



A Pentesters Guide to Hacking ActiveMQ Based JMS Applications

This white paper was written by:
Gursev Singh Kalra
Senior Principal Consultant
McAfee Foundstone Professional
Services

Table of Contents

Introduction	4
Messaging 101	4
Anatomy of a JMS Message	4
Message Broker	5
Messaging Models	5
JMS API	6
Apache ActiveMQ Basics	7
ActiveMQ Authentication Options	8
ActiveMQ Authorization Controls	9
ActiveMQ Administration Console	9
Pentesting JMS Applications	10
Discovery and Configuration	10
Understanding ActiveMQ Configuration File	10
Review for Weak Configuration and Known Vulnerabilities	11
Data Protection in Storage and Transit	12
Insecure Communication	12
Insecure Password Storage	13
Weak Encryption Password	13
Unencrypted KahaDB	13
User Management and Authentication	13
No Authentication	13
Simple Authentication Plug-In	13
JAAS Authentication Plug-In	14
Account Lockout Testing	14
Data Validation and Error Handling	14
Injection Attacks	14
Attacking Other Clients	15
Authorization	15
Destination Access	15
Exploitation	16
Reading Queues with QueueBrowser	16
Retrieving Messages from Topics with TopicSubscriber	16
Retrieving Messages from Topics with Durable Subscribers	17
Additional Durable Subscriber Attacks	17
Retrieving Statistics for the Broker and its Destinations	18
Dynamic Destinations	18

JMSDigger—A GUI-Based JMS Assessment Tool	19
Generic JMS Operations.....	19
ActiveMQ Specific Operations.....	19
Conclusion	20
Appendix A	21
JMS API Based Anonymous Authentication Check.....	21
JMS API Based Credential Brute Force Code.....	22
Example Password and Configuration File Decryption Code.....	24
About The Author	26
About McAfee Foundstone Professional Services	26

Introduction

Enterprise Messaging Systems (EMS) form the transactional backbone of many large organizations worldwide. They are highly reliable, flexible, and scalable systems that allow asynchronous message processing between two or more applications. This paper provides guidance on penetration testing techniques to assess the security of ActiveMQ¹ based EMS Systems written using Java Messaging Service (JMS) API^{2,3}. Applications that have been written using JMS API are also known as JMS Applications.

The paper begins with an introduction to JMS concepts that are relevant to the penetration testing techniques discussed afterwards, and then introduces JMSDigger⁴, an open source tool to engage and assess ActiveMQ based JMS Applications.

Please note that this paper does not aim to provide a comprehensive tutorial on JMS concepts or on writing code based on JMS API. There are many excellent JMS⁵ and ActiveMQ⁶ books that you can refer to if you are interested.

Messaging 101

Sun Microsystems created JSR-914⁷ as JMS API specification with an aim to provide an abstract interface to communicate with messaging providers and to write portable messaging clients (in Java) for JMS compliant message brokers. JMS applications are made up of several JMS clients and generally one JMS provider. The JMS provider is also known as a messaging server. The JMS clients create messages and send those messages to the JMS provider, which in turn routes the messages to the other clients based on its configuration and business rules.

Let us now understand the structure of a message, messaging models, and the JMS API.

Anatomy of a JMS Message

Messages are used to deliver application data and event notifications. A JMS Message is made up of message headers, message properties, and a message body as shown in the image below. Message headers contain metadata that describes message attributes like message priority, message ID, message type, message routing information, and message expiry, among others. Message properties are the additional headers that can be added to a message by the developer or message broker or can be JMS defined properties. The message body contains the actual content that is to be delivered.

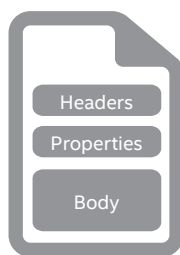


Figure 1. A JMS Message.

Messages are delivered between the clients via virtual channels called destinations that are hosted by the message broker. JMS API supports six primary message types:

1. **Message**: The `Message` type has no payload and is typically used for event notifications.
2. **TextMessage**: The `TextMessage` type carries plain text or it can be used to carry specialized data formats like XML, SOAP⁸ messages, JSON⁹, and others as its payload.
3. **MapMessage**: The `MapMessage` type contains name-value pairs as its payload.
4. **BytesMessage**: The `BytesMessage` type contains an array of primitive bytes as its payload.
5. **ObjectMessage**: The `ObjectMessage` type contains serialized Java objects as its payload.
6. **StreamMessage**: The `StreamMessage` type contains a stream of primitive Java types like `byte`, `int`, `char`, and others as its payload.

Messages are never addressed to any specific client but to the destinations. The message delivery mechanism is determined by the messaging model (discussed below) used by a particular application.

Message Broker

Message brokers form the core of the Enterprise Messaging Systems. Messages are transmitted between the messaging clients via virtual channels hosted by the message brokers, also called messaging servers. These virtual channels are also called destinations. The message brokers ensure that application data is delivered with high reliability, in an efficient manner, and minimizes coupling between messaging clients.

