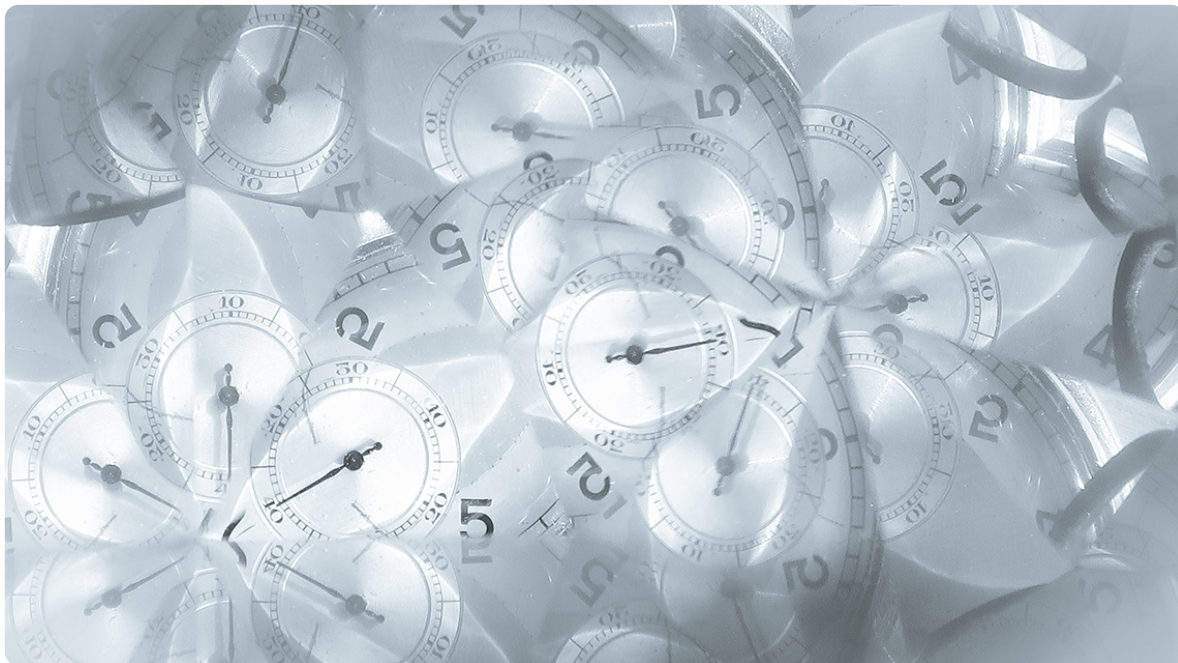


34 | 数组和集合，可变量的安全陷阱

范学雷 2019-03-22



00:00


09:05

讲述：刘飞 大小：8.32M

在前面的章节里，我们讨论了不少不可变量的好处。在代码安全中，不可变量也减少了很多纠葛的发生，可变量则是一个非常难缠的麻烦。

评审案例

我们一起看下这段 JavaScript 代码。


 复制代码

```
1 var mutableArray = [0, {  
2   toString : function() {  
3     mutableArray.length = 0;  
4   }  
5 }, 2];  
6  
7 console.log("Array before join(): ", mutableArray);  
8 mutableArray.join('');  
9 console.log("Array after join(): ", mutableArray);  
10
```

调用 `mutableArray.join()` 前后，你知道数组 `mutableArray` 的变化吗？调用 `join()` 前，数组 `mutableArray` 包含两个数字，一个函数（`{10, {}, 20}`）。调用 `join()` 后，数组 `mutableArray`

就变成一个空数组了。这其中的秘密就在于 join() 的实现，执行了数组中 toString() 函数。而 toString() 函数的实现，把数组 `mutableArray` 设置为空数组。

