JAVA/JSP 代码安全

■ 文档编号	V1.0	■ 密级	商业机密
■ 版本编号		■ 日期	2009.3.23

■ **版权声明**

■ **版本变更记录**

时间	版本	说明	修改人
2009.3.23	V1.0		张杰

■ **适用性声明**

本模板用于撰写绿盟科技内外各种正式文件，包括技术手册、标书、白皮书、会议通知、公司制度等文档使用。

目录

一. 前言 1

二. 安全编码 2

2.1 架构安全 2
2.1.1 MVC 框架 2
2.1.2 会话安全 2
2.1.3 安全基础 3
2.2 授权、认证和加密 3
2.2.1 源码加密 3
2.2.2 数据加密 3
2.2.3 认证和授权 3
2.2.4 口令策略 4
2.2.5 访问策略 4
2.2.6 隐私保护 5
2.3 输入验证 5
2.3.1 集中式输入验证 5
2.3.2 服务器端输入验证 5
2.3.3 建议采用白名单放弃黑名单 5
2.3.4 验证输入长度、格式、字符合法性等 5
2.3.5 验证所有的输入 6
2.3.6 防范元字符攻击 6
2.4 数据库连接安全 7
2.4.1 使用参数化 SQL 语句 7
2.4.2 数据库中的数据也是不可信的 8
2.4.3 确保数据库资源能够被释放 8
2.5 文件安全 8
2.5.1 严格控制文件上传 8
2.5.2 严格分配文件访问权限策略 10
2.5.3 安全的临时文件 10
2.5.4 访问竞争条件 10
2.6 开发调试 11
2.6.1 日志 11
2.6.2 调试 11
2.6.3 错误处理 12

一. 前言

Java 是一种通用的面向对象编程语言，类似于 C++语言。它能生成与平台无关的已编译 代码，这些代码能被 JVM 执行，并适用于分布式应用环境、异构系统和各种网络环境。Java 语言还可以从各个层面———从 Java 语言结构到 JVM 运行环境、从类库到完整应用———为应用 及其底层系统提供安全性和完整性。以下是 Java 语言的一些固有特性，它们构成了 Java 平台 的安全性:

Java 语言指定了所有基本类型的长度和所有运算的执行顺序，因此，代码在不同 JVM 中执 行的顺序均与指定的顺序相同。

Java 语言为类型和过程的名称定义了名称空间管理，从而提供了对变量和方法的访问控 制功能。通过限制不可信代码对关键对象的访问，可以保护程序的安全。例如，使用 public （公共）、protected（保护）、private（私有）、package（包）等限定类型成员以限制访问。

Java 语言不允许定义或引用指针，也就是说程序员既不能创建指向内存的指针，也不能 创建指向内存的偏移量。在类文件中，对方法和实例变量的所有引用都是通过符号名称实现 的。禁止使用指针有助于防止计算机病毒等恶意程序和指针滥用，如使用从对象指针开始的 偏移指针直接访问私有方法或超出数组边界等。

Java 对象封装支持“按约定编程”，这样可以重用已通过测试的代码。

Java 语言是一种强类型语言。编译时，Java 编译器会对类型不匹配进行详尽的检查。这 种机制将确保运行时的数据类型变量与编译时的兼容并保持一致。

Java 语言允许将类或方法声明为 final（最终的）。任何被声明为 final 的类或方法都不 能被覆盖，从而有助于保护代码免受恶意攻击，如创建一个继承类，然后用它来替代原始类 并覆盖方法。

Java 的垃圾收集机制也有助于保护 Java 程序，因为它提供透明的内存分配和垃圾内存回 收机制，而不是通过手动干预的方式来释放内存，从而确保程序在运行过程中的完整性，并 防止因程序无意或错误释放内存而导致 JVM 崩溃。

通过这些特性，Java 提供了安全编程语言，使程序员们可以自由编写代码，并在本地执 行代码或通过网络发布。

二. 安全编码

2.1 架构安全

2.1.1 MVC 框架

在开发 JAVA 的 web 系统时，建议使用 struts 或 Spring 框架进行开发。两者都是基于 MVC 模式构建 J2EE 的常用框架。使用这些成熟的框架可以帮助我们开发出更干净、更可管理、并且更易于测试的代码，而且我们还可以方便的使用这些框架中所包含的安全特性来加强我们的系统。