A large number of message brokers like ActiveMQ, RabbitMQ, SonicMQ, and IBM WebSphereMQ support JMS API for message exchange.

Messaging Models

Messaging models represent different approaches to messaging. A message broker may support either one or both the messaging models discussed below. Messaging models are also known as messaging domains.

Point-to-Point Model

The virtual channel used for communication in the point-to-point messaging model is called a Queue. In this model, the JMS clients that produce the messages are called the senders and the clients that consume the messages are called the receivers. In the point-to-point messaging model, each message can be consumed only once and it gets discarded after it is delivered to any one receiver. The send-and-receive operations to and from Queues can be performed either in a synchronous or an asynchronous fashion. The receivers can also acknowledge message consumption to the sender. The point-to-point messaging model is also referred to as p2p model.

The messages on a Queue behave differently when a `QueueBrowser` API is used to retrieve messages without disrupting a Queue's contents. When a `QueueBrowser` is used, the querying JMS client receives an enumeration of all messages on the Queue at a point in time, which the JMS client can iterate through and analyze Queue contents for any business decisions. This feature is very useful as it allows one to analyze Queue content without losing any messages.

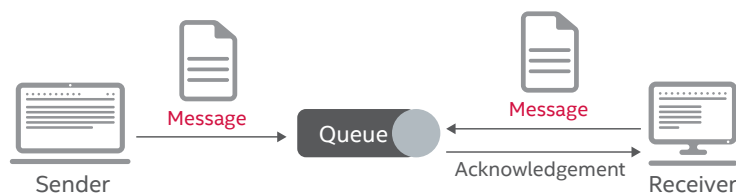


Figure 2. The image shows a point-to-point messaging model.

Publish-and-Subscribe Model

The virtual channel used for communication in the publish-and-subscribe messaging model is called a Topic. The JMS clients that produce messages are called the publishers and the JMS clients that consume the messages are called the subscribers. The publishers send messages to the Topic and the subscribers read the messages off the Topic. However, there is no direct communication between the JMS publishers and subscribers. The publish-and-subscribe messaging model is also referred to as pub/sub model. The pub/sub messaging model is subscription based.

There are two types of subscribers in the publish-and-subscribe model—nondurable and durable subscribers.

The nondurable subscribers receive the messages only when they are connected to and actively listening on a particular Topic. All messages during the period of inactivity are lost for the nondurable subscribers. Durable subscribers, however, receive all messages even if the subscriber is neither active nor connected to the message broker. Durable subscribers retain the messages until the messages are retrieved, or until the messages expire, whichever occurs first. Durable subscribers can be either dynamically or statically created by the broker administrators in the configuration files. Durable subscribers are uniquely identified by a combination of a client identifier (for the JMS client) and a durable subscriber name.

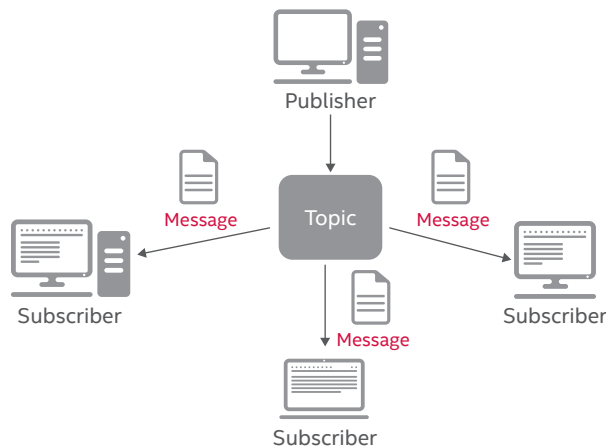


Figure 3. Image shows a publish-and-subscribe messaging model.

JMS API

JMS is an API specification and not a wire-level protocol. It is a vendor-agnostic API that can be used with any JMS compliant messaging broker. Depending on the message broker, the JMS API can also be used to communicate with non-Java or non-JMS messaging clients as we will discuss in the sections below. The JMS API can be classified into three categories:

1. The general API.
2. Point-to-point API.
3. Publish-and-subscribe API.

The API used for writing a JMS application is based on the messaging model used. The April 2002 JMS 1.1 specification, however, allows the general API to send and receive messages to and from Queues or Topics with some restrictions. Since the JMS API code is portable, it can be used to assess JMS applications created using different types of message brokers.

Apache ActiveMQ Basics

ActiveMQ is an open-source, JMS compliant message broker with a full JMS client. It provides a number of client libraries in different programming languages like Java, Ruby, Python, C, C++, and C# and can therefore be used to integrate clients written in different programming languages. For example, Java based JMS clients can talk to messaging clients written in C/C++. It supports a range of communication protocols including—but not limited to—AMQP, OpenWire, WebSockets, and Stomp as shown in the list below.

