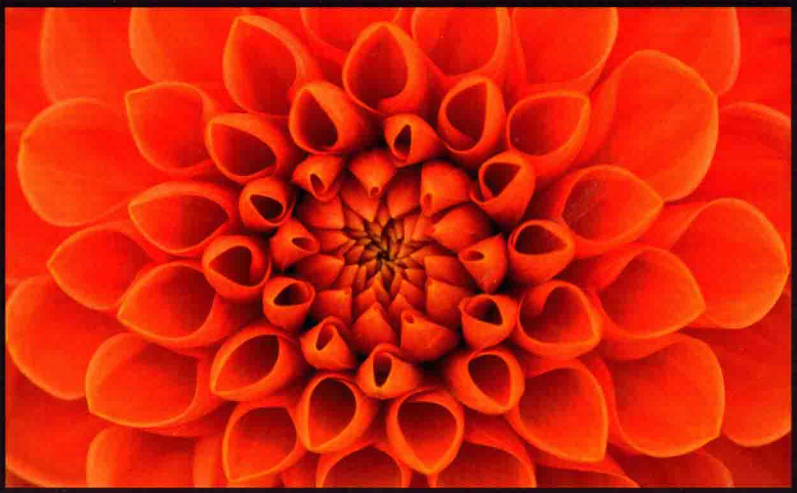
PEARSON



Java编码指南

编写安全可靠程序的75条建议

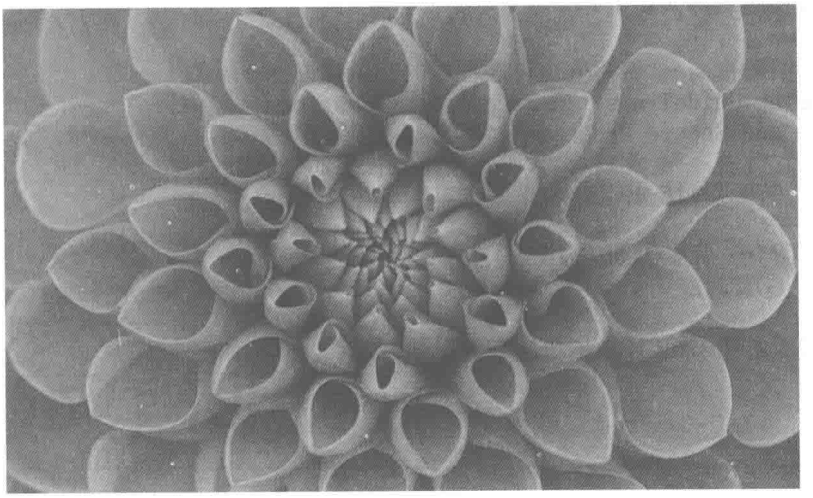
（英文版）

[美] Fred Long Dhruv Mohindra Robert C. Seacord
Dean F. Sutherland David Svoboda 著

Java Coding Guidelines

75 Recommendations for Reliable and Secure Programs

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS



Java编码指南

编写安全可靠程序的75条建议

（英文版）

[美] Fred Long Dhruv Mohindra Robert C. Seacord
Dean F. Sutherland David Svoboda 著

Java Coding Guidelines

75 Recommendations for Reliable and Secure Programs

人民邮电出版社
北 京

图书在版编目（CIP）数据

Java编码指南 : 编写安全可靠程序的75条建议 = Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs : 英文 / (美) 朗 (Long,F.) 等著. -- 北京 : 人民邮电出版社, 2015.10
ISBN 978-7-115-40401-5

Ⅰ. ①J… Ⅱ. ①朗… Ⅲ. ①JAVA语言－程序设计－英文 Ⅳ. ①TP312

中国版本图书馆CIP数据核字(2015)第222143号

内 容 提 要

本书是《Java 安全编码标准》一书的扩展，书中把那些不必列入 Java 安全编码标准但是同样会导致系统不可靠或不安全的 Java 编码实践整理出来，并为这些糟糕的实践提供了相应的文档和警告，以及合规的解决方案。读者可以将本书作为 Java 安全方面的工具书，根据自己的需要，找到自己感兴趣的规则进行阅读和理解，或者在实际开发中遇到安全问题时，根据书中列出的大致分类对规则进行索引和阅读，也可以通读全书的所有规则，系统地了解 Java 安全规则，增强对 Java 安全特性、语言使用、运行环境特性的理解。

本书给出了帮助 Java 软件工程师设计出高质量的、安全的、可靠的、强大的、有弹性的、可用性和可维护性高的软件系统的 75 条编码指南，适合所有 Java 开发人员阅读，也适合高等院校教师和学生学习和参考。

◆ 著　[美] Fred Long　Dhruv Mohindra　Robert C. Seacord　Dean F. Sutherland　David Svoboda

