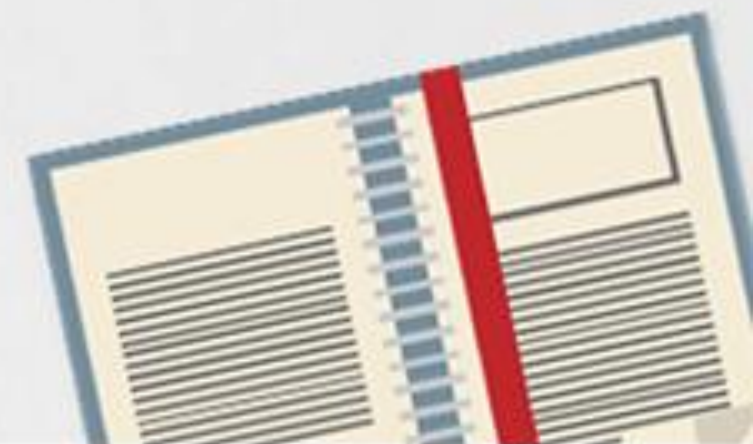
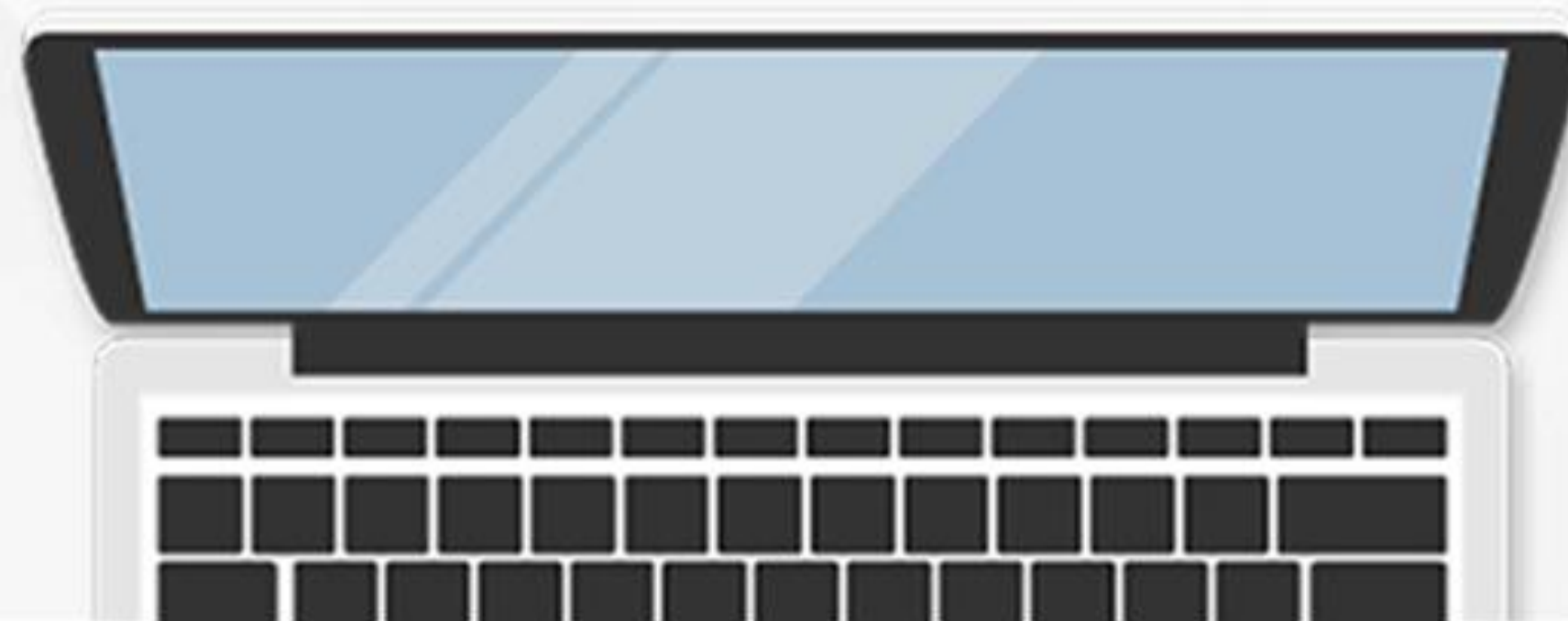
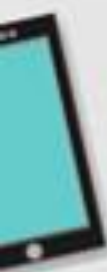