- AMQP
- RSS
- HTTP
- STOMP
- REST
- WebSockets
- OpenWire
- XMPP

ActiveMQ is highly configurable and most of the configuration information lives as child nodes inside the `broker` element of its configuration file. The configuration file can be reviewed to obtain the information such as:

1. Topic names.
2. Queue names.
3. Transport protocols enabled, assigned ports, and their configuration.
4. Credentials.
5. Authentication and Authorization details, etc.

```
<destinations>
  <topic name="TestTopic" physicalName="jms.TestTopic"/>
  <queue name="TestQueue" physicalName="jms.TestQueue"/>
</destinations>
```

Figure 4. Image shows a Topic and Queue as seen in ActiveMQ configuration file.

```
<transportConnectors>
  <transportConnector name="openwire" uri="tcp://0.0.0.0:61616"/>
  <transportConnector name="amqp" uri="amqp://0.0.0.0:5672"/>
</transportConnectors>
```

Figure 5. Image shows `openwire` and `amqp` transport protocols enabled in ActiveMQ configuration file.

ActiveMQ Authentication Options

ActiveMQ supports several mechanisms to authenticate JMS clients. ActiveMQ's authentication schemes are plug-in based and they can be changed on the broker with almost no impact on the JMS client code. This offers tremendous flexibility to the programmers. We will now look at the different authentication schemes supported by ActiveMQ.

The Default (No Authentication)

When no authentication plug-in is enabled, ActiveMQ accepts connection from any JMS client, with or without any username and password, and allows the client to interact with Queues, Topics, and perform other actions permitted as per the JMS API specification. This mode is similar to anonymous authentication mode for FTP and it offers no security.

Simple Authentication Plug-In

ActiveMQ does not allow anonymous authentication when this plug-in is enabled. The simple authentication plug-in uses hardcoded usernames and passwords in the configuration file. The simple authentication does not offer any mechanism to enforce account lockouts with failed login attempts. Additionally, the simple authentication plug-in is not scalable and is insecure because it requires clear text username and passwords to be hard coded in the configuration files.

```
<plugins>
  <simpleAuthenticationPlugin>
    <users>
      <authenticationUser username="admin" password="admin" groups="admins"/>
      <authenticationUser username="general" password="general" groups="general"/>
    </users>
  </simpleAuthenticationPlugin>
</plugins>
```

Figure 6. Image shows simple authentication plug-in enabled in ActiveMQ configuration file.

JAAS¹⁰ Based Authentication

ActiveMQ can also support custom authentication mechanisms with its JAAS plug-in support. Also, some organizations may want to centrally manage ActiveMQ authentication rather than maintaining separate credentials for the messaging brokers. They can leverage ActiveMQ's JAAS plug-in and write their own authentication routines to query their central LDAP server or their central SQL databases for authentication.

```
<plugins>
  <jaasAuthenticationPlugin configuration="activemq-domain" />
  <statisticsBrokerPlugin/>
</plugins>
```

Figure 7. Image shows JAAS plug-in enabled in ActiveMQ configuration file.

ActiveMQ Authorization Controls

ActiveMQ's authorization engine applies the security policies and enforces the type of access (read, write, administer, etc.) a JMS client is allowed for different Topics, Queues, and the messages. It supports two types of authorization controls. The first is at the destination level, and the second is at the message level.

Destination Level Authorization

Destination level authorization controls require the JMS clients to have a certain minimum level of privileges before they are allowed to connect to any Topic or Queue. The access control rules can selectively allow or deny read, write, or administration access to different JMS clients for any Topic or Queue based on ActiveMQ configuration.

```
<authorizationPlugin>
<map>
  <authorizationMap>
    <authorizationEntries>
      <authorizationEntry queue="test.%" read="test" write="test" admin="testers" />
    </authorizationEntries>
  </authorizationMap>
</map>
</authorizationPlugin>
```

Figure 8. Image shows Queue authorization entry in ActiveMQ configuration file.

Message Level Authorization

This message level authorization allows fine-grained access control where the access checks are enforced based on the contents of a message. Typically, the access control logic is written as a Java plug-in, compiled and packaged into a JAR file, and then added to ActiveMQ's classpath. The plug-in information is then specified in the ActiveMQ configuration file.

```
<messageAuthorizationPolicy>
  <bean class="com.mcafee.MessageAuthorization"
    xmlns="http://www.springframework.org/schema/beans" />
</messageAuthorizationPolicy>
```

Figure 9. Image shows message level authorization plug-in configuration.

ActiveMQ Administration Console

ActiveMQ provides a powerful web-based administration console which runs on Jetty¹¹. It allows you to manage several aspects of the message broker as mentioned below:

1. Create, read, update, or delete Topics and Queues.
2. Create and/or delete durable subscribers.
3. View active connections.
4. Send messages to Topics and Queues.

The default web console configuration runs over plain text HTTP protocol and is protected by basic authentication starting with version 5.8.0.

Now that we have some understanding of ActiveMQ and JMS concepts, we will now look at the security assessment methodology for JMS applications.

Pentesting JMS Applications

The Pentesting methodology discussed below is modeled around McAfee® Foundstone® Professional Services' security assessment framework and assumes that the environment under test is based on ActiveMQ.