责任编辑　杨海玲

责任印制　张佳莹　焦志炜

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路 11 号

邮编　100164　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

固安县铭成印刷有限公司印刷

◆ 开本：720×960　1/16

印张：18

字数：298 千字　2015 年 10 月第 1 版

印数：1－2 000 册　2015 年 10 月河北第 1 次印刷

著作权合同登记号　图字：01-2015-6171 号

定价：59.00 元

读者服务热线：(010)81055410　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

目录

Chapter 1

Security

The Java programming language and runtime system were designed with security in mind. For example, pointer manipulation is implicit and hidden from the programmer, and any attempt to reference a null pointer results in an exception being thrown. Similarly, an exception results from any attempt to access an array or a string outside of its bounds. Java is a strongly typed language, and all implicit type conversions are well defined and platform independent, as are the arithmetic types and conversions. The Java Virtual Machine (JVM) has a built-in bytecode verifier to ensure that the bytecode being run conforms to the Java Language Specification: Java SE 7 Edition (JLS) so that all the checks defined in the language are in place and cannot be circumvented.

The Java class loader mechanism identifies classes as they are loaded into the JVM, and can distinguish between trusted system classes and other classes that may not be trusted. Classes from external sources can be given privileges by digitally signing them; these digital signatures can also be examined by the class loader, and contribute to the class's identification. Java also provides an extensible fine-grained security mechanism that enables the programmer to control access to resources such as system information, files, sockets, and any other security-sensitive resources that the programmer wishes to use. This security mechanism can require that a runtime security manager be in place to enforce a security policy. A security manager and its security policy are usually specified by command-line arguments, but they may be installed programmatically, provided that such an action is not already disallowed by an existing security policy. Privileges to access resources may be extended to nonsystem Java classes by relying on the identification provided by the class loader mechanism.

Enterprise Java applications are susceptible to attacks because they accept untrusted input and interact with complex subsystems. Injection attacks (such as cross-site scripting [XSS], XPath, and LDAP injection) are possible when the components susceptible to these attacks are used in the application. An effective mitigation strategy is to whitelist input, and encode or escape output before it is processed for rendering.

This chapter contains guidelines that are concerned specifically with ensuring the security of Java-based applications. Guidelines dealing with the following security nuances are articulated.

1. Dealing with sensitive data
2. Avoiding common injection attacks
3. Language features that can be misused to compromise security
4. Details of Java's fine-grained security mechanism

■ 1. Limit the lifetime of sensitive data

Sensitive data in memory can be vulnerable to compromise. An adversary who can execute code on the same system as an application may be able to access such data if the application

- Uses objects to store sensitive data whose contents are not cleared or garbage-collected after use
- Has memory pages that can be swapped out to disk as required by the operating system (for example, to perform memory management tasks or to support hibernation)
- Holds sensitive data in a buffer (such as `BufferedReader`) that retains copies of the data in the OS cache or in memory
- Bases its control flow on reflection that allows countermeasures to circumvent the limiting of the lifetime of sensitive variables
- Reveals sensitive data in debugging messages, log files, environment variables, or through thread and core dumps

Sensitive data leaks become more likely if the memory containing the data is not cleared after using the data. To limit the risk of exposure, programs must minimize the lifetime of sensitive data.

Complete mitigation (that is, foolproof protection of data in memory) requires support from the underlying operating system and Java Virtual Machine. For example, if swapping sensitive data out to disk is an issue, a secure operating system that disables swapping and hibernation is required.

Noncompliant Code Example

This noncompliant code example reads user name and password information from the console and stores the password as a `String` object. The credentials remain exposed until the garbage collector reclaims the memory associated with this `String`.

```
class Password {
  public static void main (String args[]) throws IOException {
    Console c = System.console();
    if (c == null) {
      System.err.println("No console.");
      System.exit(1);
    }

    String username = c.readLine("Enter your user name: ");
    String password = c.readLine("Enter your password: ");

    if (!verify(username, password)) {
      throw new SecurityException("Invalid Credentials");
    }

    // ...
  }

  // Dummy verify method, always returns true
  private static final boolean verify(String username,
      String password) {
    return true;
  }
}
```

Compliant Solution

This compliant solution uses the `Console.readPassword()` method to obtain the password from the console.

```
class Password {
  public static void main (String args[]) throws IOException {
    Console c = System.console();

    if (c == null) {
      System.err.println("No console.");
      System.exit(1);
    }

    String username = c.readLine("Enter your user name: ");
    char[] password = c.readPassword("Enter your password: ");
```

```
    if (!verify(username, password)) {
      throw new SecurityException("Invalid Credentials");
    }

    // Clear the password
    Arrays.fill(password, ' ');
  }

  // Dummy verify method, always returns true
  private static final boolean verify(String username,
     char[] password) {
    return true;
  }
}
```

The `Console.readPassword()` method allows the password to be returned as a sequence of characters rather than as a `String` object. Consequently, the programmer can clear the password from the array immediately after use. This method also disables echoing of the password to the console.

