = sText.replaceAll(“#”, “＃”);  sText = sText.replaceAll(“’”, “&quote;”);  sText = sText.replaceAll(“\””, “&quote;”);  sText = sText.replaceAll(“<”, “&lt;”);  sText = sText.replaceAll(“>”, “&gt;”);  sText = sText.replaceAll(“script”, “”);  } |

防范跨站脚本攻击应过滤以下特殊字符：

|  |
| --- |
| 特殊/关键字符  '(单引号)  "(双引号)  <  >  空格键  TAB 键  Script  & |

* 1. 验证输入文件名

从用户接收输入文件名时，应确保文件名具有严格的格式。

一段有漏洞的程序：

|  |
| --- |
| String fileName = request.getParameter("filename");  String content = request.getParameter("content");  FileWriter fw = new FileWriter(new File(“/home/y/share/templates/” + fileName));  fw.write(content); |

应该对文件名做严格的检查，比如使用正则表达式，限定文件名只能包括字母和数字。

* 1. 外部程序调用漏洞

应避免在程序中调用系统命令，如果必须使用系统命令，应对输入参数做严格的过滤，一段存在漏洞的程序如下：

String ip = request.getParameter(“ip”);

Process p = Runtime.getRuntime().exec(“/bin/sh nslookup ” + ip);

应使用正则表达式对输入的参数（IP）做严格过滤，否则用户可以提交如下格式的IP数据：

192.168.0.1; [自己的命令]来执行自己的命令。

* 1. 整数溢出

一段存在漏洞的程序如下：

|  |
| --- |
| public void sample(byte[] b, int off, int len) {  if (len < 0 || off < 0 || len + off > b.length) {  // do A  ……  } else {  // do B  ……  }  } |

正确的做法如下：

|  |
| --- |
| public void sample(byte[] b, int off, int len) {  if (len < 0 || off < 0 || len > b.length - off) {  // do A  ……  } else {  // do B  …  }  } |

# SQL注入

* 1. 漏洞描述

SQL注入是指利用现有应用程序，将(恶意)的SQL命令注入到后台数据库引擎执行的攻击方法。通常 Web 应用程序在后端使用数据库，查询数据库的语言一般是 SQL（各大数据库供应商都有自己的不同版本）。Web应用程序通常会获取用户输入（取自 HTTP 请求），将它并入 SQL 查询中，然后发送到后端数据库。接着应用程序通过处理查询结果，最终向用户显示结果。 如果应用程序对用户（攻击者）的输入处理不够安全，攻击者能够利用 SQL 发起攻击。在此情况下，攻击者能够注入恶意的数据，当该数据并入 SQL 查询中时，将修改原来的查询语句使攻击者能进行敏感数据查询等恶意操作。

* 1. 漏洞危害

恶意攻击者能够利用SQL注入攻击造成危害：

a） 查看、修改或删除数据库条目和表；

b） 篡改网页内容，网页挂马；

c） 查询用户的机密信息，包括账户、个人私密信息、交易信息等等；

d） 访问数据库系统表，查询数据库及承载服务器敏感信息；

e） 获取数据库访问权限，甚至获得DBA权限；

f） 运行各种操作系统命令，获取承载主机操作系统和网络的访问权限。

* 1. 产生原理描述

如果应用程序使用用户的输入（如用户名和密码）来查询用户帐户的数据库表，以认证用户，而攻击者能够将恶意数据注入查询的用户名部分（和/或密码部分），查询被更改成完全不同的数据查询，能够修改数据库的查询，或在数据库服务器上运行 Shell 命令的查询。

下面的代码动态构建和执行一个 SQL 查询，查找与给定名称匹配的 item。查询限定只有当当前用户名与item的所有者名称匹配时，才向当前用户显示item。

|  |
| --- |
| String user Name = ctx.getAuthenticatedUserName();  String user Name =requwst.getParameter(“itemName”);  String query = “SELECT\*FROM items WHERE owner =”  + username +”’AND itemname = “’  + itemName + “ “;  ResultSet rs = stmt.execute(query );  … |

代码中的查询原本希望执行如下语句：

|  |
| --- |
| SELECT\*FROM items  WHERE owner = <userName>  AND itemname = <itemName>; |

然而，因为查询语句是通过连接常量字符串和用户输入的字符串来动态构造，所以只有当itemName中不包含单引号字符时，查询才能正确执行。如果一个用户名为wiley的攻击者输入字符串” ‘name’ OR ‘a’ = ‘a’”作为itemName的值，那么查询会变成如下形式：

|  |
| --- |
| SELECT\*FROM items  WHERE owner = ‘wiley’  AND itemname = ‘name’ OR ‘a’ = ‘a’; |

额外的条件OR ‘a’ = ‘a’导致了子句的值总是为true，于是该查询语句从逻辑上来说等价于：

SELECT \* FROM items

如果用户名为wiley的攻击者输入字符串‘name’；DELETE FROM items;--”的值，那么查询语句会变成如下两个查询：

|  |
| --- |
| SELECT\*FROM items  WHERE owner = ‘wiley’  AND itemname = ‘name’  DELETE FROM items;  --’ |

很多数据服务器，包括 Microsoft(R)SQL Server2000，都支持依次执行用分号隔开的多个 SQL 语句。这种攻击语句在 Oracle 和其他不支持批量处理分号隔开的 SQL 语句的数据库服务器上会导致错误，但在允许批量处理的数据库上，这种攻击使得攻击者能够针对数据库执行任意指令。 连字符--将语句的剩余部分当作是注释而不去执行，能够把修改后的查询语句中最后的单引号给注释掉。

* 1. 解决方案

针对SQL注入攻击的特点，应从以下几个方面进行防范：

a） 使用参数化方式进行查询

应尽可能避免使用拼接的动态 SQL 语句，而使用参数化方式进行查询。一段示例代码如下所示：

|  |
| --- |
| String user = request.getParameter("username");  String pass = request.getParameter("password");  String query = "SELECT id FROM users WHERE username=? AND password=?";  PreparedStatement stmt = con.prepareStatement(query);  stmt.setString(1, user);  stmt.setString(2, pass);  ResultSet rs = stmt.executeQuery();  if (rs.next())  {  // 登录成功  int id = rs.getInt(1);  ...  }  else  {  // 登录失败  ...  } |

b） 检查用户输入的有效性

应在服务端验证用户输入的值和类型是否符合程序的预期要求，一般应验证以下内容：

1） 必需字段

必需字段：检查字段不为空，并且其长度要大于零，不包括行距和后面的空格，示例代码如下:

|  |
| --- |
| // Java 示例-必需字段检查  public Class Validator {  ...  public static boolean validateRequired(String value) {  boolean isFieldValid = false;  if (value != null && value.trim().length() > 0) {  isFieldValid = true;  }  return isFieldValid;  }  ...  }  ...  String fieldValue = request.getParameter("fieldName");  if (Validator.validateRequired(fieldValue)) {  // fieldValue is valid, continue processing request  ...  } |

2） 字段数据类型

字段数据类型检查：检查输入字段类型是否符合预期要求，比如说所有 HTTP 请求参数或 cookie 值的类型都是“字符串”。

|  |
| --- |
| // Java 示例-验证数字字段类型  public Class Validator {  ...  public static boolean validateInt(String value) {  boolean isFieldValid = false;  try {  Integer.parseInt(value);  isFieldValid = true;  } catch (Exception e) {  isFieldValid = false;  }  return isFieldValid;  }  ...  }  // Java 示例-将HTTP 请求参数转换为其各自的数据类型  ...  String fieldValue = request.getParameter("fieldName");  if (Validator.validateInt(fieldValue)) {  // convert fieldValue to an Integer  Integer integerValue = Integer.getInteger(fieldValue);  // store integerValue in a request attribute  request.setAttribute("fieldName", integerValue);  }  ...  // Use the request attribute for further processing  Integer integerValue = (Integer)request.getAttribute("fieldName");  ... |