# JAVA安全开发常识



# 目录

## CATALOGUE

- 第一章：背景
- 第二章：拒绝服务
- 第三章：敏感信息
- 第四章：注入问题
- 第五章：可访问性和可扩展性
- 第六章：序列化和反序列化问题
- 第七章：编码问题
- 第八章：第三方组件安全问题
- 第九章：安全编码原则
- 第十章：源代码检测产品

# 第一章：背景

Java平台设计因素之一就是为了提供一个安全的代码环境。虽然Java自身的安全体系结构侧重于保护用户和系统免受网络上的恶意程序攻击。但是如果受信任的代码执行错误指令信息，则会不可避免的造成安全问题，这样的安全问题，往往包含在JAVA应用之中，包括访问文件，控制打印机、控制摄像头，系统执行恶意删除命令等。追其根源大部分都是编码设计中出现了纰漏。

为了尽量减少编程错误引起的安全缺陷，Java开发者应该遵循建议的安全开发规范。

# 第二章：拒绝服务

- 定义

通过恶意手段耗尽被攻击对象的资源，目的是让目标计算机或网络无法提供正常的服务或资源访问，使目标系统或服务系统停止响应甚至崩溃，而在此攻击中并不包括侵入目标服务器或目标网络设备。这些服务资源包括网络带宽、文件系统空间容量、开放的进程或者允许的连接。这种攻击会导致资源匮乏，无论计算机的处理速度多快、内存容量多大、网络带宽的速度多快都无法避免这种攻击带来的后果。

# 拒绝服务

- “Zip炸弹”

Zip是一种压缩文件格式，如果文件以文本格式为主的话，压缩比例会很高。150mb的纯文本格式，可以压缩到590kb。因此恶意攻击者频繁发送这种文件给后端解析，资源很快会被耗尽。

```
static final int BUFFER = 512;
// ...

// external data source: filename
BufferedOutputStream dest = null;
FileInputStream fis = new FileInputStream(filename);
ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
ZipEntry entry;
while ((entry = zis.getNextEntry()) != null) {
    System.out.println("Extracting: " + entry);
    int count;
    byte data[] = new byte[BUFFER];
    // write the files to the disk
    FileOutputStream fos = new FileOutputStream(entry.getName());
    dest = new BufferedOutputStream(fos, BUFFER);
    while ((count = zis.read(data, 0, BUFFER)) != -1) {
        dest.write(data, 0, count);
    }
    dest.flush();
    dest.close();
}
zis.close();
```

# 拒绝服务

- “Zip炸弹”

解决方案： 限制文件大小及文件类型

```
static final int TOOBIG = 0x6400000; // 100MB

// ...

// write the files to the disk, but only if file is not insanely big
if (entry.getSize() > TOOBIG) {
    throw new IllegalStateException("File to be unzipped is huge.");
}
if (entry.getSize() == -1) {
    throw new IllegalStateException(
        "File to be unzipped might be huge.");
}
FileOutputStream fos = new FileOutputStream(entry.getName());
dest = new BufferedOutputStream(fos, BUFFER);
while ((count = zis.read(data, 0, BUFFER)) != -1) {
    dest.write(data, 0, count);
}
```



# 拒绝服务

- Xml外部实体攻击（较少见）

Xml文件可以动态的加载内容，外部数据可以通过DTD嵌入到xml文档中，通过构造指向本地路径（/dev/random 或者 /dev/tty），当程序加载的时候，程序可能会意外终止或者阻塞，导致服务不能完成。例如：

```
<?xml version="1.0"?>
<!DOCTYPE foo SYSTEM "file:/dev/tty">
<foo>bar</foo>
```

当程序加载的时候，会尝试读入文件/dev/tty,在一些linux系统，读取这个文件，会阻塞程序，直到系统终端输入数据。

# 拒绝服务

- Xml外部实体攻击

## 解决方案：

定义白名单，自定义实现EntityResolver接口，实现接口方法public InputSource resolveEntity (String publicId, String systemId)，过滤systemId。