Noncompliant Code Example

This noncompliant code example uses a `BufferedReader` to wrap an `InputStreamReader` object so that sensitive data can be read from a file:

```
void readData() throws IOException{
  BufferedReader br = new BufferedReader(new InputStreamReader(
    new FileInputStream("file")));
  // Read from the file
  String data = br.readLine();
}
```

The `BufferedReader.readLine()` method returns the sensitive data as a `String` object, which can persist long after the data is no longer needed. The `BufferedReader.read(char[], int, int)` method can read and populate a `char` array. However, it requires the programmer to manually clear the sensitive data in the array after use. Alternatively, even if the `BufferedReader` were to wrap a `FileReader` object, it would suffer from the same pitfalls.

Compliant Solution

This compliant solution uses a directly allocated NIO (new I/O) buffer to read sensitive data from the file. The data can be cleared immediately after use and is not cached or buffered in multiple locations. It exists only in the system memory.

```
void readData(){
  ByteBuffer buffer = ByteBuffer.allocateDirect(16 * 1024);
  try (FileChannel rdr =
       (new FileInputStream("file")).getChannel()) {
    while (rdr.read(buffer) > 0) {
      // Do something with the buffer
      buffer.clear();
    }
  } catch (Throwable e) {
    // Handle error
  }
}
```

Note that manual clearing of the buffer data is mandatory because direct buffers are not garbage collected.

Applicability

Failure to limit the lifetime of sensitive data can lead to information leaks.

Bibliography

[API 2013]	Class `ByteBuffer`
[Oracle 2013b]	"Reading ASCII Passwords from an InputStream Example" from the Java Cryptography Architecture [JCA] Reference Guide
[Tutorials 2013]	I/O from the Command Line

2. Do not store unencrypted sensitive information on the client side

When building an application that uses a client–server model, storing sensitive information, such as user credentials, on the client side may result in its unauthorized disclosure if the client is vulnerable to attack.

For web applications, the most common mitigation to this problem is to provide the client with a cookie and store the sensitive information on the server. Cookies are created by a web server, and are stored for a period of time on the client. When the client reconnects to the server, it provides the cookie, which identifies the client to the server, and the server then provides the sensitive information.

Cookies do not protect sensitive information against cross-site scripting (XSS) attacks. An attacker who is able to obtain a cookie either through an XSS attack, or directly by attacking the client, can obtain the sensitive information from the server

using the cookie. This risk is timeboxed if the server invalidates the session after a limited time has elapsed, such as 15 minutes.

A cookie is typically a short string. If it contains sensitive information, that information should be encrypted. Sensitive information includes user names, passwords, credit card numbers, social security numbers, and any other personally identifiable information about the user. For more details about managing passwords, see Guideline 13, "Store passwords using a hash function." For more information about securing the memory that holds sensitive information, see Guideline 1, "Limit the lifetime of sensitive data."

Noncompliant Code Example

In this noncompliant code example, the login servlet stores the user name and password in the cookie to identify the user for subsequent requests:

```
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) {

  // Validate input (omitted)

  String username = request.getParameter("username");
  char[] password =
    request.getParameter("password").toCharArray();
  boolean rememberMe =
    Boolean.valueOf(request.getParameter("rememberme"));

  LoginService loginService = new LoginServiceImpl();

  if (rememberMe) {
    if (request.getCookies()[0] != null &&
      request.getCookies()[0].getValue() != null) {
      String[] value =
        request.getCookies()[0].getValue().split(";");

      if (!loginService.isUserValid(value[0],
          value[1].toCharArray())) {
        // Set error and return
      } else {
        // Forward to welcome page
      }
    } else {
      boolean validated =
        loginService.isUserValid(username, password);
```

```
      if (validated) {
        Cookie loginCookie = new Cookie("rememberme", username +
                          ";" + new String(password));
        response.addCookie(loginCookie);
        // ... forward to welcome page
      } else {
        // Set error and return
      }
    }
  } else {
    // No remember-me functionality selected
    // Proceed with regular authentication;
    // if it fails set error and return
  }

  Arrays.fill(password, ' ');
}
```

However, the attempt to implement the remember-me functionality is insecure because an attacker with access to the client machine can obtain this information directly on the client. This code also violates Guideline 13, "Store passwords using a hash function."

Compliant Solution (Session)

This compliant solution implements the remember-me functionality by storing the user name and a secure random string in the cookie. It also maintains state in the session using `HttpSession`:

```
protected void doPost(HttpServletRequest request,
  HttpServletResponse response) {

  // Validate input (omitted)

  String username = request.getParameter("username");
  char[] password =
    request.getParameter("password").toCharArray();
  boolean rememberMe =
    Boolean.valueOf(request.getParameter("rememberme"));
  LoginService loginService = new LoginServiceImpl();
  boolean validated = false;
  if (rememberMe) {
    if (request.getCookies()[0] != null &&
        request.getCookies()[0].getValue() != null) {
```