## Java

### 命令执行

- 无回显：

```
import java.io.IOException;
public class Test1 {
      public static void main(String[] args) throws IOException {
            Runtime.getRuntime().exec("calc");
      }
}
```

- 需要回显怎么办的问题，主要是用IO流将命令执行后的字节加载出来，然后最基本的按行读取，就可以了
- 有回显：

```
import java.io.*;

public class Testa {
      public static void main(String[] args) throws IOException {
            Process process = Runtime.getRuntime().exec("ping baidu.com");
            InputStream inputStream = process.getInputStream();
            InputStreamReader inputStreamReader =   new InputStreamReader(inputStream);
            BufferedReader inputBufferedReader = new BufferedReader(inputStreamReader);
            StringBuilder stringBuilder=new StringBuilder();
            String line = null;
            while ((line = inputBufferedReader.readLine()) != null) {
                  stringBuilder.append(line);
                  System.out.println(line);
            }
            inputBufferedReader.close();
            inputBufferedReader=null;
            inputStreamReader.close();
            inputStreamReader=null;
            inputStream.close();
            inputStream=null;
      }
}
```

- Windows下调用 cmd或者powershell去执行命令，但是powershell一般会限制执行策略，所以使用cmd一般是比较保险的：

```
String [] cmd={"cmd","/C","calc.exe"};
Process proc =Runtime.getRuntime().exec(cmd);
```

- linux一般使用`bash`进行命令执行，有的情况，可能没有bash，可以使用`sh`来进行替代

```
String [] cmd={"/bin/sh","-c","ls"};
Process proc =Runtime.getRuntime().exec(cmd);
```

- 通过`System.getProperty("os.name");`获取系统名称

```
import java.io.*;

public class Test {
      public static void main(String[] args) throws IOException {
            String property = System.getProperty("os.name");
            String [] cmd1={"cmd","/C","dir"};
            String [] cmd2={"/bin/sh","-c","ls"};
            String [] cmd = null;
            System.out.println(property);
            if (property.contains("Windows")){
                  cmd= cmd1;
            }
            else {
                  cmd= cmd2;
            }

            Process process =Runtime.getRuntime().exec(cmd);
            //取得命令结果的输出流
            InputStream inputStream = process.getInputStream();
            //用输出读取去读
            InputStreamReader inputStreamReader =   new InputStreamReader(inputStream);
            //创建缓冲器
            BufferedReader inputBufferedReader = new BufferedReader(inputStreamReader);
            StringBuilder stringBuilder=new StringBuilder();
            String line = null;
            while ((line = inputBufferedReader.readLine()) != null) {
                  stringBuilder.append(line);
                  System.out.println(line);
            }
            inputBufferedReader.close();
```

```
                inputBufferedReader=null;
                inputStreamReader.close();
                inputStreamReader=null;
                inputStream.close();
                inputStream=null;
        }
}
```

### 反射

- Java的反射（reflection）机制是指在程序的运行状态中，可以构造任意一个类的对象，可以了解任意一个对象所属的类，可以了解任意一个类的成员变量和方法，可以调用任意一个对象的属性和方法，本质上其实就是动态的生成字节码，加载到jvm中运行
- 由于java语言动态的特性，在程序运行后，所运行的类，就已经在JVM的内存中，就可以直接调用已经加载好的类去实现方法操作
- Java反射机制的核心是在程序运行时动态加载类并获取类的详细信息，从而操作类或对象的属性和方法，本质是JVM得到class对象之后，再通过class对象进行反编译，从而获取对象的各种信息
- Java属于先编译再运行的语言，程序中对象的类型在编译期就确定下来了，而当程序在运行时可能需要动态加载某些类，这些类因为之前用不到，所以没有被加载到JVM，通过反射，可以在运行时动态地创建对象并调用其属性，不需要提前在编译期知道运行的对象是谁
- 反射调用方法时，会忽略权限检查，可以无视权限修改对应的值，因此容易导致安全性问题
- 反射举例——加载jdbc驱动

```
public class Connect {
    1 个用法
    static final String JDBC_DRIVER = "com.mysql.cj.jdbc.Driver";
    1 个用法
    static final String DB_URL = "jdbc:mysql://localhost:3306/test";
    1 个用法
    static final String USER = "root";
    1 个用法
    static final String PASS = "123456";
    4 个用法
    Connection connection;
    2 个用法
    Statement statement;
    3 个用法
    Connect(){

        try {
            Class.forName(JDBC_DRIVER);
            connection=DriverManager.getConnection(DB_URL,USER,PASS);
            statement=connection.createStatement();
        }catch (Exception e){
            System.out.println(e.getMessage());
        }

    }
}
```