# 拒绝服务

- 文件资源释放

使用文件资源进行操作后，并未对其释放，例如：

```
try (final InputStream in = new FileInputStream(file)) {  
    use(in);  
}
```

如果对文件资源的申请过多，会因为资源处理完未释放，导致内存资源耗尽，需要关闭资源。

```
final InputStream in = new FileInputStream(file);  
try {  
    use(in);  
} finally {  
    in.close();  
}
```

# 拒绝服务

- 数据库资源释放

没有正确及时关闭数据库连接资源(Connection, Statement, ResultSet)。

```
Statement stmt = null;
ResultSet rs = null;
Connection conn = getConnection();
try {
    stmt = conn.createStatement();
    rs = stmt.executeQuery(sqlQuery);
    processResults(rs);
} catch (SQLException e) {
    // forward to handler
} finally {
    try {
        if (rs != null) {rs.close();}
    } catch (SQLException e) {
        // forward to handler
    } finally {
        try {
            if (stmt != null) {stmt.close();}
        } catch (SQLException e) {
            // forward to handler
        } finally {
```

```
            try {
                if (conn != null) {conn.close();}
            } catch (SQLException e) {
                // forward to handler
            }
        }
    }
}
```

# 第三章：敏感信息

在程序捕获到错误或异常行为时，经常会直接将错误信息反馈到用户端，攻击者会利用此类问题获取系统敏感信息，以对系统进行更进一步的攻击。

# 敏感信息

- 避免直接输出异常信息

当系统数据或调试信息通过输出流或者日志功能流出程序时就会发生信息泄漏。

例如：以下代码针对标准的错误流输出了一个异常：

```
try {  
...  
} catch (Exception e) {  
e.printStackTrace();  
}
```

依据这一系统配置，该信息可转储到控制台，写成日志文件，或者显示给远程用户。在某些情况下，该错误消息恰好可以告诉攻击者入侵这一系统的可能性究竟有多大。例如，一个数据库错误消息可以揭示应用程序容易受到 **SQL Injection** 攻击。其他的错误消息可以揭示有关该系统的更多间接线索。在上述例子中，搜索路径可以暗示应用程序安装在哪种操作系统上，以及管理员在配置应用程序时都做了哪些方面的努力。

# 敏感信息

- 避免在日志上输出敏感信息

当系统通过日志功能将未处理的用户信息直接输出到日志文件中，一旦黑客攻入系统，获取日志，就会发生信息泄漏。

例如：以下代码输出了登录操作的日志，日志里包含了敏感信息：

```
if (loginSuccessful) {  
    logger.save("User login succeeded for: " + username+ "pw: " +pw);  
}
```



# 敏感信息

- 避免明文保存敏感配置信息

在各类应用系统中，经常需要使用口令、客户账号、卡号等信息。例如，可能需要使用用户 ID 和密码来连接到数据库，或者可能存储客户用于访问应用程序的用户 ID 和密码。虽然可以在通过网络传递信息时使用安全套接字层 (SSL) 来加密信息，但是，当信息同时存储在服务器和客户端上时，还是必须对其进行保护，一旦黑客攻入系统，获取配置文件，就可以对其中的某些系统和数据造成危害  
例如：将数据库地址、用户名、密码明文保存在应用配置文件中。



# 编码问题

- 硬编码问题

代码中带有敏感信息（连接地址，用户名，密码等），导致信息泄露，增加安全隐患。

建议：

将敏感数据保存在属性文件中，无论什么时候需要这些数据，都可以从该文件读取。如果数据极其敏感，那么在访问属性文件时，应用程序应该使用一些加密 / 解密技术。

# 第四章：注入问题

- 跨站攻击

- 持久式XSS：恶意代码持久保存在服务器上，即Persistent。
- 反射式XSS：恶意代码不保留在服务器上，而是通过其他形式实时通过服务器反射给普通用户。