3） 字段长度

字段长度检查：检查输入字段长度是否符合预期要求。

|  |
| --- |
| // Java 示例-userName 字段的长度是否在 8 至 20 个字符之间  public Class Validator {  ...  public static boolean validateRange(int value, int min, int max) {  return (value >= min && value <= max);  }  ...  }  ...  String fieldValue = request.getParameter("numberOfChoices");  if (Validator.validateRequired(fieldValue)) {  if (Validator.validateInt(fieldValue)) {  int numberOfChoices = Integer.parseInt(fieldValue);  if (Validator.validateRange(numberOfChoices, 10, 20)) {  // numberOfChoices is valid, continue processing request  ...  }  }  } |

4） 字段选项

字段选项检查：根据功能需求定义的允许选项来验证用户输入的数据。

|  |
| --- |
| // 字段选项示例  public Class Validator {  ...  public static boolean validateOption(Object[] options, Object value) {  boolean isValidValue = false;  try {  List list = Arrays.asList(options);  if (list != null) {  isValidValue = list.contains(value);  }  } catch (Exception e) {  }  return isValidValue;  }  ...  }  ...  // Allowed options  String[] options = {"option1", "option2", "option3");  // Verify that the user selection is one of the allowed options  String userSelection = request.getParameter("userSelection");  if (Validator.validateOption(options, userSelection)) {  // valid user selection, continue processing request  ...  } |

5） 字段模式

字段模式检查：检查用户输入是否与功能需求定义的模式匹配。例如，使用以下正则表达式：^[a-zA-Z0-9]\*$验证 userName 字段是否符合“仅允许字母数字字符，且不区分大小写”。

|  |
| --- |
| import org.apache.regexp.RE;  import org.apache.regexp.RESyntaxException;  public Class Validator {  ...  public static boolean matchPattern(String value, String expression) {  boolean match = false;  if (validateRequired(expression)) {  RE r = new RE(expression);  match = r.match(value);  }  return match;  }  ...  }  ...  // Verify that the userName request parameter is alpha-numeric  String userName = request.getParameter("userName");  if (Validator.matchPattern(userName, "^[a-zA-Z0-9]\*$")) {  // userName is valid, continue processing request  ...  } |

6） 对用户输入进行过滤

在服务器端对用户输入进行过滤。针对关键字和非法的 HTML 代码等，可以编写函数对

其进行检查或过滤。需要检查或过滤的特殊字符至少包含如下表.所示：

特殊字符

特殊字符（不区分大小写） 关键字（不区分大小写）

|  |  |
| --- | --- |
| |（竖线符号）  & （& 符号）  ;（分号）  $（美元符号）  %（百分比符号）  @（at 符号）  '（单引号）  "（引号）  \'（反斜杠转义单引号）  \"（反斜杠转义引号）  <>（尖括号）  ()（括号）  +（加号）  CR（回车符，ASCII 0x0d）  LF（换行，ASCII 0x0a）  ,（逗号）  \（反斜杠）  \*（星号） | And  Exec  Insert  Select  Delete  Update  Count  Chr  Mid  Master  Truncate  Char  Declare  Backup  script |

7） 加固应用程序服务器和数据库，利用最低权限账户与数据库连接。

配置可信任的IP接入和访问（例如，IPSEC），来控制哪些机器能够与数据库服务器通信。

从数据库服务器上移除所有的示例脚本和应用程序。

不应使用sa、dba、admin等具备数据库DBA权限的账户，为每一个应用程序的数据库的连接账户使用一个专用的最低权限权账户。如果应用程序仅需要读取访问，应将数据库的访问限制为只读。

应从生产数据库中移除未用的存储过程。

将对应用程序的访问仅授权给用户创建的存储过程。

禁止应用程序访问不必要的系统存储过程。

应用程序尽量使用存储过程，利用存储过程，将数据访问抽象化，使用户不能直接访问表或视图。

# 跨站脚本

* 1. 漏洞描述

跨站脚本攻击（也称为XSS）指利用网站漏洞从用户端恶意盗取信息。用户在浏览网站、使用即时通讯软件、甚至在阅读电子邮件时，点击其中的链接，攻击者通过在链接中插入恶意代码，系统在接收到包含恶意代码的请求之后会产成一个包含恶意代码的页面，和合法页面相同，当用户浏览器显示该页面内容时会执行攻击者插入的而恶意代码，攻击者就能够盗取用户信息。

* 1. 漏洞危害

攻击者能够通过跨站脚本攻击造成如下危害：

a） 盗取各类用户帐号，如机器登录帐号、用户网银帐号、各类管理员帐号；

b） 盗窃企业重要的具有商业价值的资料；

c） 实施非法转账；

d） 强制发送电子邮件；

e） 网站挂马；

f） 控制受害者机器向其它网站发起攻击；

g） 控制企业数据，包括读取、篡改、添加、删除企业敏感数据的能力。

* 1. 产生原理描述

发送到 web 浏览器的恶意内容以 JavaScript 段的形式出现，但也包括 HTML、Flash 或者其他浏览器能够执行的代码类型。基于XSS攻击一般包括：传送私有数据（如cookies或者其他session信息）给攻击者、将受害者的浏览器重定向到攻击者控制的 web 内容中或者假借有漏洞的站点在用户机器上进行其他恶意操作。

跨站脚本（XSS）发生在以下两种情况：

a） 反射型跨站，数据通过不可靠的源进入Web 应用，一般是web request；

b） 存储型跨站，包含在动态内容中的数据在没有经过安全检测就存储到数据库中提供给其他用户使用。

* 1. 反射型跨站

下面是一段存在反射型跨站脚本漏洞的代码：test.jsp 代码段从一个 HTTP request 中读雇员的eid，并将它显示出来。

|  |
| --- |
| <% String eid = request.getParameter("eid"); %>  ...  Employee ID: <%= eid %>  当采用如下方式访问 test.jsp 时，eid 包含了 HTML 代码，该 HTML 代码就会被浏览器  执行：  http://XXXXX/test.jsp?eid=<script>alert(document.cookie)</script> |

在eid变量中插入的代码为显示当前浏览器的cookie信息。

正如上面的示例所述，造成 XSS 攻击的原因是 HTTP 响应中包含了未经验证的数据，数据直接从HTTP 请求中读取，在 HTTP 响应中反射回去。反射型 XSS 通常是由攻击者诱使用户向有漏洞的 web应用程序提供危险内容，然后危险内容会反射给用户并由浏览器执行。传递恶意内容最常见的方法是将恶意内容作为一个参数包含在URL中，然后将URL公开发布或者通过email发给受害者。以这种形式构造的URL是很多钓鱼（phishing）攻击的核心，通过钓鱼攻击，攻击者获取受害者的信任，诱使受害者访问指向有漏洞站点的 URL。当站点将攻击者的内容反射给用户后，恶意内容将被执行，从用户机器上窃取用户的私有信息（如包含session信息的cookies）发送给攻击者或者进行其他恶意活动。

* 1. 存储型跨站

下面是一段存在存储型跨站脚本漏洞的代码：JSP 代码段根据一个雇员的 ID 在数据库中查询对应的名字，并将其打印出来。如果数据库中的数据来源于用户的输入并且没有进行相应的安全性检查，将会造成用户的浏览器执行恶意代码。

|  |
| --- |
| <%...  Statement stmt = conn.createStatement();  ResultSet rs = stmt.executeQuery("select \* from emp where  id="+eid);  if (rs != null)  rs.next();  String name = rs.getString("name");  %>  Employee Name: <%= name %> |

与反射型跨站示例一样，当 name 符合规范的时候，这段代码能够正确运行。但当 name 有问题时，代码段中并没有任何措施来阻止其运行。如果name 的值来自于用户所提供的数据，那么数据库将会成为恶意内容的渠道。没有对数据库中存储的所有数据进行适当的输入验证，那么攻击者就能够在用户的浏览器上执行恶意内容。这种类型的攻击被称为存储型XSS。