Please note that even though the methodology focuses on ActiveMQ, the techniques are generic and can be applied to assess messaging applications based on other message brokers.

Discovery and Configuration

Understanding ActiveMQ Configuration File

Request the application owners to provide you with the ActiveMQ configuration file and analyze it to extract the following information:

- Authentication and Authorization mechanisms used.
- Topic and Queue names.
- Communication protocols enabled and used.
- ActiveMQ web console configuration.
- Network interfaces and corresponding listening ports.

Analyze the installed ActiveMQ version for known configuration issues and vulnerabilities. A summary of common configuration issues and a few vulnerabilities that were identified during my research is provided below.

1. ActiveMQ versions 5.8.0 onwards protect their web administration console with basic authentication with default credentials of admin/admin which is seldom changed.
2. The web admin console runs over plain text HTTP protocol which is insecure.
3. ActiveMQ's web console prior to 5.8.0 did not require authentication (see CVE-2013-3060¹²). Random Internet users could connect to ActiveMQ and perform administration functions like viewing contents of JMS destinations, delete JMS destinations, and create new JMS destinations and durable subscribers—among other operations. Unauthorized access to this interface can allow attackers to cause severe damage to the messaging application and anyone who knows the administrative URL can potentially manage the ActiveMQ instance. Performing simple Internet searches reveals several open and exposed production ActiveMQ instances as shown in two images below.

Review for Weak Configuration and Known Vulnerabilities

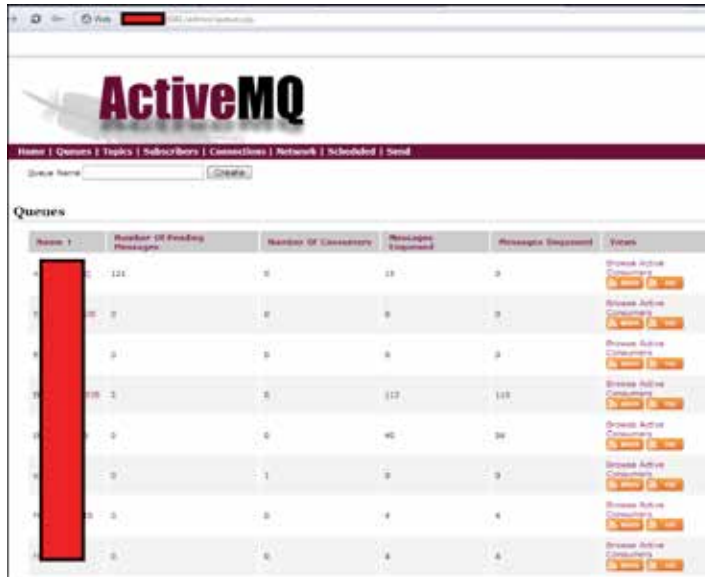


Figure 10. An unprotected ActiveMQ administrative interface available over the Internet shows queue names.



Figure 11. An unprotected ActiveMQ administrative interface available over the Internet shows active client connections.

- 4. ActiveMQ's transport connectors and administrative web console application are configured by default to listen on 0.0.0.0. Because of this, they start listening on all network interfaces of the machine and end up exposed on the Internet.

This allows inadequately hardened ActiveMQ instances on perimeter devices to be susceptible to attacks originating from the Internet. The attackers could potentially connect to ActiveMQ server, retrieve or send messages, edit or delete JMS destinations, and perform other actions permitted by the broker's API.

```

<!--
  The transport connectors expose ActiveMQ over a given protocol to
  clients and other brokers. For more information, see:

  http://activemq.apache.org/configuring-transport.html
-->
<transportConnectors>
  <transportConnector name="openwire" uri="tcp://0.0.0.0:61616"/>
</transportConnectors>

```

Figure 12. ActiveMQ openwire configuration to listen on 0.0.0.0.

```

$ ruby dumpbytes.rb 0 61616
* \000\000\000\031\001ActiveMQ\000\000\000\001\000\000\000\007\000\000\000\0\0
00\0CacheSize\005\000\000\000\000\000\000\000\000\000\000\000\000\000\000
sabled\001\000\000\000\000\000\000\000\000\000\000\000\000\000\000
*\020\020\021TcpNoDelayEnabled\001\000\000\025MaxInactivityDuration\006\000\000
\000\000\000\000\000\000\024FlightEncodingEnabled\001\000\000\021StackTraceEnabled
\001\001"

```

Figure 13. Connection to an open ActiveMQ broker's openwire interface over the Internet with 'ActiveMQ' string in response.



Figure 14. Administrative web console of a production ActiveMQ instance available on the Internet.

5. Make sure to check ActiveMQ version for known vulnerabilities. For example, Multiple XSS vulnerabilities (CVE-2013-1880¹³, CVE-2013-1879¹⁴, CVE-2012-6092¹⁵) were reported in ActiveMQ web console.