2.1.2 会话安全

- Session 会话 ID 要足够长足够随机

  为了防止攻击者暴力猜解 session 会话 ID 标识，应至少包含 128 位安全随机数标识符。
- 认证用户完成后应开启一个全新的用户 ID

  为了防止 session 会话 ID 劫持，应当在每次认证完成后开启全新的用户 ID，保证会话安全。
- 限制会话闲置时间

  为了防止用户忘记注销，应当限制会话闲置时间，一般不要超过 30 分钟。
- 限制会话生命周期

  为了防止会话劫持，我们应当强制限制一个会话的生命存活周期，一般不要超过 5 个小时。生命周期结束后应当重新进行验证。
- 允许用户自行注销会话

  应当允许用户安全的结束自己的会话
- 会话结束后清除数据

  一个会话结束后，应该安全的从硬盘和内存中清除相关数据
- 防范 CSRF 攻击

  Cookies Hashing， HTTP 来路验证，验证码，一次性令牌。

2.1.3 安全基础

应当在完全不信任用户输入和客户端提交的数据的前提下开发应用程序。这其中可能包括：任何用户输入、GET 提交、POST 提交、cookie、javascript 等。

2.2 授权、认证和加密

2.2.1 源码加密

Java 虽然是会将代码编译成字节码的二进制形式，但是只要有一个反编译器，任何人都可以逆向分析出明文的代码。Java 的灵活性使得源代码很容易被窃取，但与此同时，它也使通过加密保护代码变得相对容易，我们唯一需要了解的就是 Java 的 ClassLoader 对象。当然，在加密过程中，有关 Java Cryptography Extension(JCE)的知识也是必不可少的。

2.2.2 数据加密

使用公开的高强度的可信任的算法进行加密。通常使用的加密算法 比较简便高效,密钥简短，加解密速度快，破译极其困难。如： MD5/SHA1，DSA，DESede/DES，Diffie-Hellman 的使用。

2.2.3 认证和授权

- DNS 并不可信

  由于 DNS 服务器是很容易受到欺骗攻击的，客户端 DNS 缓存中毒的情况下，可能会连接到虚假的服务器系统上，从而产出安全问题。
- 尽量使用 SSL

  应当尽量使用 SSL 进行通信，在安全通信传输的同时还可以通过校验证书来认证来源的可信性。
- 限制验证次数和频率

  合理的验证次数和频率的限制，可以大大增加暴力破解的攻击成本。
- 安全的报错信息

在返回的错误信息中不应该包含可以被恶意利用的信息，如提示用户名错误、密码错误等，这将大大增加暴力猜解的可能性。

- 随机验证码的使用

  安全强度的图片验证码等随机数的验证可以有效的提高暴力猜解的难度。

2.2.4 口令策略

- 强制要求和验证用户设置的口令强度

  如：验证口令长度、是否包含数字、字符、特殊符号等
- 允许用户更改自己的口令
- 减少使用口令过期策略
- 口令找回应通过其他途径通知
- 源码中不出现口令
- 口令加密存储
- 默认口令安全策略

2.2.5 访问策略

- 采用基于角色的访问控制

  是目前公认的解决大型企业的统一资源访问控制的有效方法。其显著的两大特征是：1.减小授权管理的复杂性，降低管理开销。2.灵活地支持企业的安全策略，并对企业的变化有很大的伸缩性
- 集中式访问控制

  建立一个集中的访问控制系统可以改进下面的问题：

  1. 将访问策略应用到所有用户交互过程中。
  2. 可以设定默认的安全策略
  3. 便于审计控制我们的系统
  4. 有利于长期维护
- 对关键操作进行二次认证

  为了防止会话劫持，需要对关键操作进行再次的认证。例如：修改口令等
- 细粒度的访问控制

2.2.6 隐私保护

避免隐私数据暴露出来，对隐私数据的访问权限要遵循最小范围原则。防止系统中的成员控制变量泄露到其他用户的会话中。

2.3 输入验证

输入验证是 web 应用程序安全防范的首道防线。如果您能不让恶意的数据进入您的程序，或者至少不在程序中处理它，您的程序在面对攻击时将更加健壮。这与防火墙保护计算机的原理很类似；它不能预防所有的攻击，但它可以让一个程序更加稳定。这个过程叫做检查、验证或者过滤您的输入。

2.3.1 集中式输入验证

- 所有输入采用一致的验证方式和策略
- 方便扩展和升级
- 防护过滤严密

2.3.2 服务器端输入验证

所有来自 web 浏览器的数据都能被修改为任意内容，因此必须在服务器端进行适当的输入验证，以避免验证被绕过。

2.3.3 建议采用白名单放弃黑名单

白名单包含允许内容的模式，黑名单包含不允许内容的模式。使用白名单法更容易，也推荐使用此方法。

2.3.4 验证输入长度、格式、字符合法性等

验证时应验证允许输入的最大和最小长度，数字的最大和最小值，允许输入的字符范围和数据类型等。

2.3.5 验证所有的输入

应该对用户的所有输入进行验证，避免有遗漏的未经验证的数据被输入。

2.3.6 防范元字符攻击

所有的类似脚本语言以及标记语言之类的强调易用性与交互性的技术都有一个共性：其可以接受可变的控制结构与数据。例如，SQL 查询：

select * from emp where name = 'Brian'