* 1. 解决方案

针对跨站脚本攻击漏洞，应该从输入或输出两个途径解决该问题：

a） 验证输入的数据类型是否正确

应在服务器端验证输入的数据类型是否正确。 验证数字字段（int 类型）的方式：

|  |
| --- |
| public Class Validator {  ...  public static boolean validateInt(String value) {  boolean isFieldValid = false;  try {  Integer.parseInt(value);  isFieldValid = true;  } catch (Exception e) {  isFieldValid = false;  }  return isFieldValid;  }  ...  }  ...  // check if the HTTP request parameter is of type int  String fieldValue = request.getParameter("fieldName");  if (Validator.validateInt(fieldValue)) {  // fieldValue is valid, continue processing request  } |

b） 使用白名单对输入数据进行验证

应在服务器端采用白名单对输入数据进行验证，拒绝包含HTML特殊字符的输入。

例如，如果 userName 字段应仅允许字母数字字符，且不区分大小写，则使用以下正则表达式：^[a-zA-Z0-9]\*$ 对userName字段数据进行验证。

Java 1.3 或更早的版本不包含任何正则表达式包。建议应将“Apache 正则表达式包”与 Java 1.3 一起使用。

|  |
| --- |
| import org.apache.regexp.RE;  import org.apache.regexp.RESyntaxException;  public Class Validator {  ...  public static boolean matchPattern(String value, String expression) {  boolean match = false;  if (validateRequired(expression)) {  RE r = new RE(expression);  match = r.match(value);  }  return match;  }  ...  }  ...  // 此处校验userName  String userName = request.getParameter("userName");  if (Validator.matchPattern(userName, "^[a-zA-Z0-9]\*$")) {  // userName 验证通过后的代码  ...  } |

c） 使用黑名单对输入数据进行安全检查或过滤

在服务器端，针对非法的 HTML 代码包括单双引号等，应编写函数对其进行检查或过滤。需要检查或过滤的特殊字符至少包含如下表字符：

|  |
| --- |
| 特殊/关键字符(不区分大小写)  '(单引号)  "(双引号)  <  >  空格键  TAB 键  script  &  #  %  +  $  ()圆括号  xss  expression |

示例代码如下：

|  |
| --- |
| 检查函数的示例代码如下：  public int XssCheck(pValue)  {  pValue = pValue.toUpperCase();    String[] strCross = {"SCRIPT","<",">","\"","'","&","#",”%”,”+”,”$”,”(”,”)”,"  ","\t","XSS","EXPRESSION"};  for(int i = 0;i<strCross.length;i++){  if(pValue.indexOf(strCross[i])>=0){  return 1;  //包含HTML特殊字符  }  }  return 0;  //数据不包含HTML特殊字符  } |

d） 采用开发框架自带的标签输出方式

禁止采用<% =pValue %>不安全的输出方式，应该采用标签形式输出，如：<s:hidden name="id" value="%{secAppConf.id}" />，采用标签方式输出时，系统在默认会自动对数据做HTML转换。

e） 对输出数据进行净化

应对输出数据进行过滤和转义，将敏感字符转换为其对应的实体字符来清理 HTML 特殊字符，保护应用程序免遭跨站点脚本漏洞攻击。

|  |
| --- |
| ...  public static String filter(String value) {  if (value == null) {  return null;  }  StringBuffer result = new StringBuffer(value.length());  for (int i=0; i<value.length(); ++i) {  switch (value.charAt(i)) {  case '<':  result.append("&lt;");  break;  case '>':  result.append("&gt;");  break;  case '"':  result.append("&quot;");  break;  case '\'':  result.append("&#39;");  break;  case '%':  result.append("&#37;");  break;  case ';':  result.append("&#59;");  break;  case '(':  result.append("&#40;");  break;  case ')':  result.append("&#41;");  break;  case '&':  result.append("&amp;");  break;  case '+':  result.append("&#43;");  break;  default:  result.append(value.charAt(i));  break;  }  return result;  }  ... |

f） 将Cookie设置为HttpOnly

攻击 Web 应用大多数是为了获得合法用户的 cookie 信息，所以应该减少大量的数据存储在 cookie中，在任何可能的时候使用HttpOnly cookie。

HttpOnly cookie是某些浏览器所支持的一种防御机制，应用程序能使用它来防止XSS攻击。

当一个cookie以这种方式标记时，支持它的浏览器将阻止客户端JavaScript直接访问cookie。虽然浏览器仍然会在请求的HTTP消息头中提交这个cookie，但它不会出现在document.cookie返回的字符串中。

# 失效的身份认证和会话管理

* 1. 漏洞描述

应用程序的功能一般包含权限管理和会话管理，由于应用程序设计不当，攻击者可以窃取到密码、密钥、session tokens等信息，冒充合法用户身份，获取敏感信息或者进行恶意操作。

* 1. 漏洞危害

利用不安全的权限管理和会话管理设计，恶意用户可能会窃取或操纵用户会话和 cookie，模仿合法用户，可能造成以下危害：

a） 窃取用户凭证和会话信息；

b） 冒充用户身份查看或者变更记录，甚至执行事务；

c） 访问未授权的页面和资源；

d） 执行超越权限操作。

* 1. 产生原理描述

a） 失效的身份认证

由于应用程序权限管理模块设置不完善，在用户登入登出、密码管理、用户登录信息本地存储等方面，存在认证被绕过或者身份认证信息被截获破解的弱点，比如某应用程序在用户遗忘密码，提供重置功能并将该密码发送至用户注册邮箱。由于缺乏对用户 ID、用户名和邮箱一致性的校验，或者仅在客户端做校验，导致恶意用户可能通过工具修改发送至的服务器的 ID、用户名和邮箱信息，比如提交其他用户的 ID、用户名、使用恶意用户自己的邮箱，这样就可能收到系统密码重置信息，恶意用户也就同时获得了合法用户的登录凭证。

b） 失效的会话管理

对于会话管理系统有两种类型。第一种类型是“宽容”系统，可让 Web 浏览器指定任何标识。第二种类型是“严格”系统，只接受服务器端生成的值。当使用宽容系统时，不需要连接 Web 站点，便可以维护任何会话标识。在严格系统中，攻击者需要维护“陷阱会话”并且应定期连接 Web 站点，才能防止闲置超时。

对于会话固定，如果没有活动保护，使用会话来识别已认证的用户的任何 Web 站点都可能受到攻击。使用会话标识的 Web 站点通常都是基于 cookie 的站点，但也会使用 URL 和隐藏的表单字段，基于 cookie 的会话最容易受到攻击。相对于在用户登录 Web 站点之后，再窃取用户的会话标识，会话固定被攻击的概率较高。在用户登录之前，攻击的活动部分便已启动。

会话固定攻击过程通常由三个步骤组成：

1） 安装会话：攻击者针对目标 Web 站点设下“陷阱会话“，并获取会话的标识，攻击者也可以选择攻击中所用的任意会话标识。在某些情况下，应反复联系 Web 站点来维护确定好的陷阱会话值。

2） 固定会话：攻击者将陷阱会话值引进用户的浏览器中，固定用户的会话标识。

3） 进入会话：用户登录目标 Web 站点之后，当使用固定会话标识值时，攻击者便可加以接管。 如果会话管理系统接受 URL 参数形式的会话标识，下列请求便可以强迫会话标识采用 URL 参数值。

代码片段(正常)：

|  |
| --- |
| http://example/login.php?PHPSESSID=1234 |

代码片段(加入攻击代码)：

|  |
| --- |
| http://example/<script>document.cookie="sessionid=1234;%20domain=.example.dom";</script> |

另一个类似的示例（使用 META 标记注入）：

|  |
| --- |
| http://example/<meta%20http-equiv=Set-Cookie%20content=”sessionid=1234;%20domain=.example.dom”> |