- 通过反射获取`Runtime`类

```
import java.io.IOException;
import java.lang.Class;
public class Test1 {
      public static void main(String[] args) throws IOException {
            try {
                  String className = "java.lang.Runtime";
                  Class<?> runtimeClass1 = Class.forName(className);
                  Class<?>   runtimeClass2 = java.lang.Runtime.class;
                  Class<?>   runtimeClass3 =
ClassLoader.getSystemClassLoader().loadClass(className);
            } catch (ClassNotFoundException e) {
                  throw new RuntimeException(e);
            }
      }
}
```

- `getDeclaredConstructor()`可以获得构造方法，也就是我们常用的`private`方法，其中Runtime的构造方法是`private`，我们无法直接调用，我们需要使用反射去修改方法的访问权限（使用`setAccessible`，修改为 true）

```
Constructor constructor = runtimeClass1.getDeclaredConstructor();
constructor.setAccessible(true);
```

- 通过获取的构造器进行实例化对象

```
 Object runtimeInstance = constructor.newInstance(); //这里的话就等价于 Runtime rt = new Runtime();
```

- 获取方法

```
Method runtimeMethod = runtimeClass1.getMethod("exec", String.class);
```

- `getMethod`的作用是通过反射获取一个类的某个特定的公有方法
- 当我们想获取当前类的所有成员方法时们可以使用：`Method[] methods = class.getDeclaredMethods();`

  获取当前类指定的成员方法时：`Method method = class.getDeclaredMethod("方法名");`、`Method method = class.getDeclaredMethod("方法名", 参数类型如String.class，多个参数用","号隔开);`
- 执行方法：

```
Process process = (Process) runtimeMethod.invoke(runtimeInstance, "calc");
//method.invoke(方法实例对象，方法参数值，多个参数值用","隔开);
```

- invoke就是调用类中的方法，最简单的用法是可以把方法参数化`invoke(class, method)`，还可以把方法名存进数组`v[]`，然后循环里`invoke(test,v[i])`，就顺序调用了全部方法
- `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
- 获取成员变量

```
//获取类中的成员们变量
Field fields = class.getDeclaredFields();
//获取当前类指定的成员变量
Field field   = class.getDeclaredField("变量名");
//获取成员变量的值
Object obj = field.get(类实例对象);
//修改成员变量的值
field.set(类实例对象, 修改后的值);
```

- 完整代码：

```
import java.io.IOException;
import java.lang.Class;
import java.lang.reflect.Constructor;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;

public class Test1 {
      public static void main(String[] args) throws ClassNotFoundException,
NoSuchMethodException, IllegalAccessException, InvocationTargetException,
InstantiationException, IOException {
            String name = System.getProperty("os.name");
            System.out.println(name);

            String className = "java.lang.Runtime";
            Class<?> runtimeClass1 = Class.forName(className);
            System.out.println(runtimeClass1 + "\n");
            Class<?>   runtimeClass2 = java.lang.Runtime.class;
            System.out.println(runtimeClass2 + "\n");
            Class<?>   runtimeClass3 =
ClassLoader.getSystemClassLoader().loadClass(className);
            System.out.println(runtimeClass3 + "\n");

            Constructor<?> constructor;
            constructor = runtimeClass1.getDeclaredConstructor();
            System.out.println(constructor);
            constructor.setAccessible(true);

            Object runtimeInstance = constructor.newInstance(); //这里的话就等价于
Runtime rt = new Runtime();
            System.out.println(runtimeInstance);
            Method runtimeMethod = runtimeClass2.getMethod("exec", String.class);
            System.out.println(runtimeMethod);

            Process process = (Process) runtimeMethod.invoke(runtimeInstance, "calc");
      }
}
```