下面的代码，就是 JavaScript 引擎实现数组 join() 方法的一段内部 C 代码。

 复制代码

```
1 static JSBool
2 array_toString_sub(JSContext *cx, JSObject *obj, JSBool locale,
3                   JSString *sepstr, CallArgs &args) {
4     // snipped
5     size_t seplen;
6     // snipped
7     StringBuffer sb(cx);
8     if (!locale && !seplen && obj->isDenseArray() &&
9         !js_PrototypeHasIndexedProperties(cx, obj)) {
10        // Elements beyond the initialized length are
11        // 'undefined' and thus can be ignored.
12        const Value *beg = obj->getDenseArrayElements();
13        const Value *end =
14            beg + Min(length, obj->getDenseArrayInitializedLength());
15        for (const Value *vp = beg; vp != end; ++vp) {
16            if (!JS_CHECK_OPERATION_LIMIT(cx))
17                return false;
18
19            if (!vp->isMagic(JS_ARRAY_HOLE) &&
20                !vp->isNullOrUndefined()) {
21                if (!ValueToStringBuffer(cx, *vp, sb))
22                    return false;
23            }
24        }
25    }
26    // snipped
27 }
28
```

这段代码，把数组的起始地址记录在 beg 变量里，把数组的结束地址记录在 end 变量里。然后，从 beg 变量开始，通过调用 ValueToStringBuffer() 函数，把数组里的每一个变量，转换成字符串。

我们一起来看看第一段代码，是怎么在这段 join() 实现的 for 循环代码里执行的。

1. vp 指针初始化后，指向数组的起始地址；
2. 如果 vp 的地址不等于数组的结束地址 end，就把数组变量转换成字符串，然后变换 vp 指针到下一个地址。我们一起来看看这段代码是如何操作数组 mutableArray 的：
 - a. 数组的第一个变量是 0。0 被转换成字符，vp 指针换到下一个地址；
 - b. 数组的第二个变量是 toString() 函数。toString() 函数被调用后，就会把 mutableArray 这个数组设置为空数组，vp 指针换到下一个地址；
 - c. 数组的第三个变量本来应该是 2。但是，由于数组在上一步被置为空数组，数组的第三个变量的指针指向数组外地址。
3. 由于数组已经被设置为空数组，原数组的地址可能已经被其他数据占用，访问空数组外的地址就会造成内存泄漏或者程序崩溃。

通过设置第一段代码里的 `mutableArray` 和利用这个内存泄漏的漏洞，攻击者可以远程执行任意代码，获取敏感信息或者造成服务崩溃。这是一个[通用缺陷评分系统评分为 9.9](#)的严重安全缺陷。

案例分析


我们上面讨论的第一段代码里的 `mutableArray` 的构造方式，是一个典型的用于检查 JavaScript 引擎实现或者其他 JavaScript 数组使用缺陷的技术范例。

近十多年来，陆续发现了一些相似的 JavaScript 引擎数组实现的严重安全漏洞。几乎所有主流的 JavaScript 引擎提供商都受到了影响。我们太习惯使用数组的编码模式了，数组长度的变化，很难进入我们的考量范围。因此，查看或者编写这些实现代码，我们很难发现里面的漏洞，除非我们知道了这样的攻击模式。

如果一个新语言，支持类似 JavaScript 语言里这么灵活的函数数组变量，你可以试着找找这门编程语言实现里有没有类似的安全漏洞。

如果我们从根本上来看可变量，它的安全威胁就在于**在不同的时间、地点里，可变量可以发生变化。如果编写代码时，意识不到不同时空里的变化，就会面临安全威胁。**

我们再来看一下可变量的例子。在 Java 语言里，`java.util.Date` 是一个从 JDK 1.0 开始就支持的类。我们可以构建一个对象，来表示构建时的时间，然后再修改成其他时间。就像下面的这段伪代码这样。