利用 HTTP 响应头发出 cookie，攻击者强迫目标 Web 站点或域中的任何其他站点发出会话标识 cookie。许多方法都可以做到这一点：

——进入域中的某 Web 服务器（例如：维护不良的 WAP 服务器）；

——将攻击者的 Web 服务器添加到域中 ；

——利用 HTTP 响应分割攻击。

在登录过程前后对会话标识的比较，如果并未更新，表示有可能伪装用户。初步得知会话标识值后，远程攻击者有可能充当已登录的合法用户。利用“跨站点脚本编制”漏洞可以获取会话标识值，导致受害者的浏览器在联系易受攻击的站点时使用预定的会话标识；启用“固定会话”也可以获取会话标识值，导致站点在受害者的浏览器中显示预定的会话标识。

* 1. 解决方案

a） 始终生成新的会话。

用户登录成功生成新ID：

|  |
| --- |
| request.getSession().invalidate();//清空session  Cookie cookie = request.getCookies()[0];//获取cookie  cookie.setMaxAge(0);//让cookie过期 |

用户再输入信息登录时，就会产生一个新的session了。

b） 防止用户操纵会话标识。

具体措施如下：

1） 用户密码强度(普通-6位以上；重要-8位以上；极其重要：使用多种验证方式)。

2） 不应使用简单或可预期的密码恢复问题。

3） 应经登录页面进行加密处理；登录出错时不应提供太多的提示，应使用统一的出错提示；登录验证成功后更换Session ID。

4） 第一次登录强制时修改密码；对多次登录失败的帐号进行短时锁定；设置会话闲置超时。

5） 提供用户注销退出功能，用户关闭浏览器或者注销时，删除用户Session。

6） 使用128位以上具备随机性的SessionID，不应在URL中显示Session ID。

c） 保护Cookie

在应用程序中为Cookie设置安全属性：Secure flag和HttpOnly flag。

# 不安全的直接对象引用

* 1. 漏洞描述

直接对象引用，是指应用程序没有检查用户是否具备访问权限直接调用系统的资源，比如文件、数据库记录、或者是修改URL表单的参数。

* 1. 漏洞危害

不安全的直接对象引用造成如下危害：

a） 用户无需授权访问其他用户的资料；

b） 用户无需授权访问支撑系统文件资料；

c） 用户无需授权访问权限外信息。

* 1. 产生原理描述

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 |

攻击者能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 |

攻击者有可能通过ViewLog功能来访问 Web 服务器上的任意文件。用“%2f”来编码“/”字符便可能做到这一点，它可以避开过滤，因而允许对任何目录的访问。

|  |
| --- |
| http://SERVER/admin-serv/tasks/configuration/ViewLog?file=  boot.ini&num=5000&str=&directories=  admin-serv%2Flogs%2f..%2f..%2f..%2f..%2f..%2f..%2f&id=admin-serv |

* 1. 解决方案

a） 使用自定义的映射名称

不应在 URL 或网页中直接引用内部文件名或数据库关键字，应使用自定义的映射名称来取代直接对象名，防止将重要的关键字和文件名泄露给用户。

b） 配置目录访问权限和检测用户访问权限

应根据最小权限原则，配置应用程序的访问权限，应禁止访问Web 目录之外的文件。

c） 检测用户访问权限

在访问对象时，应检测用户访问权限。如下面一段代码示例，在用户查询 CartID 时，同时验证用户USERID。

|  |
| --- |
| //检测用户访问权限  int cartID = Integer.parseInt( request.getParameter( "cartID" ) );  User user = (User)request.getSession().getAttribute( "user" );  String query = "SELECT \* FROM table WHERE  cartID=" + cartID + " AND userID=" + user.getID(); |

d） 验证用户输入

验证用户输入和URL请求，拒绝包含./或../的请求或者是转码的“.%2f..%2f”。（代码示例参考SQL注入，对用户输入进行过滤）。

|  |
| --- |
| public String PathCheck(String sText) {  for(;;)  {  if(sText.indexOf("./")>=0)  sText = sText.replaceAll(“./”, “”);  else  break;  }  for(;;)  {  if(sText.indexOf("..//")>=0)  sText = sText.replaceAll(“..//”, “”);  else  break;  }  } |

# 跨站请求伪造

* 1. 漏洞描述

跨站请求伪造（CSRF）是一种挟制终端用户在当前已登录的Web 应用程序上执行非本意操作的攻击方法。攻击者利用的社会工程诡计，例如通过电子邮件或者是聊天软件发送的链接，攻击者将迫使Web 应用程序的用户去执行攻击者选择的操作。

在跨站请求伪造攻击中，攻击者构造恶意URL请求，然后诱骗合法用户访问此URL链接，在Web应用中以此用户权限执行特定操作。与反射型XSS 的主要区别是：反射型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。

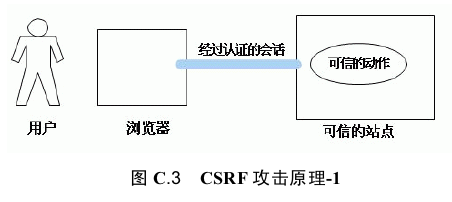
* 1. 漏洞危害

当CSRF针对普通用户发动攻击时，将对终端用户的数据和操作指令构成严重的威胁，当受到攻击的终端用户具有管理员帐户的时候，CSRF攻击将危及整个Web 应用程序。

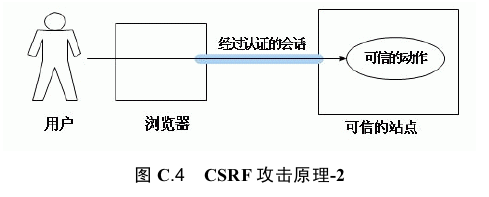
**示例：**如果用户登录网络银行去查看其存款余额，他没有退出网络银行系统就去了自己喜欢的论坛去灌水，如果攻击者在论坛中精心构造了一个恶意的链接并诱使该用户点击了该链接，那么该用户在网络银行帐户中的资金就有可能被转移到攻击者指定的帐户中。

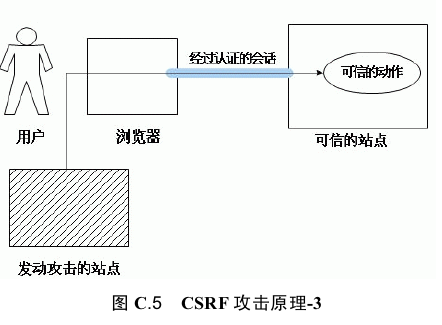
* 1. 产生原理描述

图C.3、图C.4和图C.5介绍了CSRF攻击的一般原理。



Web 浏览器已经跟可信的站点建立了经过认证的会话。之后只要是通过该 Web 浏览器这个认证的会话所发送的请求，都被视为可信的动作。





图C.5.发生了一个CSRF攻击。发起攻击的站点致使浏览器向可信的站点发送一个请求，该可信的站点认为，来自该Web 浏览器的请求都是经过认证的有效请求，所以执行此“可信的动作”。

* 1. 解决方案

应采用如下手段防范跨站脚本攻击：

a） 限制身份认证cookie的到期时间

身份认证 cookie 的合法时间越短，攻击者利用 Web 应用程序的机会就越小。时间越短，用户操作不方便。因此，需要在安全性和方便性之间进行平衡。

如下代码中，具体功能代码前加入一段检查用户登录超时时间的代码，通过检查用户最后一次操作时间与当前时间的差值是否小于系统设置值（本例中为600秒）来判断：

|  |
| --- |
| //获取当前系统时间  long currentTime =new Date().getTime()/1000;  Long lastTime =  ((Date)request.getSession().getAttribute("LAST\_TIME")).getTime()/1000;  If(currentTime - lastTime > 600)  {  session.invalidate();// 销毁会话  response.getWriter().println(  "<script language=\"JavaScript\" type=\"text/JavaScript\">alert(\""+ 由于您长时间没有操  作，您的登录已超时，请重新登录！  + "\");window.close();</script>"  ) ;  }  Else  {  ……  //实际功能处理代码  ……  //更新session中记录的当前用户最后操作时间  } |

b） 要求用户提交额外的信息

执行重要业务之前，要求用户提交额外的信息。例如要求用户在进行重要业务前输入口令或图形验证码，这可以防止攻击者发动CSRF攻击(只要浏览器中没有包含口令或验证码信息)。