### 反射补充篇

- forName有两个函数重载：
  - `Class<?> forName(String name)`
  - `Class<?> forName(String name, **boolean** initialize, ClassLoader loader)`

- 第一个就是最常见的获取class的方式，可以理解为第二种方式的一个封装：

```
Class.forName(className)
// 等于
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`是一个“加载器”，告诉Java虚拟机如何加载这个类，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`，在`forName`的时候，构造函数并不会执行，即使设置`initialize=true`
- 关于类初始化调用顺序：
  - 代码：

```
package org.example;

public class Main {
     {
          System.out.printf("Empty block initial %s\n", this.getClass());
     }
      static {
          System.out.printf("Static initial %s\n", Main.class);
     }
      public Main() {
          System.out.printf("Initial %s\n", this.getClass());
     }

      public static void main(String[] args) {
          Main main=new Main();
     }
}
```

  - 运行后可以看到调用顺序为：`static->{}->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“
- `class.newInstance()`的作用就是调用这个类的无参构造函数，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：
  - 使用的类没有无参构造函数

- 使用的类构造函数是私有的

```
Class clazz = Class.forName("java.lang.Runtime");
clazz.getMethod("exec", String.class).invoke(clazz.getMethod("getRuntime").invoke(clazz),
"calc.exe");
//等于
Class clazz = Class.forName("java.lang.Runtime");
Method execMethod = clazz.getMethod("exec", String.class);
Method getRuntimeMethod = clazz.getMethod("getRuntime");
Object runtime = getRuntimeMethod.invoke(clazz);
execMethod.invoke(runtime, "calc.exe");
```

- 在一些源码里可以看到，类名的部分包含`$`符号，`$`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$C2.class`，可以把他们看作两个无关的类，通过`Class.forName("C1$C2")`即可加载这个内部类
- 获得类以后，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
- 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，可以使用`getConstructor`方法，通过反射实例化该类
- 和`getMethod`类似，`getConstructor`接收的参数是构造函数列表类型，因为构造函数也支持重载，所以必须用参数列表类型才能唯一确定一个构造函数
- 获取到构造函数后，使用`newInstance`来执行
- 比如，常用的另一种执行命令的方式`ProcessBuilder`，使用反射来获取其构造函数，然后调用`start()`来执行命令：

```
Class clazz = Class.forName("java.lang.ProcessBuilder");
((ProcessBuilder)clazz.getConstructor(List.class).newInstance(Arrays.asList("calc.exe")))
.start();
```

- ProcessBuilder有两个构造函数：
  - `public ProcessBuilder(List<String> command)`
  - `public ProcessBuilder(String... command)`
- 上面用到了第一个形式的构造函数，所以在`getConstructor`的时候传入的是`List.class`，但是前面这个Payload用到了Java里的强制类型转换，有时候利用漏洞的时候（在表达式上下文中）是没有这种语法的，所以，我们仍需利用反射来完成这一步：

```
Class clazz = Class.forName("java.lang.ProcessBuilder");
clazz.getMethod("start").invoke(clazz.getConstructor(List.class).newInstance(Arrays.asLis
t("calc.exe")));
```

- 通过`getMethod("start")`获取到`start`方法，然后`invoke`执行，`invoke`的第一个参数就是`ProcessBuilder Object`了
- 如果要使用`public ProcessBuilder(String... command)`这个构造函数，就涉及到Java里的可变长参数（varargs）了
- Java支持可变长参数，就是当定义函数的时候不确定参数数量的时候，可以使用`...`这样的语法来表示这个函数的参数个数是可变
  的
- 对于可变长参数，Java其实在编译的时候会编译成一个数组，也就是说，如下这两种写法在底层是等价的（也就不能重载）：