 复制代码

```
1 public void verify(Date targetDate) {
2     // Verify that a contract is valid in the day of targetDate.
3
4     // <snipped>
5
6     // Display that the contract is valid in the day of targetDate
7 }
8
9 void checkContract() {
10     Date today = new Date();
11
12     // create a new thread that modify the date to a new date.
13     // For example, today.setYear(100) will reset the year to 2000.
14
15     // verify that the contract is valid today.
16     veify(today);
17 }
18
```


上面的代码中，`verify()` 方法和修改日期的线程间就存在竞争关系。如果日期修改在 `verify()` 实现的验证和显示之间发生，显示的日期就不是验证有效的日期。这就会给人错误的信息，从而导致错误的决策。这个问题就是 TOCTOU (time-of-check time-of-use) 竞态危害的一种常见形式。

可变量局部化

类似于 TOCTOU 的安全问题，让 `java.util.Date` 的使用充满了麻烦。

那么该怎么防范这种漏洞呢？当然，最有效的方法就是使用不可变量。对于可变量的参数，也可以使用拷贝等办法把共享的变量局部化。由于可变量可以在不同时空发生变化，所以，无论是传入参数，还是返回值，都要拷贝可变量。这样共享的变量，就转换成了局部变量。

比如上面的例子，我们就可以改成：

 复制代码

```
1 public void verify(Date targetDate) {
2     // Create a copy of the targetDate so as to avoid the
3     // impact of any changes.
4     Date inputDate = new Date(targetDate.getTime());
5
6     // Verify that a contract is valid in the day of inputDate.
7
8     // <snipped>
9
10    // Display that the contract is valid in the day of inputDate
11 }
12
13 void checkContract() {
14     Date today = new Date();
15
16     // create a new thread that modify the date to a new date.
17     // For example, today.setYear(100) will reset the year to 2000.
18
19     // verify that the contract is valid today.
20     // Use a clone of the today date so as to avoid the impact of
21     // any changes in the verify() implementation.
22     verify((Date)today.clone());
23 }
24
```

不知道你有没有注意到，在 `verify()` 的实现里，我们使用了 `Date` 的构造函数来做拷贝；而在 `checkContract()` 的实现里，我们使用了 `Date` 的 `clone()` 方法来做拷贝。为什么不都使用更简洁的 `clone()` 方法呢？

在 `checkContract()` 的实现里，`today` 变量是 `Date` 类的一个实例。我们都了解，`Date` 类的 `clone()` 方法的实现，的确做到了日期的拷贝。而对于作为参数传入 `verify()` 方法的 `targetDate` 对象，我们并不清楚它是不是一个可靠的 `Date` 的继承类。这个继承类的 `clone()` 实现，有没有做日期的拷贝，我们也不知晓，因此，`targetDate` 对象的 `clone()` 方法，不一定安全可靠。所以，在 `verify()` 实现里，使用 `clone()` 拷贝传入的参数，也不可靠。

类的继承还有很多麻烦的地方，后面的章节，我们还会接着讨论继承的安全缺陷。


支持实例拷贝

在一定的场景下，安全的编码需要通过拷贝把可变量局部化。这也就意味着，**我们设计一个可变类的时候，需要考虑支持实例的拷贝。要不然，这个类的使用，可能会遇到无法安全拷贝的**

麻烦。


实例的拷贝，可以使用静态的实例化方法，或者拷贝构造函数，或者使用公开的拷贝方法。需要注意的是，如果公开的拷贝方法可以被继承，继承类的实现方式就不可预料了。那么，这个公开的拷贝方法的使用就是不可靠的。支持公开的拷贝方法，一般只适用于 final 公开类。

静态的实例化：

 复制代码


```
1 public static MutableClass getInstance(MutableClass mutableObject) {  
2     // snipped  
3 }  
4
```

拷贝构造函数：

 复制代码


```
1 public MutableClass(MutableClass mutableObject) {  
2     // snipped  
3 }  
4
```

公开拷贝方法：

 复制代码

```
1 public final class MutableClass {  
2     // snipped  
3     @Override  
4     public Object clone() {  
5         // snipped  
6     }  
7 }  
8
```