如下代码中，具体功能代码前加入一段检查用户是否输入了正确的图性验证码的代码，只有图形验证码检查通过才开始执行具体的功能代码：

|  |
| --- |
| // 比较验证码  String input\_code = request  .getParameter("validate\_code");// 用户输入的验证码  String right\_code = (String) request.getSession()  .getAttribute("validate\_code");// 存在于Session中的正确验证码  if (! input\_code.trim().equalsIgnoreCase(right\_code)) {  request.setAttribute("MSG", “验证码有误”);  return mapping.findForward("fail");  else  {  ……  //实际功能处理代码  ……  } |

c） 使用一次性令牌

使用一次性令牌，Get 请求中在 URL 里面加入一个令牌，对于 Post 请求，在隐藏域中加入一个令牌。这个令牌由 server 端生成，由编程人员控制客户端发送请求时使请求携带本令牌，然后在 Server 端进行验证。在令牌的设计上应避免采用如下错误的方案：

1） 使用和 Session 独立的令牌生成方式。

2） 完全使用 Session 认证信息作为令牌的生成方式。

正确的令牌设计应该是使用 Session 信息做 Hash，用得出的哈希值来做 CSRF 的令牌。

如下代码：

1） 生成令牌代码函数

|  |
| --- |
| Public String getTokenCode(HttpServletRequest request){  String tokenCode =request.getSession.getId+Math.abs(new SecureRandom().nextLong());  Return tokenCode;  } |

2） 输出页面链接时，POST域中包含此一次性令牌（a\_dynamic\_value）

|  |
| --- |
| <form method=“transfer\_money”>  <input type=“text” name=“from” />  <input type=“text” name=“to” />  <input type=“text” name=“amount” />  <input type=“hidden” name=“secret”  value=“<%= a\_dynamic\_value %>” />  </form> |

3） 后台处理时，检查传入的一次性令牌是否合法，只有包含合法的令牌数据才执行后续的功能代码

|  |
| --- |
| //获取令牌数据  String input\_token =request.getParameter(“secret”);  String right\_token = getTokenCode(request);  //检查令牌是否合法  if (! input\_token.equalsIgnoreCase(right\_token)) {  {  request.setAttribute("MSG", “令牌有误”);  return mapping.findForward("fail");  }  Else  {  ……  //实际功能处理代码  ……  } |

d） 检查 HTTP 头部 Referer 信息

检查 HTTP 头部 Referer 信息，这是防止 CSRF 的最简单容易实现的一种手段。根据 RFC 对HTTP 协议里面 Referer 的定义，Referer 信息跟随出现在每个 Http 请求头部。Server 端在收到请求之后，可以去检查这个头信息，只接受来自本域的请求而忽略外部域的请求。该方法仅能作为补充，因为检查方式过于简单也有它自身的弱点：

1） 首先是检查 Referer 信息并不能防范来自本域的攻击。

2） 攻击者可以伪造 Referer 信息。

如下代码中，具体功能代码前加入一段检查当前访问者的HTTP Referer信息代码，只有Referer信

息的主机匹配才继续执行功能代码：

|  |
| --- |
| //获取当前URL中的主机名  String url\_host =request.getParameter(“host”);  //获取当前访问者的HTTP Refer信息  //获取HTTP Refer信息中的主机名  String http\_host =request.getRemoteAddr();  If( ! url\_host .equals(http\_host))  {  request.setAttribute("MSG", “对不起，您的地址不在访问权限内 !”);  return mapping.findForward("fail");  }  Else  {  ……  //实际功能处理代码  ……  } |

Web 开发人员可以根据自己对应用程序的功能的理解来确定安全级别，从而选择使用不同的保护措施，应在同一应用程序内部结合使用多种方法来进行保护。

e） 避免在URL中明文显示特定操作的参数内容

在URL中明文显示特定操作的参数内容，当该Web 应用系统存在跨站脚本欺骗漏洞时，能使攻击者很容易地使用该漏洞对Web应用系统进行攻击。例如，银行转账操作的URL http://www.yinhang.com? &account=number&transfer=X时，就会很容易使攻击者利用该漏洞对账户进行转账操作。所以，应避免在URL中明文显示特定操作的参数内容。

# 安全配置错误

* 1. 漏洞描述

由于操作系统、应用服务器、数据库服务器、应用程序、中间件以及相关应用程序所使用的框架的不安全配置，造成恶意用户能够利用系统默认账户、默认配置页面（操作系统、Web 服务器、数据库服务器、中间件漏洞），对应用系统进行攻击，如：窃取系统敏感信息、尝试控制服务器。

* 1. 漏洞危害

安全配置错误，可能造成以下危害：

a） 恶意用户获取系统敏感信息，比如程序路径，配置信息等。

b） 恶意用户通过默认账户登录系统，获取系统机密信息，甚至操纵整个服务器

* 1. 产生原理描述

主要原因：

a） 服务器没有及时安装补丁；

b） 网站没有禁止目录浏览功能；

c） 网站允许匿名用户直接上传文件；

d） 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；

e） Web 服务器安装并运行不需要的服务，比 FTP或SMTP等；

f） 出错页面向用户提供过多具体的错误信息，比如call stack；

g） 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；

h） SQL服务器没有限制系统存储过程的使用，比如xp\_cmdshell。

当系统因为逻辑或资料错误，会将程序中的错误信息暴露在网页上。例如：错误的 SQL 语句，使得恶意攻击者可以利用不同错误信息进行资料收集。

* 1. 解决方案

应采用如下手段防范安全配置错误：

a） 自定义出错页面，统一错误页面提示；

b） 安装最新版本的软件及最小化安装(只安装需要的组件)；

c） 避免使用默认路径，修改默认帐号和密码，禁用预设帐号；

d） 主机、数据库、WEB服务器、中间件应保持自动更新，并进行加固；

e） 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；

f） 严格检查所有与验证和权限有关的设定；

g） 检查文件目录访问权限是否符合最小化原则；

h） 检查所有服务器是否都受到了防火墙和过滤器的保护。

# 不安全的加密存储

* 1. 漏洞描述

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，将会导致攻击者获得重要信息。

* 1. 漏洞危害

机密资料容易被攻击者破解，造成资料外泄。

* 1. 产生原理描述

在编码过程中存在如下行为时，均会产生此问题：

a） 对于重要信息，比如银行卡号、密码等，直接以明文写入数据库或配置文件；

b） 使用程序员编写的加密算法进行简单加密；

c） 使用MD5，SHA-1等低强度的算法；

d） 将加密信息和密钥存放在一起。

* 1. 解决方案

在系统设计阶段时，应采用AES、RSA public key cryptography和SHA-256等较安全的算法。

应采取如下手段防止“不安全的加密存储”漏洞：

a） 对所有重要信息进行加密；

b） 使用足够强度的加密算法，比如AES、RSA；

c） 存储密码时，用SHA-256等健壮哈希算法进行处理；

d） 产生的密钥不应与加密信息一起存放；

e） 严格控制对加密存储的访问等。

如果使用MD5算法，应采用两遍MD5的方式，对原始数据进行保护，如下示例：

|  |
| --- |
| 第一遍MD5： strTmp = md5(strSuorce);  中间结果字串变形：strTmp = strTmp+”XXXX”;  第二遍MD5： strResult = md5(strTmp); |