```
public void hello(String[] names) {}
public void hello(String...names) {}
```

- 对于反射来说，如果要获取的目标函数里包含可变长参数，其实认为它是数组就行了，将字符串数组的类`String[].class`传给`getConstructor`，获取`ProcessBuilder`的第二种构造函数：

```
Class clazz = Class.forName("java.lang.ProcessBuilder");
clazz.getConstructor(String[].class)
```

- 在调用`newInstance`的时候，因为这个函数本身接收的是一个可变长参数，传给`ProcessBuilder`的也是一个可变长参数，二者叠加为一个二维数组，所以整个Payload如下：

```
Class clazz = Class.forName("java.lang.ProcessBuilder");
((ProcessBuilder)clazz.getConstructor(String[].class).newInstance(new String[][]
{{"calc.exe"}})).start();
```

- 改为反射的写法：

```
Class clazz1 = Class.forName("java.lang.ProcessBuilder");
clazz1.getMethod("start").invoke(clazz1.getConstructor(String[].class).newInstance(new
String[][]{{"calc.exe"}}));
```

- 如果一个方法或构造方法是私有方法，就涉及到`getDeclared`系列的反射了，与普通的`getMethod`、`getConstructor`区别是：

    - `getMethod`系列方法获取的是当前类中所有公共方法，包括从父类继承的方法
    - `getDeclaredMethod`系列方法获取的是当前类中“声明”的方法，是实在写在这个类里的，包括私有的方法，但从父类里继承来的就不包含了
- `getDeclaredMethod`的具体用法和`getMethod`类似，`getDeclaredConstructor`的具体用法和`getConstructor`类似
- `Runtime`这个类的构造函数是私有的，需要用`Runtime.getRuntime()`来获取对象，也可以直接用`getDeclaredConstructor`来获取这个私有的构造方法来实例化对象，进而执行命令：

