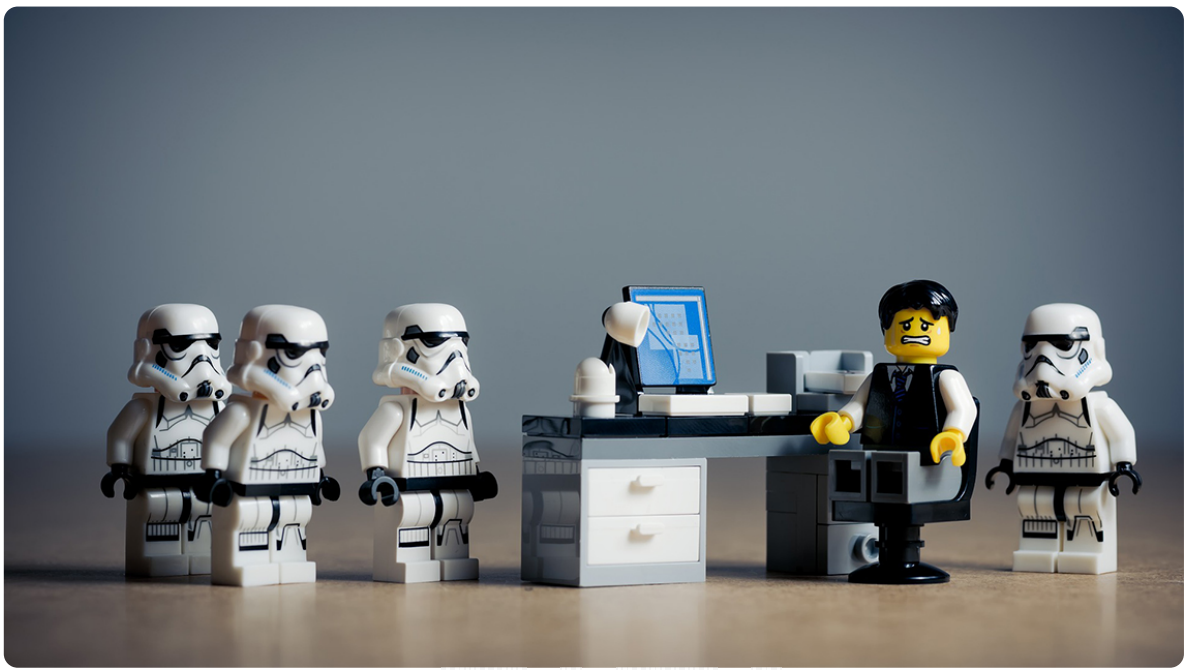
38 | 对象序列化的危害有多大?

范学雷 2019-04-01



00:00 09:02

讲述：刘飞　　大小：16.54M

如果一个函数或者对象，不管它位于多么遥远的地方，都可以在本地直接被调用，那该有多好呀！这是一个非常朴素、美好的想法。基于这个设想，诞生了很多伟大的技术和协议，比如远程过程调用（RPC）、远程方法调用（RMI）、分布式对象（Distributed Object）、组件对象模型（COM）、公共对象请求代理（CORBA）和简单对象访问协议（SOAP）等……这个列表还可以很长很长。

躲在这些协议背后的核心技术之一，就是**序列化**。简单地说，序列化就是要把一个使用场景中的一个函数或者对象以及它们的执行环境，打包成一段可以传输的数据，然后把该数据传输给另外一个使用场景。在这个使用场景中，该数据被拆解成适当的函数或者对象，包括该函数或者对象的执行环境。这样，该函数或者对象就可以在不同的场景下使用了。

数据拆解的过程，就是反序列化。打包、传输、拆解是序列化技术的三个关键步骤。由于传输的是数据，打包和拆解可能使用不同的编程语言，运行在不同的操作系统上。这样就带来了跨平台和跨语言的好处。而数据能够传输，就意味着可以带来分布式的好处。数据当然也可以存储，而可以存储意味着相关对象的生命周期的延长，这是不是也是一个非常值得兴奋的特点?

的确是一个美妙的想法，对吧？ **如果一个想法不是足够好，它也不会造成足够坏的影响。**

评审案例

我们用 Java 语言的例子来看看序列化的问题。先一起来看一段节选的 Java 代码。你能看出这段代码有什么问题吗？该怎么解决这个问题？

复制代码

```
public class Person implements Serializable {
    // <snipped>

    private String firstName;
    private String lastName;
    private String birthday;
    private String socialSecurityNumber;

    public Person(String firstName, String lastName,
            String birthday, String socialSecurityNumber) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.birthday = birthday;
        this.socialSecurityNumber = socialSecurityNumber;
    }

    // <snipped>
}

