

Java安全漫谈 - 06.RMI篇(3)

这是[代码审计知识星球](#)中Java安全的第六篇文章。

上一篇我们详细说了如何利用codebase来加载远程类，在RMI服务端执行任意代码。那么，从原理上来讲，codebase究竟是如何传递进而被利用的呢？

我们曾在第4篇文章抓过RMI的数据包，当时通过数据包简单梳理了RMI通信的组成部分与过程。这次我们尝试抓取了上一篇文章中攻击RMI的数据包，当然也有2个TCP连接：

- 1. 本机与RMI Registry的通信（在我的数据包中是1099端口）
- 2. 本机与RMI Server的通信（在我的数据包中是64000端口）

我们用 tcp.stream eq 0 来筛选出本机与RMI Registry的数据流：

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.195	140.238.34.216	TCP	66	27982 → 1099 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
2	0.169310	140.238.34.216	192.168.1.195	TCP	66	1099 → 27982 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
3	0.169410	192.168.1.195	140.238.34.216	TCP	54	27982 → 1099 [ACK] Seq=1 Ack=1 Win=131328 Len=0
4	0.170324	192.168.1.195	140.238.34.216	RMI	61	JRMI, Version: 2, StreamProtocol
5	0.338060	140.238.34.216	192.168.1.195	TCP	54	1099 → 27982 [ACK] Seq=1 Ack=8 Win=29312 Len=0
6	0.338836	140.238.34.216	192.168.1.195	RMI	75	JRMI, ProtocolAck
7	0.339075	192.168.1.195	140.238.34.216	RMI	74	Continuation
8	0.344141	192.168.1.195	140.238.34.216	RMI	104	JRMI, Call
9	0.512504	140.238.34.216	192.168.1.195	TCP	54	1099 → 27982 [ACK] Seq=22 Ack=78 Win=29312 Len=0
10	0.515825	140.238.34.216	192.168.1.195	RMI	357	JRMI, ReturnData
11	0.556590	192.168.1.195	140.238.34.216	TCP	54	27982 → 1099 [ACK] Seq=78 Ack=325 Win=131072 Len=0
22	1.110576	192.168.1.195	140.238.34.216	RMI	55	JRMI, Ping
24	1.278943	140.238.34.216	192.168.1.195	RMI	55	JRMI, PingAck
25	1.279153	192.168.1.195	140.238.34.216	RMI	69	JRMI, DgcAck
29	1.488061	140.238.34.216	192.168.1.195	TCP	54	1099 → 27982 [ACK] Seq=326 Ack=94 Win=29312 Len=0
37	1.738609	192.168.1.195	140.238.34.216	TCP	54	27982 → 1099 [RST, ACK] Seq=94 Ack=326 Win=0 Len=0

可见，在与RMI Registry通信的时候Wireshark识别出了协议类型。我们选择其中序号是8的数据包，然后复制Wireshark识别出的 Java Serialization 数据段：

tcp.stream eq 0

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.195	140.238.34.216	TCP	66	27982 → 1099 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
2	0.169310	140.238.34.216	192.168.1.195	TCP	66	1099 → 27982 [SYN, ACK] Seq=0 Ack=1 Win=29200 Len=0 MSS=1460 SACK_PERM=1 WS=128
3	0.169410	192.168.1.195	140.238.34.216	TCP	54	27982 → 1099 [ACK] Seq=1 Ack=1 Win=131328 Len=0
4	0.170324	192.168.1.195	140.238.34.216	RMI	61	JRMI, Version: 2, StreamProtocol
5	0.338060	140.238.34.216	192.168.1.195	TCP	54	1099 → 27982 [ACK] Seq=1 Ack=8 Win=29312 Len=0
6	0.338836	140.238.34.216	192.168.1.195	RMI	75	JRMI, ProtocolAck
7	0.339075	192.168.1.195	140.238.34.216	RMI	74	Continuation
8	0.344141	192.168.1.195	140.238.34.216	RMI	104	JRMI, Call
9	0.512504	140.238.34.216	192.168.1.195	TCP	54	1099 → 27982 [ACK] Seq=22 Ack=78 Win=29312 Len=0
10	0.515825	140.238.34.216	192.168.1.195	RMI	357	JRMI, ReturnData
11	0.556590	192.168.1.195	140.238.34.216	TCP	54	27982 → 1099 [ACK] Seq=78 Ack=325 Win=131072 Len=0
22	1.110576	192.168.1.195	140.238.34.216	RMI	55	JRMI, Ping
24	1.278943	140.238.34.216	192.168.1.195	RMI	55	JRMI, PingAck
25	1.279153	192.168.1.195	140.238.34.216	RMI	69	JRMI, DgcAck
29	1.488061	140.238.34.216	192.168.1.195	TCP	54	1099 → 27982 [ACK] Seq=326 Ack=94 Win=29312 Len=0
37	1.738609	192.168.1.195	140.238.34.216	TCP	54	27982 → 1099 [RST, ACK] Seq=94 Ack=326 Win=0 Len=0