# 越权访问

* 1. 漏洞描述

Web 应用包含一些“隐藏”的 URL，不显示在网页链接中，但用户能通过直接在浏览器中输入URL访问到 “隐藏”页面。

* 1. 漏洞危害

将导致非授权用户进行系统操作，能造成系统数据泄漏或数据被篡改。

* 1. 产生原理描述

WEB应用访问控制机制有缺陷，部分的页面缺乏用户权限检查机制，如下例子：

一个 WEB 应用存在用户管理功能，允许管理员添加、删除、修改系统用户，但在用户管理页面中没有对当前访问者的权限检查，攻击者能进行一些简单尝试找到用户管理功能的访问方式：

http://example.com/admin/usrmgt

通过直接访问用户管理功能URL获取了添加、删除、修改系统用户的权限。

* 1. 解决方案

防止未经授权的 URL 访问，应选择适当的身份验证机制，并为每个功能页面加入授权检测的方法：

a） 检测机制默认情况下拒绝所有的访问；

b） 针对每个功能页面明确授予特定的用户和角色允许访问。

如下代码示例：

|  |
| --- |
| //设置检查结果标志为false  boolean check\_result =false;// 默认情况下拒绝所有的访问  //检查当前用户是否登录，为否的提示用户需要登录  If(request.getSession(false)==false){  request.setAttribute("MSG", “对不起，您还没有登录，请登录!”);  return mapping.findForward("fail");    }  //检查用户的权限是否足够，足够时置标志为true  If( checkRights(userId) ==true){  check\_result =true;  //当标志为true时，执行功能代码  } |

# 传输层保护不足

* 1. 漏洞描述

WEB 应用程序在进行数据传输时，可能会选择 HTTP 或者 HTTPS。HTTP 在传输的过程中，未对传输数据做保护，传输数据有可能被恶意用户截获、篡改。对于未使用 SSL/TSL 保护的传输通道，称之为传输层层保护不足。

* 1. 漏洞危害

传输层保护不足，可能会造成如下危害：

a） 导致用户凭证和用户会话信息被窃取；

b） 导致敏感信息泄露、篡改。

* 1. 产生原理描述

如果某网站的登录页面没有进行加密，攻击者能利用工具截取网络数据包来获得用户的登录凭据信息。

* 1. 解决方案

针对传输层保护不足，应采用如下解决办法：

a） 对所有验证页面都使用SSL或TLS加密；

b） 对所有敏感信息的传输都使用SSL/TLS加密；

c） 在网页中不要混杂HTTP和HTTPS内容；

d） 对Cookie使用Secure标签；

e） 保持服务器证书的有效性/合法性；

f） 只允许SSL 3.0或TLS 1.0以上版本协议；

# 未验证的重定向和转发

* 1. 漏洞描述

未验证的重定向和转发，是指系统在进行页面转发和重定向时重定向或者转发的页面不是系统可信页面，并且重定向或转发前未经安全验证。

* 1. 漏洞危害

攻击者可以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。

* 1. 产生原理描述

应用程序在重定向或转发时，重定向或转发的目标 URL 包含输入的数据，如果应用程序对输入数据缺乏安全检查，攻击者能在输入数据中构造URL信息，从而诱骗用户访问恶意站点，如下示例：

系统的登录功能页面login.jsp接受一个backurl参数，当登录成功时会自动重定向当backurl指定的URL。

|  |
| --- |
| String backurl = request.getParameter("backurl");  ……  //登录成功时  response.sendredirect(backurl); |

攻击者可以构造如下链接：

|  |
| --- |
| http://example.com/login.jsp?backurl=http://www.example1.com |

当用户访问该链接时，用户登录成功后浏览器即会重定向到一个钓鱼网站(www.example1.com)

* 1. 解决方案

在编码时应尽量减少使用重定向或跳转，如果使用了重定向或跳转，应采取如下安全措施：

a） 不直接从输入中获取URL

应用程序应该尽量避免用户直接指定重定向或跳转的 URL，输入数据中不应直接包含表示重定向或跳转的URL，而以映射的代码表示URL，如：http://example.com/login.jsp?backurl=234

b） 检查重定向或跳转的目标URL是否为本系统URL

检查重定向或跳转的目标URL是否为本系统URL，如下代码：

|  |
| --- |
| String backurl = request.getParameter("backurl");  获取目标中的域名  If(目标中的域名 ==”www.example1.com”)  {  response.sendredirect(backurl);  } |

c） 对输入进行过滤

当重定向或跳转的内容中包含输入数据时，应对输入数据做严格检查，检查/过滤输入数据中的特殊字符：\r\n和\n\n。

|  |
| --- |
| public static String filter(String value) {  if (value == null) {  return null;  }  StringBuffer result = new StringBuffer(value.length());  for (int i=0; i<value.length(); ++i) {  switch (value.charAt(i)) {  case ‘\r':  break;  case '\n':  break;  default:  result.append(value.charAt(i));  break;  }  return result;  }  ... |

# 上传任意文件

* 1. 漏洞描述

上传任意文件漏洞是指用户可以上传任意文件，程序没有检测上传文件大小、类型是否是符合预期要求，或仅仅在客户端对上传文件大小、上传文件类型进行限制，未在服务器端进行验证。

* 1. 漏洞危害

如果上传文件脚本未对用户所上传文件进行任何限制，或者上传文件检测能够被绕过，则能会对服务器造成危害。恶意用户能够通过上传木马文件，获取目标服务器的控制权限。

* 1. 产生原理描述

由于程序员对用户文件上传部分的控制不足或者处理存在缺陷，而导致用户能越过其本身权限向服务器上上传可执行的动态脚本文件。

如下面代码所示：

FormFile theFile = advertiseform.getFilepath();

String up\_path = servlet.getServletContext().getRealPath("/");

if (theFile != null) {

String newfilename = this.FileWrite(theFile,up\_path);.

//代码中未对FileWrite中没有对上传文件的类型限制(后缀)；

* 1. 解决方案

针对上传任意文件漏洞，应采取如下措施进行防范：

a） 对上传文件类型进行检查

仅允许上传特定的文件类型，检查不能依据HTTP HEAD信息，应直接检查上传文件的扩展名，如下示例：

|  |
| --- |
| //代码示例：限制上传文件为XLS  String xlsPath = props.getProperty("xls");  bis = new BufferedInputStream(file.getInputStream());  File upload = new File(xlsPath + "/" + fileName);  if ("".equals(fileName)) {  ActionMessages messages = new ActionMessages();  messages.add("file", new ActionMessage("文件名不能为空!",false));  saveErrors(request, messages);  return mapping.findForward("fetchxls");  }  String[] fArray = fileName.split("\\.");  if (fArray.length == 1) {  ActionMessages messages = new ActionMessages();  messages.add("file", new ActionMessage("文件格式无效!",false));  saveErrors(request, messages);  return mapping.findForward("fetchxls");  }  if (!"xls".equalsIgnoreCase(fArray[fArray.count()-1])) {  ActionMessages messages = new ActionMessages();  messages.add("file", new ActionMessage("文件格式无效!",false));  saveErrors(request, messages);  return mapping.findForward("fetchxls");  }  if (file.getFileSize() == 0) {  ActionMessages messages = new ActionMessages();  messages.add("file", new ActionMessage("文件长度为0，上传失败!",false));  saveErrors(request, messages);  return mapping.findForward("fetchxls");  }  bos = new BufferedOutputStream(new FileOutputStream(upload));  int available = bis.available(); |

b） 限制上传文件的访问权限

1） 在服务器利用系统权限限制上传文件和目录的权限，拒绝执行权限；

2） 将上传目录从WEB空间移出，保证用户无法直接从URL访问到上传文件。

# 命令注入

* 1. 漏洞描述

命令注入是指系统在没有指定完整路径的情况下执行命令可能使得攻击者可以通过改变＄PATH 或者其他环境变量来执行恶意代码。

* 1. 漏洞危害

攻击者能向存在命令注入的系统提交经过构造的输入，输入中包含攻击者试图运行的系统命令，当构造的系统命令运行成功时攻击者将非法获得数据或者网络资源，甚至可以控制系统服务器。

