

Java安全漫谈 - 01.反射篇(2)

这是[代码审计知识星球](#)中Java安全的第二篇文章。

继续说一下反射，上次主要讲了forName。

在正常情况下，除了系统类，如果我们想拿到一个类，需要先 `import` 才能使用。而使用forName就不需要，这样对于我们的攻击者来说就十分有利，我们可以加载任意类。

另外，我们经常在一些源码里看到，类名的部分包含 `$` 符号，比如fastjson在 `checkAutoType` 时候就会先将 `$` 替换成 `.`：<https://github.com/alibaba/fastjson/blob/fcc9c2a/src/main/java/com/alibaba/fastjson/parser/ParserConfig.java#L1038>。`$` 的作用是查找内部类。

Java的普通类 `C1` 中支持编写内部类 `C2`，而在编译的时候，会生成两个文件：`C1.class` 和 `C1$C2.class`，我们可以把他们看作两个无关的类，通过 `Class.forName("C1$C2")` 即可加载这个内部类。

获得类以后，我们可以继续使用反射来获取这个类中的属性、方法，也可以实例化这个类，并调用方法。

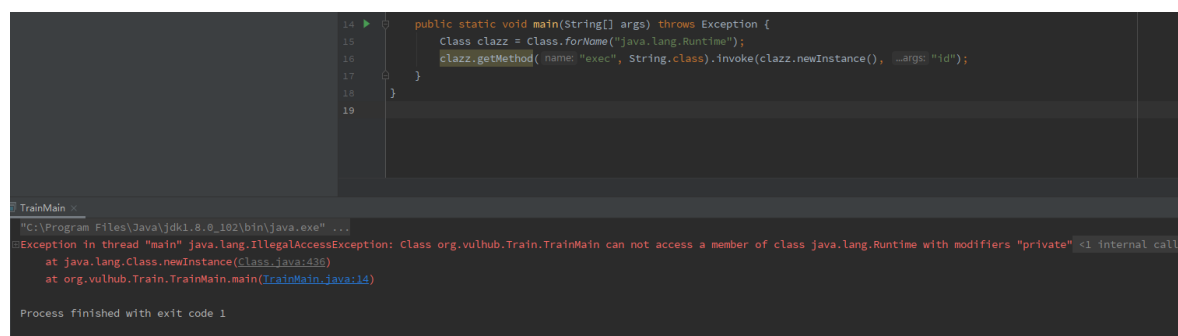
`Class.newInstance()` 的作用就是调用这个类的无参构造函数，这个比较好理解。不过，我们有时候在写漏洞利用方法的时候，会发现使用 `newInstance` 总是不成功，这时候原因可能是：

1. 你使用的类没有无参构造函数
2. 你使用的类构造函数是私有的

最最最常见的情况就是 `java.lang.Runtime`，这个类在我们构造命令执行Payload的时候很常见，但我们不能直接这样来执行命令：

```
1 Class clazz = Class.forName("java.lang.Runtime");
2 clazz.getMethod("exec", String.class).invoke(clazz.newInstance(), "id");
```

你会得到这样一个错误：



```
14 public static void main(String[] args) throws Exception {
15     Class clazz = Class.forName("java.lang.Runtime");
16     clazz.getMethod("exec", String.class).invoke(clazz.newInstance(), "id");
17 }
18
19
```

```
Exception in thread "main" java.lang.IllegalAccessException: Class org.vulhub.Train.TrainMain can not access a member of class java.lang.Runtime with modifiers "private" <1 internal call>
at java.lang.Class.newInstance(Class.java:436)
at org.vulhub.Train.TrainMain.main(TrainMain.java:14)

Process finished with exit code 1
```