# 注入问题

- 跨站攻击--标准化字符序列

通常为了避免诸如跨站注入的出现，在程序中通过系统拦截器和过滤器对外部输入的字符过滤（例如过滤<script>），但是如果外部输入使用的是UNICODE编码，例如s="\uFE64" + "script" + "\uFE65"，而\uFE64和\uFE65标准化模式分别是<和>，则能有效地避开过滤器。

解决方案：使用函数对外部输入标准化

```
// Normalize
s = Normalizer.normalize(s, Form.NFKC);

// Validate
Pattern pattern = Pattern.compile("[<>]");
Matcher matcher = pattern.matcher(s);
if (matcher.find()) {
    // Found black listed tag
    throw new IllegalStateException();
}
```

# 注入问题

- 动态SQL问题

SQL 注入是一种攻击方式，在这种攻击方式中，恶意代码被插入到字符串中，然后将该字符串传递到数据库系统的实例以进行分析和执行。任何构成 SQL 语句的过程都应进行注入漏洞检查，因为数据库系统将执行其接收到的所有语法有效的查询。一个有经验的、坚定的攻击者甚至可以操作参数化数据。

```
...  
String userName = ctx.getAuthenticatedUserName();  
String itemName = request.getParameter("itemName");  
String query = "SELECT * FROM items WHERE owner = '"  
                + userName + "' AND itemname = '"  
                + itemName + "'";  
ResultSet rs = stmt.execute(query);
```

但是，由于这个查询是动态构造的，由一个常数基查询字符串和一个用户输入字符串连接而成，因此只有在 `itemName` 不包含单引号字符时，才会正确执行这一查询。如果一个用户名为 `wiley` 的攻击者在 `itemName` 中输入字符串“`name' OR 'a'='a`”，那么构造的查询就会变成：

附加条件 `OR 'a'='a` 会使 `where` 从句永远评估为 `true`;

# 注入问题

- 动态SQL问题

解决:

```
PreparedStatement prep = conn.prepareStatement("SELECT * FROM USERS WHERE  
PASSWORD=?");  
prep.setString(1, pwd);
```

过滤敏感字符。常见的一些敏感的sql特殊字符如下（以|分隔符）：

'|and|exec|insert|select|delete|update|count|\*|%|chr|mid|master|truncate|char|declare|;|or|-|+|,

# 注入问题

- XML注入问题

Xml因为其灵活可扩展性以及其跟JAVA语言的无缝衔接，广泛应用在JAVA应用开发中。但是XML

自身也会存在注入问题，例如：

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity>
</item>
```

外部输入的购买量：

```
1</quantity><price>1.0</price><quantity>1
```

最后的结果为：

```
<item>
  <description>Widget</description>
  <price>500.0</price>
  <quantity>1</quantity><price>1.0</price><quantity>1</quantity>
</item>
```

如果使用( org.xml.sax 或者 javax.xml.parsers.SAXParser)解析XML，则价格最后显示为1



# 注入问题

- Xml注入问题

解决：

使用XSD约束（推荐）

过滤敏感字符。常见的一些敏感的XML特殊字符如下（以|分隔符）：<|>|CDATA|--等

# 注入问题

- Xpath注入

随着简单 XML API、Web 服务和 Rich Internet Applications (RIAs) 的发展,已经有很多人使用 XML

文档代替关系数据库,但XML应用程序可能容易受到代码注入的攻击,尤其是 **XPath** 注入攻击。

```
<?xml version="1.0" encoding="UTF-8"?>
<users>
<user>
<firstname>Ben</firstname>
<lastname>Elmore</lastname>
<loginID>abc</loginID>
<password>test123</password>
</user>
<user>
<firstname>Shlomy</firstname>
<lastname>Gantz</lastname>
<loginID>xyz</loginID>
<password>123test</password>
</user>
</users>
```

```
XPath xpath = factory.newXPath();
XPathExpression expr =
    xpath.compile("//users/user[loginID
        /text()='"+loginID
        +" and
        password/text()='"+password+" " ]
        /firstname/text()");
Object result = expr.evaluate(doc,
    XPathConstants.NODESET);
NodeList nodes = (NodeList) result;