* 1. 产生原理描述

命令注入攻击有两种形式：

a） 攻击者可以改变应用程序执行的命令。

b） 攻击者可以改变命令执行的环境。

针对第二种情况，即攻击者可以通过改变环境变量或者在搜索路径中插入可执行恶意代码来改变原有命令目的。此类型的命令注入攻击通常发生在：

1） 攻击者改变应用程序的环境；

2） 应用程序执行命令时没有指定完整的路径或者校验执行的代码；

3） 通过执行该命令，应用程序会给予攻击者不应该具有的特殊的权限或者能力。

下面代码使用系统变量 APPHOME 来定位程序安装的目录，然后根据此目录的相关路径来执行初始化脚本。

|  |
| --- |
| …  String val = Environment.GetEnvironmen Variable (“APPHOME”)  String cmd = val + INITCMD;  ProcessStarInfo startInfo = new ProcessStartInfo(cmd);  Process.Start(startInfo); |

示例中的代码中，攻击者可以通过修改系统变量 APPHOME，使其指向一个包含恶意代码的路径，进而利用程序的高级权限来执行任意命令。因为程序没有对读入的环境变量值进行验证，攻击者能欺骗应用程序，让其执行恶意代码，从而控制系统。

下面的代码来自于一个 web 管理程序，它允许用户使用批处理文件包对 Oracle 数据库的备份，然后运行 cleanup.bat 脚本来删除临时文件。脚本 rmanDB.bat 接收一条命令行参数来决定进行何种备份。因为对数据库的访问是受限的，所以应用程序会以特殊权限来执行备份。

|  |
| --- |
| …  String btype = requwst.getParameter(“backuptype”);  String cmd = new String(“cmd.exe/K\”c:\\rmanDB.bat “  + btype + “&&c:\\utl\\cleanup.bat\””)  System.Runtime.getRuntime().exec(cmd)  … |

程序没有对从用户处读取的 backuptype 参数进行任何验证。通常 Runtime.exec()函数不会执行多个命令，但在程序为了在只调用一次Runtime.exec()的情况下运行多个命令，先运行了cmd.exe，将会执行由&&分隔开的多个命令；如果攻击者传入一个”&& dek c:\\dbms\\\*.\* ”形式的字符串，那么应用程序会将该命令与程序指定的其它命令一起执行。因为应用程序需要有足够的权限来与数据库交互，所以攻击者注入的任何命令都将以同样的权限执行。

下面的代码来自一个 web 应用程序，它允许用户通过访问接口来修改他们的系统密码。在网络环境中，修改密码的步骤之一是在/var/yp目录运行make命令。

|  |
| --- |
| …  System. Runtime.getRuntime().exec(“make”)  … |

程序没有指定一个完整的路径，并且在执行 Runtime.exec()调用前没有清理环境。如果攻击者能够修改环境变量$PATH，让其指向一个叫做 make 的恶意代码，并使程序在该环境中执行，那么程序装载的二进制代码将会被恶意代码替换。因为程序需要有足够的权限来执行系统操作，所以攻击者的 make将能够以同样的权限运行，可能使攻击者能够完全控制整个系统。

* 1. 解决方案

命令注入攻击主要是因为用户输入或者环境变量值没有进行验证所造成的。为了防范命令注入攻击，应该对用户输入进行校验，过滤非法字符（如“&&”等）。同时，在程序中应该尽量不要从环境中读入数据，即使不得不依赖环境变量，也应该严格限制这些变量的路径和所起的作用，减少程序受到的影响。

a） 采用白名单验证输入

在服务器端采用正则表达式，对用户输入进行校验。如下代码：

|  |
| --- |
| 检查函数的示例代码如下：  public static boolean Check(String str){  int length=str.length();  String ss="[a-zA-Z\_0-9]";  for(int i=0;i<length;i++){  String s=str.substring(i,i+1);  if(!Pattern.matches(ss,s)){  return false;  }  }  return true;  } |

在上述代码中，通过设定白名单规则检查输入只能包含大小写字母、数字。

b） 过滤输入数据中的特殊字符

在服务器端过滤输入数据中的特殊字符，针对用户输入应该过滤的特殊字符为，如下表C.6.：

|  |
| --- |
| 特殊/关键字符(不区分大小写)  &&  ; |

过滤的示例代码如下：

|  |
| --- |
| public String CommandEncode(String sText) {  sText = sText.replaceAll(“&”, “”);  sText = sText.replaceAll(“;”, “”);  } |

c） 去除应用程序运行时的不必要权限

应去除应用程序运行时的不必要权限。很多攻击行为只有在获得高级特权（比如系统管理员 root权限）后才可以生效。

# 信息泄露和不恰当的错误处理

* 1. 漏洞描述

程序在错误的WEB页面中存在一些可能提供给攻击者攻击线索的调试信息。

* 1. 漏洞危害

可能会收集有关 Web 应用程序的敏感信息，如用户名、密码、机器名或敏感文件位置。

* 1. 产生原理描述

程序员在 Web 应用程序中使用 HTML 注释帮助调试应用程序，可能在注释中会遗留重要数据（例如：与 Web 应用程序相关的文件名、旧的链接或原非供用户浏览的链接、旧的代码片段等），造成信息泄露。

不恰当的错误处理，是指应用程序在处理用户输入错误时，抛出的错误信息，给恶意用户提供了判断潜在漏洞的条件。比如说用户名和密码输入错误分别提示“用户名错误”，“密码错误”，这就给恶意用户提供了一个条件用来固定用户名。登录界面

* 1. 解决方案

采用如下方法，能防止信息泄露和不确当的错误处理漏洞：

a） 除了必要的注释之外，将所有的调试语句删除。

b） 对返回客户端的提示信息进行统一和格式化。

如下示例为用户登录失败时的错误信息提示，当用户名错误或密码错误时，系统均返回“用户名或者密码错误”。

|  |
| --- |
| 正确的用户名或者密码错误错误处理示例代码如下：  try {  result = dispatcher.runSync("userLogin", UtilMisc.toMap("login.username",  username, "login.password", password, "visitId", visitId));  Enumeration c=session.getAttributeNames();  while(c.hasMoreElements()){  session.removeAttribute( (String)c.nextElement() );  }  } catch (GenericServiceException e) {  Debug.logError(e, "Error calling userLogin service", module);  request.setAttribute("\_ERROR\_MESSAGE\_", "<b>用户名或者密码错误</b><br>"  + e.getMessage());  return "error";  }    ……    } else {  String errMsg = (String) result.get(ModelService.ERROR\_MESSAGE);  if (errMsg.equals("User not found."))  errMsg="用户名或者密码错误";  errMsg = "<b>"+errMsg+"</b><br>" ;  request.setAttribute("\_ERROR\_MESSAGE\_", errMsg);  return "error";  } |

c） 制作统一的出错提示页面。

为应用系统定制统一的错误提示页面，只显示错误编号，不显示任何系统信息。当出错或发生异常时，均将页面定向到该错误提示页面。

# 目录浏览

* 1. 漏洞描述

目录浏览是指允许用户通过修改输入来操作文件系统的路径。

* 1. 漏洞危害

目录浏览漏洞会导致攻击者非法获取系统信息，包括配置文件信息、机密文件信息（如UNIX服务器的/etc/passwd 文件，Windows 服务器的 SAM 文件）等，甚至可以得到数据库用户名和密码，进而通过数据库获得整个系统的权限。

* 1. 产生原理描述

如下两段代码描述了目录浏览漏洞产生的原因：

下面的代码del.jsp从HTTPreques获取用户输入做为文件名来删除一个系统文件：

|  |
| --- |
| String rName = request.getParameter(“reportName”);  File rFile = new File(“/usr/local/apfr/reports/” + rName );  …  rFile.delete() |

