

Java安全漫谈 - 01.反射篇(1)

这是[代码审计知识星球](#)中Java安全的第一篇文章。

Java安全可以从反序列化漏洞开始说起，反序列化漏洞又可以从反射开始说起。

反射是大多数语言里都必不可少的组成部分，对象可以通过反射获取他的类，类可以通过反射拿到所有方法（包括私有），拿到的方法可以调用，总之通过“反射”，我们可以将Java这种静态语言附加上动态特性。

这半年我老老提到“动态特性”，我给这四个字赋予了一个含义：“一段代码，改变其中的变量，将会导致这段代码产生功能性的变化，我称之为动态特性”。我将在今年的KCON上给大家分享关于PHP中的动态特性的议题，这里就不展开说了。

PHP本身拥有很多动态特性，所以可以通过“一句话木马”来执行各种功能；Java虽不像PHP那么灵活，但其提供的“反射”功能，也是可以提供一些动态特性。比如，这样一段代码，在你不知道传入的参数值的时候，你是不知道他的作用是什么的：

```
1 public void execute(String className, String methodName) throws Exception {
2     Class clazz = Class.forName(className);
3     clazz.getMethod(methodName).invoke(clazz.newInstance());
4 }
```

上面的例子中，我演示了几个在反射里极为重要的方法：

- 获取类的方法：`forName`
- 实例化类对象的方法：`newInstance`
- 获取函数的方法：`getMethod`
- 执行函数的方法：`invoke`

基本上，这几个方法包揽了Java安全里各种和反射有关的Payload。

`forName` 不是获取“类”的唯一途径，通常来说我们有如下三种方式获取一个“类”，也就是 `java.lang.Class` 对象：

- `obj.getClass()` 如果上下文中存在某个类的实例 `obj`，那么我们可以直接通过 `obj.getClass()` 来获取它的类
- `Test.class` 如果你已经加载了某个类，只是想获取到它的 `java.lang.Class` 对象，那么就直奔它的 `class` 属性即可。这个方法其实不属于反射。
- `Class.forName` 如果你知道某个类的名字，想获取到这个类，就可以使用 `forName` 来获取

在安全研究中，我们使用反射的一大目的，就是绕过某些沙盒。比如，上下文中如果只有Integer类型的数字，我们如何获取到可以执行命令的Runtime类呢？也许可以这样（伪代

码）：`1.getClass().forName("java.lang.Runtime")` 这里写的应该有问题，`getClass`已经获取到了Class

关于绕沙盒，之前Code-Breaking 2018我出了一道SpEL的题目，分享一篇第三方Writeup：<http://rui0.cn/archives/1015>

forName有两个函数重载：

- `Class<?> forName(String name)`
- `Class<?> forName(String name, **boolean** initialize, ClassLoader loader)`

第一个就是我们最常见的获取class的方式，其实可以理解为第二种方式的一个封装：

```
1 Class.forName(className)
2 // 等于
3 Class.forName(className, true, currentLoader)
```

默认情况下，`forName` 的第一个参数是类名；第二个参数表示是否初始化；第三个参数就是 `ClassLoader`。

`ClassLoader` 是什么呢？它就是一个“加载器”，告诉Java虚拟机如何加载这个类。关于这个点，后面还有很多有趣的漏洞利用方法，这里先不展开说了。Java默认的 `ClassLoader` 就是根据类名来加载类，这个类名是类完整路径，如 `java.lang.Runtime`。

第二个参数 `initialize` 常常被人误解，我看到勾陈安全实验室有篇讲反射机制的文章（<http://www.polaris-lab.com/index.php/archives/450/>）里说到：

注意，有一点很有趣，使用功能“`Class`”来创建Class对象的引用时，不会自动初始化该Class对象，使用`forName()`会自动初始化该Class对象

The screenshot shows an IDE with a Java file named `reflectionTest.java`. The code includes a `main` method that uses `Class.forName("test.class")` to load the `test` class. A red box highlights `test.class` in the code, and a red arrow points from it to the file explorer. The file explorer shows a directory named `Step2` containing two files: `reflectionTest.class` and `test.class`. A red arrow points from the `test.class` file in the explorer to the `test` class definition in the code. A text box in the center of the image states: "代码中test.class对象的本质就是test.class文件加载到内存中的内容".

图中有说“构造函数，初始化时执行”，其实在 `forName` 的时候，构造函数并不会执行，即使我们设置 `initialize=true`。

那么这个初始化究竟指什么呢？

可以将这个“初始化”理解为类的初始化。我们先来看看如下这个类：

```

1 public class TrainPrint {
2     {
3         System.out.printf("Empty block initial %s\n", this.getClass());
4     }
5
6     static {
7         System.out.printf("Static initial %s\n", TrainPrint.class);
8     }
9
10    public TrainPrint() {
11        System.out.printf("Initial %s\n", this.getClass());
12    }
13 }

```

我刚才在[知识星球](#)里问过一个问题：上述的三个“初始化”方法有什么区别，调用顺序是什么，在安全上有什么价值？

其实你运行一下就知道了，首先调用的是 `static {}`，其次是 `{}`，最后是构造函数。

其中，`static {}`就是在“类初始化”的时候调用的，而 `{}` 中的代码会放在构造函数的 `super()` 后面，但在当前构造函数内容的前面。

所以说，`forName` 中的 `initialize=true` 其实就是告诉Java虚拟机是否执行“类初始化”。

那么，假设我们有如下函数，其中函数的参数name可控：

```

1 public void ref(String name) throws Exception {
2     Class.forName(name);
3 }

```

我们就可以编写一个恶意类，将恶意代码放置在 `static {}` 中，从而执行：

```

1 import java.lang.Runtime;
2 import java.lang.Process;
3
4 public class TouchFile {
5     static {
6         try {
7             Runtime rt = Runtime.getRuntime();
8             String[] commands = {"touch", "/tmp/success"};
9             Process pc = rt.exec(commands);
10            pc.waitFor();
11        } catch (Exception e) {
12            // do nothing
13        }
14    }
15 }

```

当然，这个恶意类如何带入目标机器中，可能就涉及到ClassLoader的一些利用方法了，本文暂不做探讨。