//绕过身份验证
//users/user[LoginID/text()=' or 1=1
    and password/text()=' or 1=1]
```

# 注入问题

- Xpath注入

## 解决：

检查提交的数据是否包含特殊字符，对特殊字符进行编码转换或替换、删除敏感字符或字符串。

数据提交到服务器端，在服务端正式处理这批数据之前，对提交数据的合法性再次进行验证。

# 注入问题

- 将不可信的外部数据传入**exec**函数（命令行注入）

java API中的**Runtime.exec()**函数，用来执行系统命令，例如执行一个**shell**脚本命令。因此导致命令行注入。

## 解决方案：

根据需求严格过滤命令，例如增加命令白名单，只允许在白名单中的命令可以通过执行。

编码实现某些应用特定需求，例如**ls/dir**命令：

```
import java.io.File;

class DirList {
    public static void main(String[] args) throws Exception {
        File dir = new File(System.getProperty("dir"));
        if (!dir.isDirectory()) {
            System.out.println("Not a directory");
        } else {
            for (String file : dir.list()) {
                System.out.println(file);
            }
        }
    }
}
```

# 注入问题

- 正则表达式注入

正则表达式广泛应用于文本字符的匹配，**JAVA**语言也提供了相应的接口函数和规则用来实现正则表达式。但是如果不合理的使用也会产生一些问题。例如：

某日志格式如下：

```
10:47:03 private[423] Successful logout name: usr1 ssn: 111223333
10:47:04 public[48964] Failed to resolve network service
10:47:04 public[1] (public.message[49367]) Exited with exit code: 255
10:47:43 private[423] Successful login name: usr2 ssn: 444556666
10:48:08 public[48964] Backup failed with error: 19
```

其提供的查询接口：

```
(.*? +public\[\\d+\\] +.*<SEARCHTEXT>.*)
```

searchtext为查询内容

输入如下字符：.\*)|(.\*，最后效果为(.\*)? +public\[\\d+\\] +.\*.\*)|(.\*.\*)，可以查询所有信息

# 注入问题

- 正则表达式注入

## 解决方案：

输入过滤，对输入的查询条件过滤，禁止不期望的字符出现；  
输出过滤，对于不符合要求的输出过滤；



# 第五章：可访问性和可扩展性

- 限制类，接口，方法和字段的可访问

**Java**包括一组相关的**Java**类和接口。如果它被指定为发布的**API**的一部分，那么将其声明为共有的。类成员和构造函数和方法也根据需求声明为**public**或者**protect**，否则声明为私有的，以避免直接对外暴露，限制外部访问权限。需要注意的是接口的成员是隐式公开。

# 可访问性和可扩展性

- 限制包的可访问性

通过限制包的访问权限并赋予特定的代码来防止不可信代码对包成员的访问，能够有效的阻止通过反射机制等加载包成员。

```
private static final String PACKAGE_ACCESS_KEY = "package.access";  
static {  
    String packageAccess = java.security.Security.getProperty(  
        PACKAGE_ACCESS_KEY  
    );  
    java.security.Security.setProperty(  
        PACKAGE_ACCESS_KEY,   
        (  
            (packageAccess == null ||  
             packageAccess.trim().isEmpty()) ?  
            "" :  
            (packageAccess + ",")  
        ) +  
        "xx.example.product.implementation."  
    );  
}
```

# 可访问性和可扩展性

- **Final 类和方法**

不允许扩展的类和方法应该声明为 **final**，这样做防止了系统外的代码扩展类并修改类的行为。

避免使用非 **final** 的公共静态变量，这样的变量附着在类（而非类的实例）上，而类可以被其它类所定位。其结果就是静态域变量可以被其它类找到并使用，因此很难保证它们的安全。（通俗点就是非 **final** 的公共 **static** 变量可以随意改变，一旦被改变后，其它地方再使用该变量会导致不可控后果）

# 可访问性和可扩展性

- Clone问题

浅拷贝

直接调用**clone**函数，对基本类型的数据复制，对象数据采用的是引用模式。

深拷贝

重写**clone**函数，根据需要，对源对象属性，重新生成一份。

# 第六章： 序列化和反序列化

- **Transient**

在包含系统资源的直接句柄和相对地址空间信息的字段前使用**transient** 关键字。如果资源，如文件句柄，不被声明为**transient**，该对象在序列化状态下可能会被修改，从而使得被反序列化后获取对资源的不当访问。

# 序列化和反序列化

- 字节流加密

保护虚拟机外的字节流的另一方式是对序列化包产生的流进行加密。字节流加密防止解码或读取被序列化的对象的私有状态。

如果决定加密，应该管理好密钥，密钥的存放地点以及将密钥交付给反序列化程序的方式等。

避免使用自制的加密算法。

案例分析：**Apache Struts S2-055** 反序列化漏洞，影响影响版本：**Version 2.5.0 to 2.5.12** 和 **Version 2.3.0 to 2.3.33**，属于高危缺陷



# 第七章：编码问题

- 忽视调用方法的返回值

方法的返回值可以用来判断一次程序调用是否成功，因此有必要对返回值做出判断，而非忽略掉。例如：