Data Protection in Storage and Transit

Insecure Communication

ActiveMQ optionally offers transport layer security, which can be implemented for the application layer protocols it supports. However, the default ActiveMQ configuration does not enforce transport layer security and is vulnerable to man-in-the-middle (MiTM)¹⁶ attacks where an attacker can capture, monitor, and modify all traffic flowing between the JMS clients and the message broker.

This vulnerability can be spotted while reviewing ActiveMQ configuration file, where the absence of `sslContext` element gives it away. However, it is important to not rely on the presence of this header alone but to thoroughly review the configuration¹⁷.

If the configuration file is not available, the JMS client and message broker communication must be reviewed with help of packet sniffers like Wireshark¹⁸.

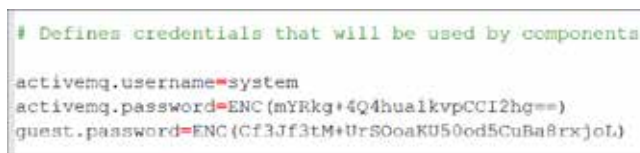
Insecure Password Storage

The `simpleAuthenticationPlugin` allows unencrypted usernames and passwords to be stored in ActiveMQ configuration files. Also, the `jaasAuthenticationPlugin` allows for an external file with plain text credentials to be used for authentication users.

It is important to review the ActiveMQ configuration files and the `jaasAuthenticationPlugin` configuration details to ensure that credentials are not stored in clear on the machine. Also, if the `jaasAuthenticationPlugin` uses an external database to store credentials, it must be reviewed for unencrypted storage.

Weak Encryption Password

ActiveMQ optionally uses password-based encryption (`jasrcrypt`¹⁹ library's `StandardPBEStrategy` class) to encrypt and store login credentials in its configuration files. Using weak passwords to encrypt login credentials can potentially expose these credentials to anyone who has access to these configuration files. System administrators often rely on password cracking tools like John the Ripper²⁰ to audit password strength. The `JMSDigger` tool which we will briefly discuss in this white paper can also assist with ActiveMQ password audits.



```
# Defines credentials that will be used by components
activemq.username=system
activemq.password=ENC(mYRkg+4Q4huaikvpCCI2hg==)
guest.password=ENC(Cf3Jf3tM+Ur50oaKU50od5CuBa8rxjoL)
```

Figure 15. The image shows encrypted strings in the default `credentials-enc.properties` file that is shipped with ActiveMQ.

Please see Appendix A for sample source code to decrypt ActiveMQ's encrypted records by brute force.

Unencrypted KahaDB

KahaDB is the default high-performance, data-persistence database used by ActiveMQ. This database is neither encrypted by default nor has the capability to perform encryption. Make sure KahaDB instance is stored on an encrypted file system partition for better security.

User Management and Authentication

No Authentication

The default ActiveMQ configuration does not offer any authentication and hence there are no user credentials. This can be tested by reviewing the configuration file for absence of `simpleAuthenticationPlugin` and `jaasAuthenticationPlugin` plug-ins.

During a black box penetration test, you can use the code snipped from Appendix A to check for anonymous authentication or you can also consider using `JMSDigger` to check for the same.

Simple Authentication Plug-In

The simple authentication plug-in uses hardcoded username and passwords in the configuration file and is confirmed by the presence of `simpleAuthenticationPlugin` child element in the configuration along with the user credentials.

This plug-in must not be used since it cannot enforce account lockouts on repeated failed login attempts and hence offers no protection against password brute force attacks.

Also, the only mechanism to change user passwords or disabling accounts is by directly editing the configuration files.

JAAS Authentication Plug-In

`jaasAuthenticationPlugin` relies on user written JAAS plug-in to perform authentication decisions. An insecurely written plug-in can expose ActiveMQ authentication to typical injection vulnerabilities like SQL injection and LDAP injection among others as we will see in the data-validation section below.

Account Lockout Testing

Please note that there is no mechanism to differentiate between `simpleAuthenticationPlugin` and `jaasAuthenticationPlugin` from a black box perspective unless `jaasAuthenticationPlugin` enforces account lockout on numerous failed login attempts. It is recommended that account lockout testing be performed using the custom code in Appendix A or by using JMSSDigger.

Data Validation and Error Handling

Injection Attacks

ActiveMQ based applications may often rely on custom code to implement JAAS plug-in for authentication, custom authorization code to perform message level authorization decisions, etc.

Insecurely written code can expose ActiveMQ authentication to typical injection vulnerabilities like SQL injection and LDAP injection among others. The vulnerable implementations can be exploited to bypass authentication and gain access to critical resources. You can also leverage JMSSDigger to test and fuzz custom authentication implementation to discover potential injection flaws.