由"select", "from", "where", and "=" with the data "*", "emp", "name" 以及"Brian"这些关键字组合。攻击者通常会使用具有特殊含义的字符或字符串来利用这些漏洞。例如在SQL 中单引号是危险的字符。在命令shell 中，分号比较危险。通过对同一段元字符的多种编码方式以及对同一种语言的多种实现方式，这个问题更加危险。

- SQL注入

  使用正确的参数化SQL防止注入攻击。

```
Connection con = DriverManager.getConnection(url, "user", "pass");
int OrderID;
// PreparedStatement precompiles an SQL statement expecting only arguments,
preventing injection of SQL
keywords.
PreparedStatement stmt = con.prepareStatement( "SELECT * FROM Orders
WHERE orderid = ?" );
stmt.setString( 1, OrderID );
ResultSet rs = stmt.executeQuery();
```

- 跨站脚本

  对尖括号和双引号等进行编码处理

```
public static String HTMLEncode(String aTagFragment){
final StringBuffer result = new StringBuffer();
final StringCharacterIterator iterator = new
StringCharacterIterator(aTagFragment);
char character = iterator.current();
while (character != StringCharacterIterator.DONE ){
if (character = = '<') {
result.append("&lt;");
}
else if (character = = '>') {
result.append("&gt;");
}
else if (character = = '\"') {
result.append("&quot;");
```

```
}
else if (character = = '\") {
result.append("&#039;");
}
else if (character = = '\\') {
result.append("&#092;");
}
else if (character = = '&') {
result.append("&amp;");
}
else {
//the char is not a special one
//add it to the result as is
result.append(character);
}
character = iterator.next();
}
return result.toString();
}
```

- 路径操纵
  使用白名单限制正斜杠(/)、反斜杠(\)、小数点(.)的输入。
- 命令注入
  使用白名单检测过滤表单参数值数据内容。

- 日志欺骗

2.4 数据库连接安全

2.4.1 使用参数化 SQL 语句

正确的使用参数化的SQL 语句，就可以通过不允许数据指向改变的方法来防御几乎所有的SQL 注入攻击。参数化的SQL 语句通常是由SQL 字符构造的，但是来自客户的数据是需要与一些绑定参数组合在一起的。也就是说，开发者使用这些绑定参数来准确的向数据库指出哪些应该被当作数据那些应该被当作命令。当程序要执行该语句的时候，它就会告知数据库这些绑定参数的运行值，这样的操作避免了数据被认为是命令语句而被执行的错误。如下面的代码：

```
Connection con = DriverManager.getConnection(url, "user", "pass");
int OrderID;
// PreparedStatement precompiles an SQL statement expecting only arguments,
preventing injection of SQL
```

```
keywords.
PreparedStatement stmt = con.prepareStatement( "SELECT * FROM Orders
WHERE orderid = ?" );
stmt.setString( 1, OrderID );
ResultSet rs = stmt.executeQuery();
```

2.4.2 数据库中的数据也是不可信的

攻击者可能在完全攻陷数据库前利用一些方法向数据库写入一些恶意数据。如果不加以防范，就很可能真的导致数据库被攻击者完全控制。

2.4.3 确保数据库资源能够被释放

由于资源泄露可能导致系统出现很难捕捉到的错误，所以应当建立一个资源管理模块并且完全按照规则进行操作。千万不要依赖Java 和.NET 的垃圾回收器来回收资源。垃圾回收器在进行回收之前还要检测对象是否适合进行垃圾回收。除非虚拟机的内存已经很低，才会进行垃圾回收，这样无法保证即将被回收的对象是处于正常的状态。