```
public void deleteFile() {  
    File someFile = new File("someFileName.txt");  
    // do something with someFile  
    someFile.delete();  
}
```

如果文件并未删除成功，可能对后续的操作带来影响，因此要增加相应的判断。

```
public void deleteFile() {  
    File someFile = new File("someFileName.txt");  
    // do something with someFile  
    if (!someFile.delete()) {  
        // handle failure to delete the file  
    }  
}
```

# 编码问题

- 空指针引用

代码中经常出现，主要是引用了未初始化的对象变量，从而造成`NullPointerException`，属于代码质量问题，影响较大，是常见的缺陷。

解决方法：在对象的使用过程中，对其进行判断是否为空。

# 编码问题

- 整数溢出

在计算机中，数据以补码的形式存在，当出现整数运算的时候，即使对边界做了检查，也有可能導致整数溢出。

```
private void checkGrowBy(long extra) {  
    if (extra < 0 || current + extra > max) {  
        throw new IllegalArgumentException();  
    }  
}
```

current+extra可能会导致整数溢出，但是同号相减不会出现这个问题：

```
private void checkGrowBy(long extra) {  
    if (extra < 0 || current > max - extra) {  
        throw new IllegalArgumentException();  
    }  
}
```

解决方案：

先决条件检查；向上类型转换；使用**BigInteger**；

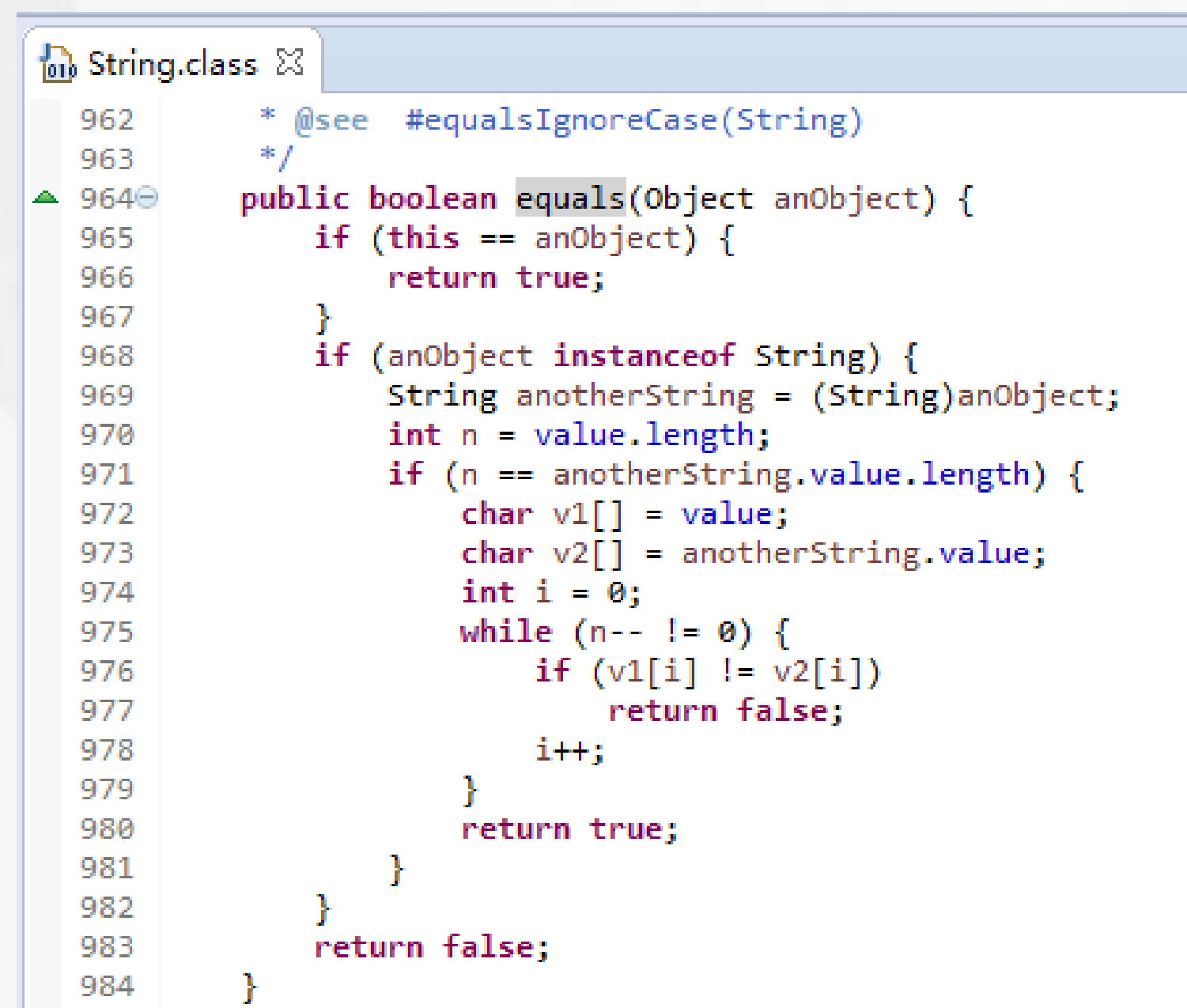
# 编码问题

- ==和equals ()

==用于基本类型的操作，例如int，long；在对象应用==的时候主要是对地址的比较。

在比较对象时，开发者通常会比较对象的属性。在没有明确实现equals() 的类（或任何超类/接口）上用 equals() 会导致调用继承java.lang.Object 的 equals() 方法。Object.equals() 将比较两个对象实例，查看它们是否相同，而不是比较对象成员字段或其他属性。

（equals方法尽量重写，而不是直接使用父类方法）