The following image shows a SQL injection error in a JAAS module that uses MySQL for backend database.

```

INFO | Requested URL path [/deleteJob.action] onto handler "/deleteJob.action"
INFO | FrommehServlet "dispatcher": initialization completed in 111 ms
INFO | ActiveMQ console at http://0.0.0.0:8161/admin
INFO | started o.e.j.w.WebAppContext[/demo,file: [redacted] apache-activemq-5.6.0/webapps/demo/]
INFO | ActiveMQ Web Demos at http://0.0.0.0:8161/demo
INFO | started o.e.j.w.WebAppContext[/fileserver,file: [redacted] apache-activemq-5.6.0/webapp
s/fileserver/]
INFO | RESTful file access application at http://0.0.0.0:8161/fileserver
INFO | Started SelectChannelConnector@0.0.0.0:8161
About to handle callback
WARN | failed to add connection ID: [redacted] 52279-1365103226748-111, reason: java.lang.SecurityException
: user name [test] or password is invalid.
INFO | Stopping http://127.0.0.1:1552780 because failed with SecurityException: user name [test] or password is
invalid
About to handle callback
INFO | SQL exception occurred
com.mysql.jdbc.exceptions.jdbc4.MySQLSyntaxErrorException: You have an error in your SQL syntax; check the man
ual that corresponds to your MySQL server version for the right syntax to use near 'pos' at line 1
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:139)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:27
)
at java.lang.reflect.Constructor.newInstance(Constructor.java:513)
at com.mysql.jdbc.Util.handleNewInstance(Util.java:411)

```

Figure 16. In the image, box 1 shows a regular authentication failure, box 2 shows SQL error indicating SQL injection vulnerability.

Attacking Other Clients

Messaging applications often have their clients written in different programming languages and technologies. Insecurely written clients can be targeted to gain unauthorized access to other organization systems or infrastructure. For example, buffer overflow vulnerability in a C/C++ based messaging client can lead to remote command execution as depicted in the image below.

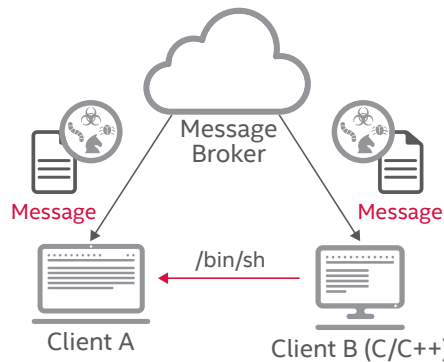


Figure 17. Command execution on a remote client via a message.

As discussed, messaging clients that consume messages from other clients can be exposed to malicious attacks. One such example is when `TextMessage` is used to exchange XML data between the clients.

When XML data is exchanged, failure to sanitize the XML and insecure XML parsing configuration can lead to cross client attacks like XML Entity Expansion²¹ or XML External Entity Injection²² attacks. Testers must keep an open eye for such scenarios.

Authorization

Destination Access

ActiveMQ does not allow JMS clients to connect to a Queue or Topic that they're not authorized to access. For each attempt to connect to Queues or Topics, the tester must ensure that connection attempts are denied by the message broker.

A successful connection is indicative of missing or weak access controls. The configuration file must also be reviewed for potentially weak authorization controls.

Exploitation

Once you have assessed Authorization controls and identified the vulnerabilities, the next step will be to perform active exploitation where you will retrieve messages from Topic and/or Queues. The availability of the JMS API with so many different messaging products allows developers to quickly port their applications between different messaging providers, and this flexibility can also be leveraged to write code for offensive purpose and target several messaging providers and messaging applications. Let us now look at some of the exploitation techniques using JMS API.

Reading Queues with QueueBrowser

A JMS Queue removes a message once it is read by any of the queue receivers. A QueueBrowser can be used to read messages from a Queue without removing them from the Queue and avoid being detected. It also offers a consistent behavior across different types of JMS providers.

Steps to retrieve messages from a Queue with QueueBrowser:

1. Initialize the environment as per the JMS provider.
2. Create an `InitialContext`.
3. Obtain a `ConnectionFactory` via JNDI lookup.
4. Obtain the Queue object via JNDI lookup.
5. Use `ConnectionFactory` from step 3 to create a new `Connection`.
6. Use the `Connection` created in step 5 to create a new `Session`.
7. Use the `Session` from step 6 to create a `QueueBrowser`.
8. Start the `Connection`.
9. Obtain an enumeration from the `QueueBrowser`'s `getEnumeration` method.
10. Iterate over the enumeration and write all the messages to a local storage.

You can also use `JMSDigger` to achieve the same.

Retrieving Messages from Topics with TopicSubscriber

Reading messages off the JMS Topics is easier than retrieving them undetected from a `QueueBrowser`. A JMS client can subscribe to a Topic and it will receive all messages addressed to the specific Topic whenever it is connected to the message broker.

Steps to retrieve messages from a Topic using `TopicSubscriber`:

Part A

1. Initialize the environment as per the JMS provider (similar to brute force example above).
2. Create an `InitialContext`.
3. Obtain a `ConnectionFactory` via JNDI lookup.
4. Obtain the Topic object using JNDI lookup.
5. Use `ConnectionFactory` from step 3 to create a new `Connection`.
6. Use the `Connection` created in step 5 to create a new `Session`.
7. Use the `Session` from step 6 to create a `TopicSubscriber`.
8. Set the `MessageListener` for the `TopicSubscriber`.
9. Start the `Connection`.