```

注意，socialSecurityNumber 表示社会保障号，是一个高度敏感、需要高度安全保护的数据。如果社会保障号以及姓名、生日等信息被泄露，那么冒名顶替者就可以用这个号码举债买房、买车，而真实用户则要背负相关的债务。一旦社会保障号被泄露，想要证明并不是你申请了贷款，远远不是一件轻而易举的事情。在有些国家，社会保障号的保护本身甚至都是一个不小的生意。在一个信息系统中，除了本人以及授权用户，任何其他人都不应该获知社会保障号以及相关的个人信息。

上述的代码，存在泄露社会保障号以及相关的个人信息的巨大风险。

案例分析

打包、传输、拆解是序列化技术的三个关键步骤。我们来分别看看这三个步骤。

首先，打包环节会把一个 Person 实例里的姓名、生日、社会保障号等信息转化为二进制数据。这段数据可以被传输、存储和拆解。任何人看到这段二进制数据，都可以拆解，还原成一个 Person 实例，从而获得个人敏感信息。这段二进制数据在传输和存储的过程中，有可能被恶意的攻击者修改，从而影响 Person 实例的还原。如果这个实例涉及到具体的商业交易，那么通过这样的攻击，还可以修改交易对象。

你看，序列化后的每一个环节，都有可能遭受潜在的攻击。序列化的问题有多严重呢？据说，**大约有一半的 Java 漏洞和序列化技术有直接或者间接的关系**。而且，由于序列化可以使用的场景非常多，序列化对象既可以看又可以改，这样就导致序列化安全漏洞的等级往往非常高，影响非

常大。甚至每年都会有公司专门收集、整理和分析序列化漏洞，这就加剧了序列化安全漏洞的影响，特别是对于那些没有及时修复的系统来说。

1997 年，Java 引入序列化技术，至今二十多年里，由于序列化技术本身的安全问题，Java 尝尽了其中的酸楚。这是一个“美妙”的想法带来的可怕错误。如果有一天，Java 废弃了序列化技术，那一点儿也不值得惊讶。毕竟，和得到的好处相比，要付出的代价实在是太沉重了！

如果你的应用还没有开始使用序列化技术，这很好，**不要惦记序列化的好处，坚持不要使用序列化**。如果你的应用已经使用了序列化技术，那么可以做些什么来防范或者降低序列化的风险呢？

额外的防护

序列化技术本身并没有内在的安全防护措施，这也是 Java 序列化为什么会这么令人诅丧的原因之一。如果一定要使用序列化技术，我们就需要设计、部署、加固序列化的安全防线。

我们先聊聊面对序列化带来的种种问题，该如何保护被序列化的敏感数据。

首先推荐的方式是，含有敏感数据的类，不要支持序列化。当然，这也就主动放弃了序列化带来的好处。

次优的方式是，不要序列化敏感数据，把敏感数据排除在序列化数据之外。比如，案例中的序列化数据可以抽象地表述为如下的四项：

复制代码

```
firstName | lastName | birthday | socialSecurityNumber

```

能不能把敏感的 socialSecurityNumber 和 birthday 排除在外呢？Java 语言的关键字 transient 就是为这一功能设计的。

复制代码

```
public class Person implements Serializable {
    // <snipped>

    private String firstName;
    private String lastName;
    private transient String birthday;              // sensitive data
    private transient String socialSecurityNumber;  // sensitive data

    public Person(String firstName, String lastName,
            String birthday, String socialSecurityNumber) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.birthday = birthday;
        this.socialSecurityNumber = socialSecurityNumber;
    }
```

```
    // <snipped>
}
```

如果把 socialSecurityNumber 和 birthday 变量声明为 transient，对象实例的序列化就会把这两个变量排除在外。这个时候，序列化数据就不包含敏感数据了。

复制代码

```
  firstName | lastName
```

排除敏感数据的序列化，还有另一种办法，那就是指定可以序列化的非敏感数据。如果把 transient 关键字提供的变量声明看成一个黑名单模式，Java 还提供了一个白名单模式。使用静态的 serialPersistentFields 变量，可以指定哪些变量可以序列化。上面的案例中，如果只序列化 firstName 和 lastName 变量，那么敏感的 socialSecurityNumber 和 birthday 变量自然就被排除在外了。

复制代码

```
public class Person implements Serializable {
    // <snipped>

    private String firstName;
    private String lastName;
    private String birthday;              // sensitive data
    private String socialSecurityNumber;  // sensitive data

    // list of serializable fields
    private static final ObjectStreamField[]
        serialPersistentFields = {
            new ObjectStreamField("firstName", Person.class),
            new ObjectStreamField("lastName", Person.class)
        };

    // <snipped>
}
```

可是，如果把敏感数据排除在序列化数据之外，也就意味着敏感数据不会在拆解后的对象实例中出现。这就使得序列化之前的实例和反序列化之后的实例并不一致。这种差异的存在，就足以使得序列化名存实亡，反序列化后的对象实例可能就没有太多的实际意义了。