A screenshot of a code editor showing the implementation of the equals() method in the String class. The code is in Java and includes a Javadoc comment, a public boolean equals(Object anObject) method signature, and the implementation logic. The implementation first checks for self-equality (this == anObject), then checks if the other object is an instance of String, and finally compares the character arrays element by element.

```
String.class
962      * @see #equalsIgnoreCase(String)
963      */
964      public boolean equals(Object anObject) {
965          if (this == anObject) {
966              return true;
967          }
968          if (anObject instanceof String) {
969              String anotherString = (String)anObject;
970              int n = value.length;
971              if (n == anotherString.value.length) {
972                  char v1[] = value;
973                  char v2[] = anotherString.value;
974                  int i = 0;
975                  while (n-- != 0) {
976                      if (v1[i] != v2[i])
977                          return false;
978                      i++;
979                  }
980                  return true;
981              }
982          }
983          return false;
984      }
```

# 编码问题

- **&和&&**

Java中&&和&都是表示与的逻辑运算符，都表示逻辑运算符and，当两边的表达式都为true的时候，整个运算结果才为true，否则为false。

&&的短路功能，当第一个表达式的值为false的时候，则不再计算第二个表达式；&则两个表达式都执行。

&可以用作位运算符，当&两边的表达式不是Boolean类型的时候，&表示按位操作.下述案例存在严重缺陷，一旦a为null,使用&做为逻辑表达式，会继续执行a.getName()，此时抛出空指针异常，程序异常中断

代码片段：

```
if(a!=null&a.getName()!=null)
```

# 编码问题

- 返回数组问题

某个方法返回一个对敏感对象的内部数组的引用，假定该方法的调用程序不改变这些对象。即使数组对象本身是不可改变的，也可以在数组对象以外操作数组的内容，这种操作将反映在返回该数组的对象中。