```
Class clazz = Class.forName("java.lang.Runtime");
Constructor m = clazz.getDeclaredConstructor();
m.setAccessible(true);
clazz.getMethod("exec", String.class).invoke(m.newInstance(), "calc.exe");
```

- 这里使用了一个方法`setAccessible`，这个是必须的，在获取到一个私有方法后，必须用`setAccessible`修改它的作用域，否则仍然不能调用

### 序列化和反序列化

- 序列化条件：
    - 该类必须实现`java.io.Serlalizable`或`java.io.Externalizable`接口
    - 该类的所有属性必须是可序列化的，如果一个属性是不可序列化的，则属性必须标明是短暂的
    - 当前类提供一个全局常量 serialVersionUID
    - ObjectInputStream和ObjectOutputStream不能序列化static和transient修饰的成员变量
- 序列化：`ObjectOutputStream类 -> writeObject()`——该方法对参数指定的obj文件进行序列化，把字节序列写到一个目标输出流中，按照java标准是 给文件一个 ser 的扩展名
- 反序列化：`ObjectInputStream类-> readObject()`——该方法是从一个输入流中读取字节序列，再把它们反序列化成对象，将其返回
- Java反序列化时会执行`readObject()`方法，所以如果readObject()方法被恶意构造的话，就有可能导致命令执行
- 反序列化漏洞成因：
    - 序列化指把Java对象转换为字节序列的过程，反序列化就是打开字节流并重构对象，那如果即将被反序列化的数据是特殊构造的，就可以产生非预期的对象，从而导致任意代码执行
    - Java中间件通常通过网络接收客户端发送的序列化数据，而在服务端对序列化数据进行反序列化时，会调用被序列化对象的`readObject( )`方法，而在Java中如果重写了某个类的方法，就会优先调用经过修改后的方法，如果某个对象重写了`readObject( )`方法，且在方法中能够执行任意代码，那服务端在进行反序列时，也会执行相应代码

- 如果能够找到满足上述条件的对象进行序列化并发送给Java中间件，Java中间件也会去执行指定的代码，即存在反序列化漏洞。

### 反序列化补充篇

- Java在序列化时一个对象，将会调用这个对象中的`writeObject`方法，参数类型是`ObjectOutputStream`，开发者可以将任何内容写入这个stream中
- 反序列化时，会调用`readObject`，开发者也可以从中读取出前面写入的内容，并进行处理
- 例子：

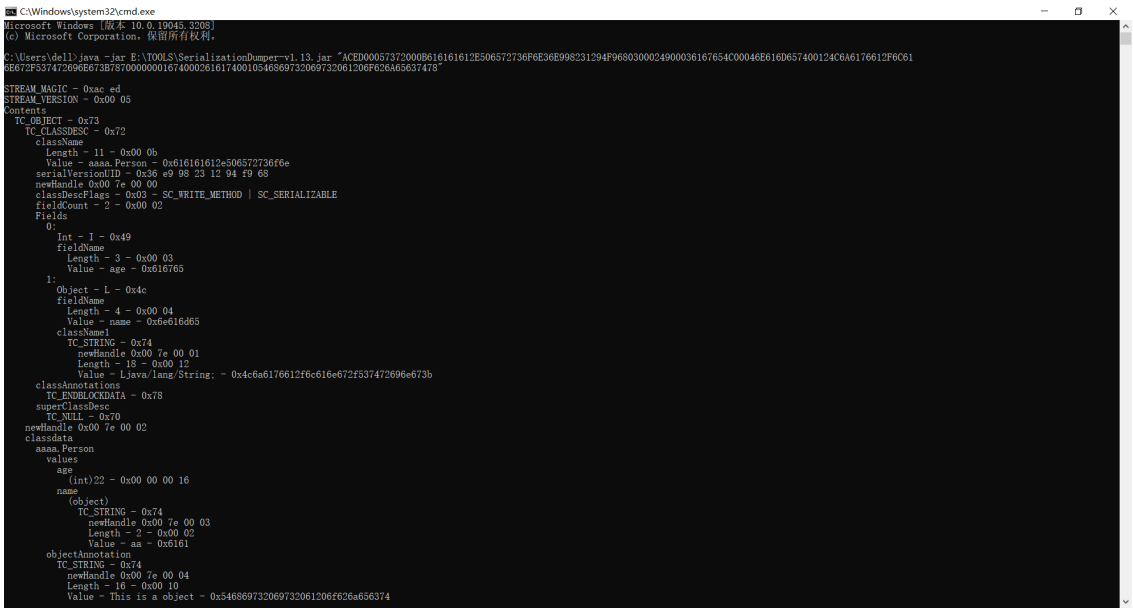
```
package aaaa;

import java.io.IOException;
public class Person implements java.io.Serializable {
      public String name;
      public int age;
      Person(String name, int age) {
            this.name = name;
            this.age = age;
      }
      private void writeObject(java.io.ObjectOutputStream s) throws
                  IOException {
            s.defaultWriteObject();
            s.writeObject("This is a object");
      }
      private void readObject(java.io.ObjectInputStream s)
                  throws IOException, ClassNotFoundException {
            s.defaultReadObject();
            String message = (String) s.readObject();
            System.out.println(message);
      }

      public static void main(String[] args) {

      }
}
```

- 在执行完默认的`s.defaultWriteObject()`后，我向stream里写入了一个字符串`This is a object`
- 对person类进行序列化：