原因是 `Runtime` 类的构造方法是私有的。

有同学就比较好奇，为什么会有类的构造方法是私有的，难道他不想让用户使用这个类吗？这其实涉及到很常见的设计模式：“单例模式”。（有时候工厂模式也会写成类似）

比如，对于Web应用来说，数据库连接只需要建立一次，而不是每次用到数据库的时候再新建一个连接，此时作为开发者你就可以将数据库连接使用的类的构造函数设置为私有，然后编写一个静态方法来获取：

```

1 public class TrainDB {
2     private static TrainDB instance = new TrainDB();
3
4     public static TrainDB getInstance() {
5         return instance;
6     }
7
8     private TrainDB() {
9         // 建立连接的代码...
10    }
11 }

```

这样，只有类初始化的时候会执行一次构造函数，后面只能通过 `getInstance` 获取这个对象，避免建立多个数据库连接。

回到正题，`Runtime`类就是单例模式，我们只能通过 `Runtime.getRuntime()` 来获取到 `Runtime` 对象。我们将上述Payload进行修改即可正常执行命令了：

```

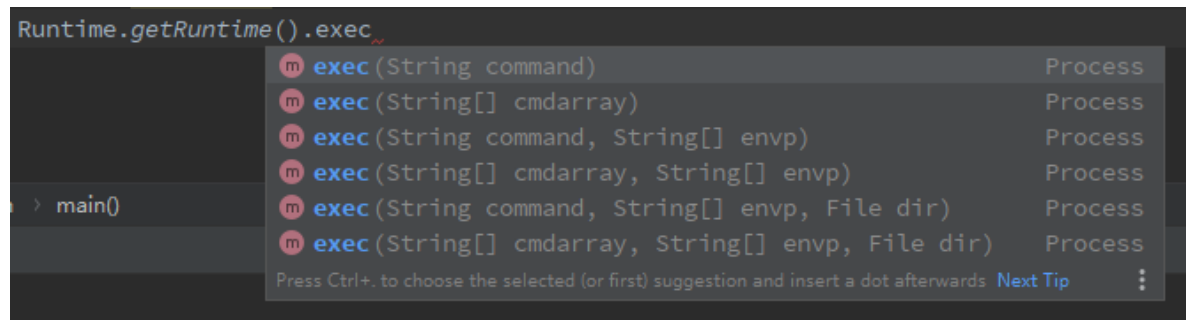
1 Class clazz = Class.forName("java.lang.Runtime");
2 clazz.getMethod("exec",
String.class).invoke(clazz.getMethod("getRuntime").invoke(clazz),
"calc.exe");

```

这里用到了 `getMethod` 和 `invoke` 方法。

`getMethod` 的作用是通过反射获取一个类的某个特定的公有方法。而学过Java的同学应该清楚，Java中支持类的重载，我们不能仅通过函数名来确定一个函数。所以，在调用 `getMethod` 的时候，我们需要传给他你需要获取的函数的参数类型列表。

比如这里的 `Runtime.exec` 方法有6个重载：



我们使用最简单的，也就是第一个，它只有一个参数，类型是String，所以我们使用 `getMethod("exec", String.class)` 来获取 `Runtime.exec` 方法。

`invoke` 的作用是执行方法，它的第一个参数是：

- 如果这个方法是一个普通方法，那么第一个参数是类对象
- 如果这个方法是一个静态方法，那么第一个参数是类

这也比较好理解了，我们正常执行方法是 `[1].method([2], [3], [4]...)`，其实在反射里就是 `method.invoke([1], [2], [3], [4]...)`。

所以我们将上述命令执行的Payload分解一下就是：

```
1 | Class clazz = Class.forName("java.lang.Runtime");
2 | Method execMethod = clazz.getMethod("exec", String.class);
3 | Method getRuntimeMethod = clazz.getMethod("getRuntime");
4 | Object runtime = getRuntimeMethod.invoke(clazz);
5 | execMethod.invoke(runtime, "calc.exe");
```

这个就比较好看懂了。

不过，看完今天这篇文章的同学应该脑子里会存在两个疑问：

- 如果一个类没有无参构造方法，也没有类似单例模式里的静态方法，我们怎样通过反射实例化该类呢？
- 如果一个方法或构造方法是私有方法，我们是否能执行它呢？