2.5 文件安全

2.5.1 严格控制文件上传

文件上传向来是 web 应用系统最具威胁的安全风险点之一，因此应当严格控制。

- 文件类型验证

  检验上传文件的后缀名，设定允许上传文件类型白名单。

- 存储路径安全

  避免允许用户设定或选择存储路径和保存的文件名。严格检测提交数据中是否含有中断符。

- 限制文件大小

```
public ActionForward execute(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response) ...{
        // webapps/uploadtest/upload
        String dir = servlet.getServletContext().getRealPath("/upload");
        List<String> list = new ArrayList<String>();//注意文件类型最好全部用小写
        list.add("jpg");
```

```
            list.add("jpeg");
            list.add("gif");
            list.add("bmp");
            // 允许上传的文件类型列表可以写在配置文件中，通过 xml 的解析获得。

            if (!dir.endsWith("/"))
                dir = dir.concat("/");
            HtmlfileForm htmlfileForm = (HtmlfileForm) form;// TODO Auto-generated
            // method stub
            FormFile file = htmlfileForm.getFile();
            if (file == null) ...{
                return mapping.getInputForward();
            }
            String fname = file.getFileName();// 获取文件名
            int fsize = file.getFileSize();// 获取文件大小
            String ext = fname.substring(fname.lastIndexOf(".") + 1, fname.length());
            // 获取文件类型，即扩展名,通过 String 类的 substring 方法截取字符串，
lastIndexOf 获取某个字符串最后出现的索引。
            ext = ext.toLowerCase();// 全部转换成小写。
            if (!list.contains(ext)) ...{// 判断该类型是否为允许上传的文件类型
                System.out.println("不支持该文件类型上传，该文件类型是：" + ext);
                // 可以在此构建 ActionMessage 对象并返回页面显示错误
                return mapping.getInputForward();
            }
            if (fsize > 1024 * 1024) ...{// 判断文件大小是否为允许上传的大小。
                // 可以在此构建 ActionMessage 对象并返回页面显示错误
                System.out.println("文件太大");
                return mapping.getInputForward();
            }
            InputStream in = null;// 输入流用来读取用户上传文件
            OutputStream out = null;// 用来将用户上传文件存出在服务器特定目录中。
            try ...{
                in = file.getInputStream();
                out = new FileOutputStream(dir + fname);
```

```
            int byteread = 0;
            byte[] bytes = new byte[8192];
            while ((byteread = in.read(bytes, 0, 8192)) != -1) ...{
                out.write(bytes, 0, byteread);
            }
        } catch (Exception e) ...{
            System.out.println(e.getMessage());
            return mapping.getInputForward();
        } finally ...{
            try ...{
                in.close();
                out.close();
            } catch (Exception e) ...{
                System.out.println(e.getMessage());
                return mapping.getInputForward();
            }
        }
        System.out.println("chenggong");
        return mapping.findForward("suc");//suc 只是一个成功跳转的设置.
    }
```

2.5.2 严格分配文件访问权限策略

使用最严格的访问权限策略来保护上传或新建的文件和目录。

2.5.3 安全的临时文件

为了安全的使用临时文件，应该在程序初始化时创建一个只能被该程序读写的文件夹。不要将该文件夹放在用户可访问到的地方，并将所有的临时文件都放在其中。

2.5.4 访问竞争条件

使用文件句柄来保证多个操作的对象是同一个文件。文件访问竞争条件可以使攻击者能够获取系统中的任何文件，或者以任意字符覆盖原有的口令。为了防止这种缺陷，最好能确保在程序对某文件进行一系列操作之后不再能被替换或者修改。在C 语言中，最好

避免使用操作文件名的函数，因为你不能够保证在函数调用域之外的空间上，它指向的仍是硬盘上同一文件。所以应该首先打开该文件获得文件句柄，然后通过使用操作文件句柄的函数来进行操作。

2.6 开发调试

2.6.1 日志

- 集中日志记录

  集中式的日志记录有助于通过日志记录反映出一个统一的系统视角，并且方便修改。尽量避免使用 system.out 和 system.err 来进行记录。

- 使用时间戳

  必须明确的把时间戳包含在日志文件内，时间戳可以形成一条清晰的事件时间线。

- 记录每个重要行为

  确保在某种重要行为发生时有相应的记录。其中重要行为主要包括：管理指令、网络通信、身份认证尝试、更改对象所有权等。

- 保护日志文件

  大多数日志记录系统都是写入日子文件，但是更高级的系统应该使用数据库保存日志文件。不论日志记录到哪里，都应当阻止攻击者获得系统的重要信息或对日志文件进行恶意修改。

2.6.2 调试

- 最终产品中不应包含调试代码和信息

  谨慎的将调试代码和系统分离开来，以便它绝对不出现在产品部署中。每个人都增加代码容易泄露系统的额外信息。

- 产品中不包含后门代码

  后门代码是调试代码的一个特殊情况，后门访问代码允许开发者和测试者通过终端用户不能用的方法访问应用程序。后门代码也是应用程序必要的测试组成部分。但是，当这种类型的调试代码意外的存在于产品中时，这个应用程序无意之中开放了交互模式，攻击者就找到了一条非正常进入系统的通路。

- 清除备份和测试文件

  确保那些没用的、临时的、备份的文件不出现在产品中。

2.6.3 错误处理

- 检查返回值

  利用函数的返回值表示函数运行成功或失败的方法并非是个坏主意。

- 抛出捕获异常

  只需要捕获需要处理的异常，过度捕获异常并不是一个好的习惯。这可能导致某些安全问题。已经检测的异常非常有用，因为它们要求程序员思考潜在的错误条件。