Frame 8: 104 bytes on wire (832 bits) captured on interface 0
eth0 (e4:f4:c6:e4:73:de) → 140.238.34.216 (140.238.34.216)
Ack: 22, Len: 50

Java RMI
Output Stream
Serialization
Magic: 0xac ed
Version: 5

所有可见项目
选中树的所有可见项目
描述
字段名称
值
作为过滤器
将字节复制为十六进制 + ASCII 转储
...as Hex Dump
...as Printable Text
...as a Hex Stream
...as Raw Binary
...as Escaped String

0000 e4 f4 c6 e4 73 de e8 4e 06 37 ad 97 08 00 45 00N-7...E-
0010 00 5a 06 08 40 00 80 06 82 64 c0 a8 01 c3 8c ee -Z-@...d.....
0020 22 d8 6d 4e 04 4b c0 cc ac 72 4b c8 bc b2 50 18 "-ml-K-...nK...P-
0030 02 01 0c b4 00 00 50 bc ed 00 05 77 22 00 00 00P.....w....
0040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00D.....M.....t
0050 00 00 00 00 00 00 02 44 15 4d c9 d4 e6 3b df 74D.....M.....t
0060 00 06 72 65 66 4f 62 6aref0b]

这段数据由0xACED开头，有经验的同学一眼就能看出这是一段Java序列化数据。我们可以使用[SerializationDumper](#)对Java序列化数据进行分析：

```
D:\program\SerializationDumper
λ java -jar SerializationDumper-v1.1.jar "aced000577220000000000000000000000000000000000000000000000000000244154dc9d4e63bdf7400067265664f626a"

STREAM_MAGIC - 0xac ed
STREAM_VERSION - 0x00 05
Contents
TC_BLOCKDATA - 0x77
  Length - 34 - 0x22
  Contents - 0x0000000000000000000000000000000000000000000000000000000000000000244154dc9d4e63bdf
TC_STRING - 0x74
  newHandle 0x00 7e 00 00
  Length - 6 - 0x00 06
  Value - refObj - 0x7265664f626a
```

SerializationDumper输出了很多预定义常量，像 `TC_BLOCKDATA` 这种，它究竟表示什么意思呢？此时我们还得借助Java序列化的协议文档：<https://docs.oracle.com/javase/8/docs/platform/serialization/spec/protocol.html>

这篇文档里用了一种类似BNF（巴科斯范式）的形式描述了序列化数据的语法，比如我们这里的这段简单的数据，其涉及到如下语法规则：

```
1  stream:
2    magic version contents
3
4  contents:
5    content
6    contents content
7
8  content:
9    object
10   blockdata
11
12  object:
13    newObject
14    newClass
15    newArray
16    newString
17    newEnum
18    newClassDesc
19    prevObject
20    nullReference
21    exception
22    TC_RESET
23
24  blockdata:
25    blockdatashort
26    blockdataalong
27
28  blockdatashort:
29    TC_BLOCKDATA (unsigned byte)<size> (byte)[size]
30
31  newString:
32    TC_STRING newHandle (utf)
33    TC_LONGSTRING newHandle (long-utf)
```

其中 `TC_BLOCKDATA` 这部分对应的是 `contents -> content -> blockdata -> blockdatashort`，`TC_STRING` 这部分对应的是 `contents -> content -> object -> newString`。都可以在文档里找到完整的语法定义。