Part B

10. Implement the `MessageListener` interface (`onMessage` method).
11. Code inside the `onMessage` method should write each method to a local storage.

You can use `JMSDigger` to achieve the same effect.

The example above was of a non-durable subscription that retrieves messages for as long as its session is alive. Since you need to be connected to the message broker when you use `TopicSubscribers` to read `Topic` contents, any active and live connections from unknown IP addresses may be detected. Another approach of retrieving messages from `JMS Topics` is by using durable subscriptions.

Retrieving Messages from Topics with Durable Subscribers

There is no need to have a live connection when using durable subscribers for message retrieval. This helps us avoid detection.

Steps to retrieve messages from a `Topic` using durable subscribers:

1. Initialize the environment as per the `JMS` provider.
2. Create an `InitialContext`.
3. Obtain a `ConnectionFactory` object via JNDI lookup.
4. Obtain the `Topic` object via JNDI lookup.
5. Use `ConnectionFactory` from step 3 to create a new `Connection`.
6. Assign a client name to the `Connection`.
7. Use the `Connection` created from step 5 to create a new `Session`.
8. Use the `Session` from step 7 to create a “named” durable subscriber in context of a `Topic` obtained in step 4.
9. Start the connection.
10. Perform **synchronous** reads for one message at a time and write them to local storage.

Steps 1 through 8 will create a new durable subscriber if the client ID and durable subscriber name combination is not unique or else connect to an existing one. Steps 9 to 10 retrieve messages and write them to local storage.

`JMS` does not enforce any restrictions on client ID usage or durable subscriber name, so it is possible that ‘client A’ creates a durable subscriber and ‘client B’ (potentially malicious) reads data out of it.

Additional Durable Subscriber Attacks

You can cause a resource starvation Denial of Service by creating durable subscribers on message broker with very high transactional loads and disconnecting from the message broker. The message broker will then start accumulating messages until the durable subscriber is erased, messages expire, or messages are read from it.

Additionally, it is possible to erase durable subscribers by spoofing the client ID and providing a durable subscriber name to cause data loss for legitimate clients.

Retrieving Statistics for the Broker and its Destinations

ActiveMQ supports a statistics plug-in²³ that can be used to retrieve metadata about the broker and its destinations. This plug-in also supports wildcard characters that can retrieve information about all the destinations in one go. Critical information obtained can then be leveraged to launch further attacks on the broker and its destination. For example, broker statistics typically include system name and ActiveMQ path as shown in the images below.

```
vm : vm://localhost
memoryUsage : 0
storeUsage : 40775
tempPercentUsage : 0
ssl :
openwire : [REDACTED]:61616
brokerId : [REDACTED]0647-1365393451879-0:1
consumerCount : 3
brokerName : localhost
expiredCount : 0
dispatchCount : 17
maxEnqueueTime : 6.0
storePercentUsage : 0
dequeueCount : 13
inflightCount : 4
messagesCached : 0
tempLimit : 53687091200
averageEnqueueTime : 1.2352941176470589
stomp+ssl :
memoryPercentUsage : 0
size : 0
tempUsage : 0
producerCount : 0
minEnqueueTime : 0.0
dataDirectory : [REDACTED]apache-activemq-5.6.0/data
enqueueCount : 42
stomp :
storeLimit : 107374182400
memoryLimit : 67108864
```

Figure 18. ActiveMQ broker statistics from a test instance using JMSDigger code.

```
memoryUsage : 0
dequeueCount : 0
inflightCount : 0
messagesCached : 0
averageEnqueueTime : 0.0
destinationName : queue://jms.TestQueue
size : 0
memoryPercentUsage : 0
producerCount : 0
consumerCount : 0
minEnqueueTime : 0.0
expiredCount : 0
dispatchCount : 0
maxEnqueueTime : 0.0
enqueueCount : 0
memoryLimit : 1040576
```

Figure 19. Image shows TestQueue statistics retrieved using JMSDigger.

Dynamic Destinations

The JSR 914 specification suggests that JMS destinations are administered objects that can only be created by system administrators and retrieved by the JMS clients with JNDI²⁴ before use. However, ActiveMQ deviates from the specification and allows dynamic destinations, which can be programmatically created and then used by the JMS clients.

Such behavior can be leveraged to compromise the security of the message broker. For example:

1. A large number of destinations could be created to consume ActiveMQ resources and potentially cause integrity and availability (resource consumption and Denial of Service) issues.
2. Malicious destinations can be created on insecure brokers and then used for unauthorized messaging, botnet C&C, and towards other end results.

JMSDigger—A GUI-Based JMS Assessment Tool

JMSDigger is a new GUI-based tool that can help security professionals engage and assess ActiveMQ based JMS applications. It has the following features.