禁用拷贝方法：

 复制代码

```
1 public class MutableClass {  
2     // snipped  
3     @Override  
4     public final Object clone() throws CloneNotSupportedException {  
5         throw new CloneNotSupportedException();  
6     }  
7 }  
8
```

实现拷贝，一般有两种方法。


一种是拷贝变量的指针或者引用，并不拷贝变量指向的内容。拷贝和原实例共享指针指向的内容，如果拷贝实例里的变量指向的内容发生了改变，原实例里的变量指向的内容也随着改变。这种拷贝方法通常称为浅拷贝（shallow copy）。

另外一种方式，是拷贝变量指向的内容。拷贝后的实例和原有的实例中，变量指向的内容虽然相同，但是相互独立的。一个实例里，变量指向的内容发生了改变，对另外一个实例没有影响。这种拷贝方法通常称为深拷贝（deep copy）。

对于一个类里的不可变量，一般我们使用浅拷贝就可以了。这也是不可变量的又一个优点。

下面的这段代码，就混合使用了可变量、不可变量，以及浅拷贝和深拷贝技术，来实现实例拷贝的一个示例。

拷贝

 复制代码

```
1 final class MyContract implements Cloneable {
2     private    String title;
3     private    Date signedDate;
4     private    Byte[] content;
5
6     // snipped
7
8     @Override
9     public Object clone() throws CloneNotSupportedException {
10         MyContract cloned = (MyContract)super.clone();
11
12         // shallow copy, String is immutable
13         cloned.title = this.title;
14
15         // deep copy, Date is mutable
16         cloned.signedDate = new Date(this.signedDate.getTime());
17
18         // deep copy, array are mutable
19         cloned.content = this.content.clone();
20
21         return cloned;
22     }
23 }
24
```

浅拷贝和原实例共享指针指向的内容，拷贝实例和原实例都可以改变指向的内容（除非内容为不可变量），这样就影响了对方的行为。所以，可变量的浅拷贝并不能解除安全隐患。

由于有两种拷贝方法，而且不同的拷贝方法适用的范围有一定的区别，我们就需要弄清楚一个类支持的是哪一种拷贝方法。特别是如果一个类使用的是浅拷贝，一定要在规范里标记清楚。要不然，就容易用错这个类的拷贝方法，从而导致安全风险。

如果一个类只提供了浅拷贝方法的实现，在使用可变量局部化解决安全隐患时，我们就会遇到很多麻烦。

麻烦的集合

出于效率的考虑，java.util 下的集合类，一般支持的是浅拷贝。比如 ArrayList 的 clone() 方法，执行的就是浅拷贝。如果使用深度拷贝，在很多场景下，集合的低下效率我们难以承受。对于类似于集合这样的类，可变量局部化就不是一个很好的解决方案。

对于集合来说，我们该怎么解决可变量的竞态危害这个问题呢？最主要的办法，就是不要把集合使用在可能产生竞态危害的场景中，我们后面再接着讨论这个问题。

小结

通过对这个案例的讨论，我想和你分享下面三点个人看法。

1. **可变量的传递，存在竞态危害的安全风险；**
2. **可变量局部化，是解决可变量竞态危害的一种常用办法；**
3. **变量的拷贝，有浅拷贝和深拷贝两种形式；可变量的浅拷贝无法解决竞态危害的威胁。**

对于这个案例，你还有什么别的看法吗？

一起来动手

数组是一个常见的难以处理的可变量。和集合一样，数组的拷贝也是有损效率的。什么时候，数组的传递需要拷贝？什么时候不需要拷贝？

不管是 C 语言，Java 还是 JavaScript，数组是一个我们编码经常使用的数据类型。你不妨检查一下你的代码，看看其中的数组使用是否存在我们上面讨论的安全问题。

欢迎你在留言区分享你的发现。

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。

© 版权归极客邦科技所有，未经许可不得转载



由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。

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。