```
package aaaa;

import java.io.FileOutputStream;
```

```
import java.io.IOException;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

public class SerializationTest {
      public static void serialize(Object obj) throws IOException{
            ObjectOutputStream oos = new ObjectOutputStream(new
FileOutputStream("./ser.bin"));
            oos.writeObject(obj);
      }

      public static void main(String[] args) throws Exception{
            Person person = new Person("aa",22);
            System.out.println(person);
            serialize(person);
      }
}
```

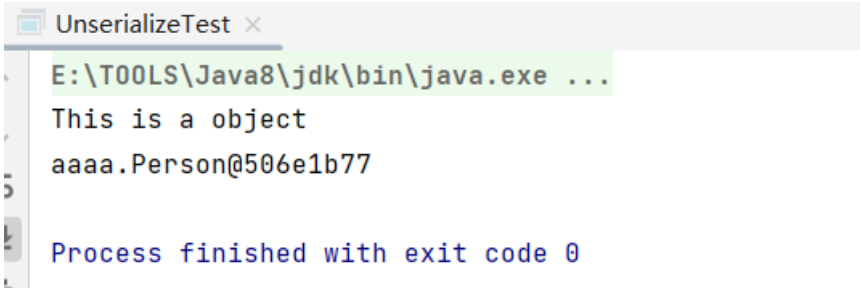
- 使用`SerializationDumper`对序列化数据进行分析：




- 字符串`This is a object`被放在`objectAnnotation`的位置，在反序列化时，对这个字符串进行读取并输出：

```
package aaaa;

import java.io.FileInputStream;
import java.io.IOException;
import java.io.ObjectInputStream;

public class UnserializeTest {
```

```
    public static Object unserialize(String Filename) throws IOException,
ClassNotFoundException{
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream(Filename));
        Object obj = ois.readObject();
        return obj;
    }

    public static void main(String[] args) throws Exception{
        Person person = (Person)unserialize("./ser.bin");
        System.out.println(person);
    }
}
```



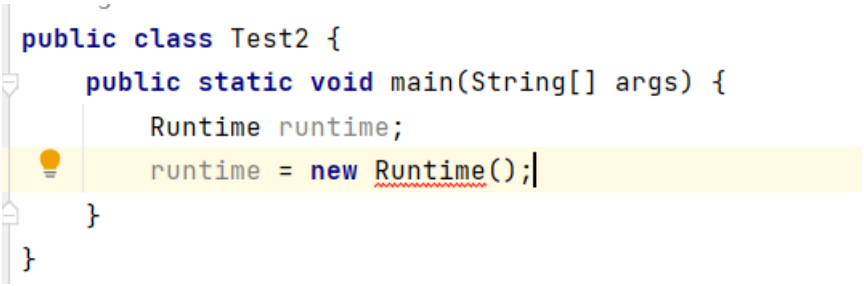

### serialVersionUID(序列号)

- 如果我们没有自定义序列化id，当我们修改User 类的时候，编译器又为我们User 类生成了一个UID，而序列化和反序列化就是通过对比其SerialversionUID来进行的，一旦SerialversionUID不匹配，反序列化就无法成功，可以自己去指定serialVersionUID
- 设置序列化ID：序列化运行时将一个版本号与每个称为SerialVersionUID的可序列化类相关联，在反序列化过程中，使用该序列号验证序列化对象的发送方和接收方是否为该对象加载了与序列化兼容的类，如果接收方为对象加载的类的UID与相应发送方类的UID不同，则反序列化将导致InvalidClassException. 可序列化类可以通过声明字段名来显式声明自己的UID，它必须是static、final和long类型，如：

```
(public/private/protected/default) static final long serialVersionUID=42L;   //建议使用
private修饰符
```

### 注意

- 不能new 一个Runtime类



- Runtime是一个单例类，单例类是不能够进行new的
- 单例模式（Singleton Pattern）是 Java 中最简单的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。这种模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。
- https://www.runoob.com/design-pattern/singleton-pattern.html

### ysoserial

- 发现有很多小模块的东西要记，又不想单独在写一篇文章，索性都堆到这一篇里了哈哈哈
- `ysoserial`可以让用户根据自己选择的利用链，生成反序列化利用数据，通过将这些数据发送给目标，从而执行用户预先定义的命令
- 使用：

```
java -jar ysoserial-master-30099844c6-1.jar CommonsCollections1 "id"
```

- 生成好的POC发送给目标，如果目标存在反序列化漏洞，并满足这个gadget对应的条件，则命令`id`将被执行