```
public class XXX {  
    private String[] xxxx;  
    public String[] getXXX() {  
        return xxxx;  
    }  
}
```

安全代码：

```
public class XXX {  
    private String[] xxxx;  
    public String[] getXXX() {  
        String temp[] = Arrays.copyOf(...);  
        return temp;  
    }  
}
```



# 编码问题

- 读取字节流或者字符流的方法，使用int类型的返回值

`InputStream.read()`提供了从输入流中返回一个0-255的整数, `FileReader.read`返回的值为0-65535.同时都以返回值为-1，作为读取流结束的标志。但是很多程序员使用如下方式：

```
FileInputStream in;  
// initialize stream  
byte data;  
while ((data = (byte) in.read()) != -1) {  
    // ...  
}
```

```
FileReader in;  
// initialize stream  
char c;  
while ((c = (char) in.read()) != -1) {  
    // ...  
}
```

因为数据在计算机中补码中存储。如果返回值是255（0xff，-1的补码为11111111），在byte中表示为-1，所以导致程序的意外终止。同样Reader也会出现上述问题

```
FileInputStream in;  
// initialize stream  
int inbuff;  
byte data;  
while ((inbuff = in.read()) != -1) {  
    data = (byte) inbuff;  
    // ...  
}
```

```
FileReader in;  
// initialize stream  
int inbuff;  
char data;  
while ((inbuff = in.read()) != -1) {  
    data = (char) inbuff;  
    // ...  
}
```

# 编码问题

- 本地方法调用安全

**JAVA**不会像**C**语言一样出现数组越界问题，**JAVA**中数组越界会抛出异常，而**C**语言不会报错，会继续执行，导致常见的堆栈溢出的攻击。如果在使用**JNI**对**C**语言调用的时候，就需对传入的参数做合理的检查。

```
public final class NativeMethod {  
  
    // public native method  
    public native void nativeOperation(byte[] data, int offset, int len);  
  
    // wrapper method that lacks security checks and input validation  
    public void doOperation(byte[] data, int offset, int len) {  
        nativeOperation(data, offset, len);  
    }  
  
    static {  
        // load native library in static initializer of class  
        System.loadLibrary("NativeMethodLib");  
    }  
}
```

# 编码问题

- 本地方法调用安全

载入的时候对传入的参数做可控安全检查，防止在JAVA平台外，导致的安全问题。

```
public final class NativeMethodWrapper {  
  
    // private native method  
    private native void nativeOperation(byte[] data, int offset, int len);  
  
    // wrapper method performs SecurityManager and input validation checks  
    public void doOperation(byte[] data, int offset, int len) {  
        // permission needed to invoke native method  
        securityManagerCheck();  
  
        if (data == null) {  
            throw new NullPointerException();  
        }  
  
        // copy mutable input  
        data = data.clone();  
  
        // validate input  
        if ((offset < 0) || (len < 0) || (offset > (data.length - len))) {  
            throw new IllegalArgumentException();  
        }  
  
        nativeOperation(data, offset, len);  
    }  
  
    static {  
        // load native library in static initializer of class  
        System.loadLibrary("NativeMethodLib");  
    }  
}
```



# 第八章： 第三方组件安全问题

- 慎重使用第三方组件

第三方的组件可能可以提供程序员想要的功能，但是我们还需要安全，而不单单是功能。代码再安全可靠，也可能被第三方组件毁于一旦。  
例如：**struts**、**spring**、**flash**等常被曝出重大安全漏洞。

尽量从官方渠道下载第三方组件；  
对第三方组件的安全测试；  
对第三方组件的安全加固；  
对第三方组件的安全补丁更新；  
对第三方组件的定期升级计划；

# 第九章：安全编码原则

## • 安全编码原则

程序只实现你指定的功能；  
永远不要信任用户输入，对用户输入数据做有效性检查；  
必须考虑异常情况并进行处理；  
不要试图在发现错误之后继续执行；  
尽可能使用安全函数进行编程；  
小心、认真、细致地编程；

# THANKS