开发人员在编码没有对输入做安全验证或检查，攻击者可以提供像“../tomcat/conf/server.xml”的文件名，从而导致应用程序删除了自己的配置文件，如下URL：

http://example.com/del.jsp?reportName=../tomcat/conf/server.xml

* 1. 解决方案

a） 不直接从输入中获取文件路径

应用程序应该尽量避免用户直接指定文件系统操作路径，输入数据中不应直接包含文件路径，而应以映射的代码表示文件路径，如：http://example.com/down.jsp?fileid=234

b） 采用白名单验证文件路径名

在服务器端采用正则表达式，对用户指定的路径进行输入校验。如下代码：

|  |
| --- |
| 检查函数的示例代码如下：  public static boolean fileCheck(String str){  int length=str.length();  String ss="[a-zA-Z\_0-9]|[.]|[\_]";  for(int i=0;i<length;i++){  String s=str.substring(i,i+1);  if(!Pattern.matches(ss,s)){  return false;  }  }  return true;  } |

在上述代码中，通过设定白名单规则检查文件名只能包含大小写字母、数字、下划线和句点。

c） 过滤文件路径名中的特殊字符

在服务器端过滤路径名中的特殊字符：../ 和..\，如下代码：

|  |
| --- |
| public String PathCheck(String sText) {  for(;;)  {  if(sText.indexOf("..\\")>=0)  sText = sText.replaceAll(“..\\”, “”);  else  break;  }  for(;;)  {  if(sText.indexOf("..//")>=0)  sText = sText.replaceAll(“..//”, “”);  else  break;  }  } |

由于攻击者可能采用在路径名中包含….////字符的形式绕过 sText.replaceAll(“../”, “”)，因此应采用循环处理进行过滤。

d） 去除应用程序不必要的权限

应去除应用程序运行时的不必要权限。一些攻击行为只有在获得高级特权（比如系统管理员 root权限）后才可以生效。

# 整数溢出

* 1. 漏洞描述

在不同的CPU系统上被编译处理后，整型和指针的尺寸一般是相同的。例如，在32位的系统中，比如x86系统中，一个整数是32位；而在64位的系统中，比如SPARC，一个整数是64位长。本文中以32位的系统环境和32位的整数为例，并且用10进制来表示它。有符号数通过变量的最高位来表示，如果最高位置 1，这个变量就被解释为负数；如果置 0，这个变量就解释为正整数。因此通常的高级语言中都有有符号整型（int）和无符号整型（unsigned int）之分。

* 1. 漏洞危害

如果一个整型溢出发生了，应用程序并不知道它的计算是错误的，因此应用程序在假定它是正确的情况下继续运行下去。正是由于其隐蔽性，整型溢出也是很难被发现的，通常的压力测试往往无法成功的引发整型溢出，对于这种漏洞的发掘，一般需要使用逆行工程对程序代码进行详细的审核。整型溢出并不能直接导致改变程序流程，它是由整型溢出造成字符串类型的溢出，从而导致覆盖系统数据结构，改变程序流程。

* 1. 产生原理描述

整数溢出根据溢出原因分为三大类：存储溢出、计算溢出和符号问题.

a） 存储溢出

存储溢出是使用不同的数据类型来存储整数造成的。例如下面程序所示：

|  |
| --- |
| int len1 = 0x10000;  short len2 = len1; |

由于len1和len2的数据类型长度不一样，len1是32位，而len2是16位，因此进行赋值操作后， len2无法容纳len1的全部位，导致了与预期不一致的结果，即len2等于0。

把短类型变量赋给长类型变量同样存在问题，例如如下代码：

|  |
| --- |
| short len2 = 1;  int len1 = len2; |

上面代码的执行结果并非如预期使 len1 等于 1，在很多编译器编译的程序中结果是使 len1 等于0xffff0001，实际上就是一个负数。这是因为当len1的初始值等于0xffffffff，而把short类型的len2赋值给len1时只能覆盖掉其低16位，这就造成了安全隐患。

b） 运算溢出

对整数变量进行运算过程中没有考虑到其边界范围，造成运算后的数值超出了其存储空间，例如下面伪代码：

|  |
| --- |
| bool func(char \*userdata, short datalength)  {  char \*buff;  ...  if(datalength != strlen(userdata))  return false;  datalength = datalength\*2;  buff = malloc(datalength);  strncpy(buff, userdata, datalength)  ...  } |

其中 userdata 是用户提交的字符串，datalength 是用户提交的字符串长度。func()函数首先保证用户提交的字符串长度和字符串的实际长度一样，然后分配一块2倍于用户提交字符串大小的缓冲区，把用户字符串数据拷贝到这个缓冲区当中。程序仅仅考虑了对字符串数据的安全要求，却没有考虑长度变量的数据类型的表示范围。

假如用户提交的datalength＝0x8000，正常情况下0x8000×2＝0x10000，但是当把0x10000赋值给short 类型变量时，最高位的 1 会因为溢出而无法表示，此时 0x8000×2＝0 了。如果用户提交的datalength>=0x8000，那么就会发生溢出。

c） 符号问题

整数是分为有符号整数和无符号整数的，由于符号的问题也可能引发安全方面的隐患。由符号引起的溢出的最典型的例子是eEye发现的Apache HTTP Server分块编码漏洞。下面分析一下这个漏洞的成因。

分块编码（chunked encoding）传输方式是HTTP 1.1协议中定义的Web 用户向服务器提交数据的一种方法，当服务器收到chunked编码方式的数据时会分配一个缓冲区存放数据，如果提交的数据大小未知，客户端会以一个协商好的分块大小向服务器提交数据。

Apache 服务器缺省也提供了对分块编码支持。Apache 使用了一个有符号变量储存分块长度，同时分配了一个固定大小的堆栈缓冲区来储存分块数据。出于安全考虑，在将分块数据拷贝到缓冲区之前，Apache 会对分块长度进行检查，如果分块长度大于缓冲区长度，Apache 将最多只拷贝缓冲区长度的数据，否则，将根据分块长度进行数据拷贝。然而在进行上述检查时，没有将分块长度转换为无符号型进行比较。因此，如果攻击者将分块长度设置成一个负值，就会绕过上述安全检查，Apache 会将一个超长(至少>0x80000000字节)的分块数据拷贝到缓冲区中，这会造成缓冲区溢出。

* 1. 解决方案

应在编程之前进行详细的预测，仔细考虑整数有关变量的大小范围等问题，在编程时将各种问题考虑到并且进行相应的处理。

a） 使用合适的数据类型

充分考虑业务需求和各种可能情况。充分保证各种数据的取值范围，从而使用合适的数据类型。

b） 不在不同范围的数据类型之间进行赋值

尽量不应在不同范围的数据类型之间进行赋值，比如：

|  |
| --- |
| Char ch;  Int i;  I=256;  ch = i; |

c） 整数进行运算时，避免越界

在对整数进行运算时，仔细考虑整数的范围大小，避免越界，如下代码：

|  |
| --- |
| public void sample(byte[] b, int off, int len) {  if (len < 0 || off < 0 || len + off > b.length) {  // do A  ……  } else {  // do B  ……  }  } |

在上述代码中，如果 len 和 off 的和值超过整数取值范围，将造成程序的逻辑判断错误，正确的写法应如下：

|  |
| --- |
| public void sample(byte[] b, int off, int len) {  if (len < 0 || off < 0 || len > b.length - off) {  // do A  ……  } else {  // do B  ……  }  } |

版本记录

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本编号/修改状态 | 拟制人/修改人 | 修改日期 | 变动内容 | 备注 |
| V1.0 | 欧阳一昭 | 2017-04-22 | 新建 |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

编制： 审核： 标准化： 批准：

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_