那么有没有一种方法，既可以保护敏感数据，也能保持对象实例序列化前后的等价呢？办法还是有的。

如果在一个完全可信任的环境下，既不用担心敏感信息的泄露，也不用担心敏感信息的修改，更不用担心对象会被用于非可信的环境，敏感数据可以正常实例化了。然而，这严重限制了对象的使用环境，如果用错了环境，就会面临严肃的安全问题。

如果对象有可能适用于非可信的环境，就要使用复杂一些的技术。比如使用加密和签名技术，解决“谁能看”和“谁能改”的安全问题。可是，复杂技术的使用，几乎意味着我们对性能要求做出了妥协。面对这样的妥协，是否还需要使用序列化，有时候也是一个两难的选择。

小结

通过对这个评审案例的讨论，我想和你分享下面两点个人看法。

1. **序列化技术不是一个有安全保障的技术，序列化数据的传输和拆解过程都可能被攻击者利用**；
2. **要尽量避免敏感信息的序列化。**

除了上述我们说到的方法，敏感信息在序列化过程中的处理和保护，还有三种常见的方法：

1. 实现 writeObject，主动地、有选择地序列化指定数据。writeObject 和 serialPersistentFields 变量都是指定序列化数据，但区别在于 writeObject() 覆盖了序列化的缺省函数，所以编码可以更自由；
2. 实现 writeReplace， 使用序列化代理；
3. 实现 Externalizable 接口。

我们把这三种方法的使用，留给讨论区，欢迎你对这三种方法做总结、分析，并与我一起交流。

一起来动手

下面的这段 Java 代码，有一个隐藏的序列化安全问题。你能找到这个问题，并且解决掉这个问题吗?

复制代码

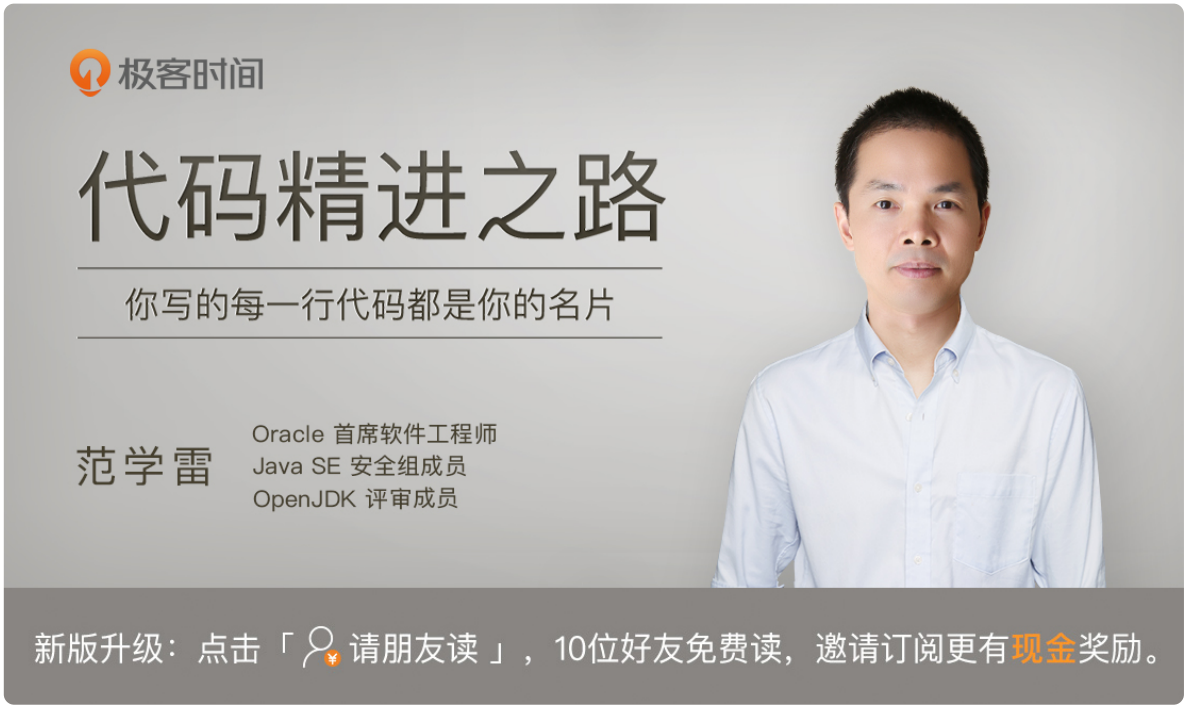
```
public class Person extends HashMap<String, String> {
    // <snipped>

    public Person(String firstName, String lastName,
            String birthday, String socialSecurityNumber) {
        super();
        super.put("firstName", firstName);
        super.put("lastName", lastName);
        super.put("birthday", birthday);
        super.put("socialSecurityNumber", socialSecurityNumber);
    }
    // <snipped>
}

```

欢迎你把自己看到的问题和想到的解决方案写在留言区，我们一起来学习、思考、精进！

如果你觉得这篇文章有所帮助，欢迎点击“请朋友读”，把它分享给你的朋友或者同事。



精选留言

> 由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。