这一整个序列化对象，其实描述的就是一个字符串，其值是 `refobj`。意思是获取远程的 `refobj` 对象。

接着我们在序号为10的数据包中获取到了这个对象：

```

1  STREAM_MAGIC - 0xac ed
2  STREAM_VERSION - 0x00 05
3  Contents
4    TC_BLOCKDATA - 0x77
5      Length - 15 - 0x0f
6      Contents - 0x01a4462ec50000016d8d8d63578008
7    TC_OBJECT - 0x73
8      TC_PROXYCLASSDESC - 0x7d
9        newHandle 0x00 7e 00 00
10       Interface count - 2 - 0x00 00 00 02
11       proxyInterfaceNames
12         0:
13           Length - 15 - 0x00 0f
14           Value - java.rmi.Remote - 0x6a6176612e726d692e52656d6f7465
15         1:
16           Length - 5 - 0x00 05
17           Value - ICalc - 0x4943616c63
18       classAnnotations
19         TC_NULL - 0x70
20         TC_ENDBLOCKDATA - 0x78
21       superClassDesc
22         TC_CLASSDESC - 0x72
23         className
24           Length - 23 - 0x00 17
25           Value - java.lang.reflect.Proxy -
26             0x6a6176612e6c616e672e7265666c6563742e50726f7879
27             serialVersionUID - 0xe1 27 da 20 cc 10 43 cb
28             newHandle 0x00 7e 00 01
29             classDescFlags - 0x02 - SC_SERIALIZABLE
30             fieldCount - 1 - 0x00 01
31             Fields
32               0:
33                 Object - L - 0x4c
34                 fieldName
35                   Length - 1 - 0x00 01
36                   Value - h - 0x68
37                   className1
38                     TC_STRING - 0x74
39                     newHandle 0x00 7e 00 02
40                     Length - 37 - 0x00 25
41                     Value - Ljava/lang/reflect/InvocationHandler; -
42                     0x4c6a6176612f6c616e672f7265666c6563742f496e766f636174696f6e48616e646c65723
43                     b
44                     classAnnotations
45                     TC_NULL - 0x70
46                     TC_ENDBLOCKDATA - 0x78
47                     superClassDesc
48                     TC_NULL - 0x70
49                     newHandle 0x00 7e 00 03
50                     classdata
51                     java.lang.reflect.Proxy
52                     values
53                     h
54                     (object)
55                     TC_OBJECT - 0x73
56                     TC_CLASSDESC - 0x72
57                     className
58                     Length - 45 - 0x00 2d

```

```

56         value - java.rmi.server.RemoteObjectInvocationHandler -
0x6a6176612e726d692e7365727665722e52656d6f74654f626a656374496e766f636174696
f6e48616e646c6572
57         serialVersionUID - 0x00 00 00 00 00 00 02
58         newHandle 0x00 7e 00 04
59         classDescFlags - 0x02 - SC_SERIALIZABLE
60         fieldCount - 0 - 0x00 00
61         classAnnotations
62             TC_NULL - 0x70
63             TC_ENDBLOCKDATA - 0x78
64         superClassDesc
65             TC_CLASSDESC - 0x72
66             className
67                 Length - 28 - 0x00 1c
68                 value - java.rmi.server.RemoteObject -
0x6a6176612e726d692e7365727665722e52656d6f74654f626a656374
69                 serialVersionUID - 0xd3 61 b4 91 0c 61 33 1e
70                 newHandle 0x00 7e 00 05
71                 classDescFlags - 0x03 - SC_WRITE_METHOD |
SC_SERIALIZABLE
72                 fieldCount - 0 - 0x00 00
73                 classAnnotations
74                     TC_NULL - 0x70
75                     TC_ENDBLOCKDATA - 0x78
76                 superClassDesc
77                     TC_NULL - 0x70
78                 newHandle 0x00 7e 00 06
79                 classdata
80                     java.rmi.server.RemoteObject
81                     values
82                     objectAnnotation
83                         TC_BLOCKDATA - 0x77
84                         Length - 55 - 0x37
85                         Contents -
0x000a556e6963617374526566000e3134302e3233382e33342e3231360000fa00276c05080
63e8d45a4462ec50000016d8d8d6357800101
86                         TC_ENDBLOCKDATA - 0x78
87                     java.rmi.server.RemoteObjectInvocationHandler
88                     values

```

这是一个 `java.lang.reflect.Proxy` 对象，其中有一段数据储存在 `objectAnnotation` 中：

`0x000a556e6963617374526566000e3134302e3233382e33342e3231360000fa00276c0508063e8d45a4462ec50000016d8d8d6357800101`，记录了 RMI Server 的地址和端口。（中间具体调用链，下来后可以自己仔细调试分析）

在拿到 RMI Server 的地址和端口后，本机就会去连接并正式开始调用远程方法。我们再用 `tcp.stream eq 1` 筛选出本机与 RMI Server 的数据流：

- https://github.com/JetBrains/jdk8u_jdk/blob/8db9d62a1cfe07fd4260b83ae86e39f80c0a9ff2/src/share/classes/java/rmi/server/RMIOClassLoader.java#L657
- https://github.com/JetBrains/jdk8u_jdk/blob/8db9d62a1c/src/share/classes/sun/rmi/server/LoaderHandler.java#L282

所以，我们在分析序列化数据时看到的 `classAnnotations`，实际上就是 `annotateClass` 方法写入的内容。