Generic JMS Operations

1. Anonymous authentication check.
2. Manual authentication check.
3. Automated credential brute force and fuzzing.
4. Retrieve messages from Topics, Queues, and durable subscribers.
5. Create new durable subscribers.
6. Erase existing durable subscribers.

ActiveMQ Specific Operations

1. Retrieve ActiveMQ broker and destination statistics.
2. Create new destinations (Topics or Queues).
3. ActiveMQ password decryption.

The two images below show a JMSDigger screenshot and example message retrieval from the test application.

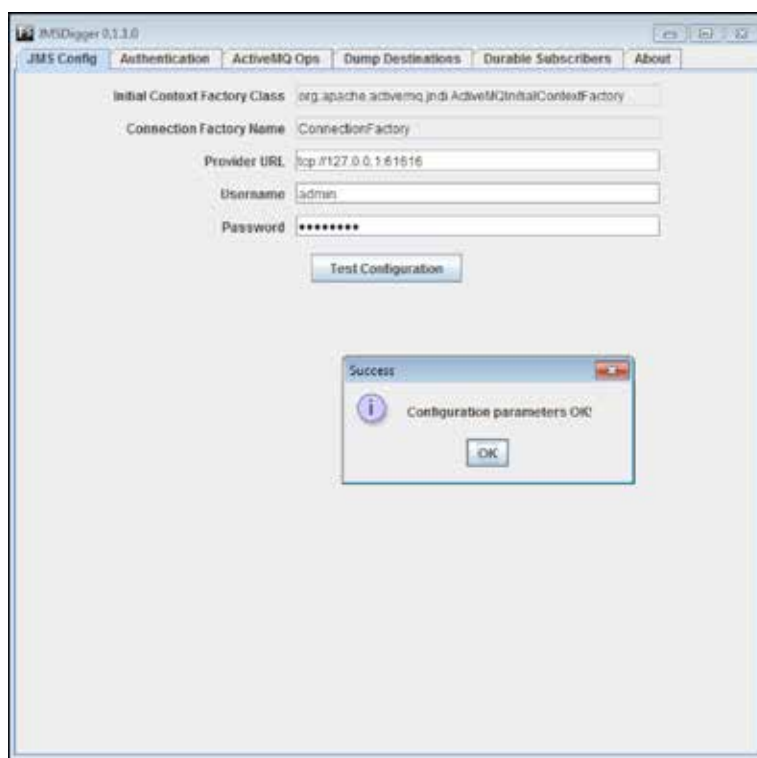


Figure 20. Image shows JMSDigger configuration check.

```

[+] Message Type : StreamMessage
[+] Extracted Stream
true, 68, c, 4.444, 3333, { 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47 } }
[+] Message Properties in XML (including binary dump in content/data element)
<org.apache.activemq.command.ActiveMQStreamMessage>
  <messageId>
    <producerId>
      <connectionId> [REDACTED] :1</connectionId>
      <sessionId>1</sessionId>
      <value>1</value>
    </producerId>
    <producerSequenceId>10</producerSequenceId>
    <brokerSequenceId>13</brokerSequenceId>
  </messageId>
  <producerId reference=" ../messageId/producerId"/>
  <destination class="org.apache.activemq.command.ActiveMQQueue">
    <string>jms_dumpQueue!Maga</string>
    <null/>
  </destination>
  <expiration>0</expiration>
  <arrival>0</arrival>
  <persistent>true</persistent>
  <priority>4</priority>
  <groupSequence>0</groupSequence>
  <compressed>false</compressed>
  <content>
    <data>AQECRAMAYwhajjU/BQAADQUFAAAABOPCQORFRkc=</data>
    <offset>0</offset>
    <length>29</length>
  </content>
  <redeliveryCounter>0</redeliveryCounter>
  <size>0</size>
  <readOnlyProperties>true</readOnlyProperties>
  <readOnlyBody>true</readOnlyBody>
  <droppable>false</droppable>
  <commandId>14</commandId>
  <responseRequired>true</responseRequired>
</org.apache.activemq.command.ActiveMQStreamMessage>

```

Message content

Message headers and properties

Figure 21. JMS destination's StreamMessage dump with JMSDigger.

Conclusion

JMS Applications have been deployed far and wide and support several large organizations worldwide. However, they have not been extensively explored for security issues. Penetration testers are encouraged to review JMS API, familiarize themselves with message brokers of their choice, and gain a deeper understanding of the technologies so they can help secure their applications.

Appendix A

JMS API Based Anonymous Authentication Check

```
package com.mcafee;

import javax.jms.JMSSecurityException;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.naming.InitialContext;
import javax.jms.JMSException;
import java.util.Properties;

public class Bruteforce {
    public static void main(String... args) throws Exception{
        InitialContext ctx;
        String cfName = "ConnectionFactory";
        Connection conn = null;
        ConnectionFactory cFact = null;

        // Prepare the environment
        Properties env = new Properties();
        // Edit the environment properties for different brokers
        env.setProperty("java.naming.factory.initial", "org.apache.activemq.
jndi.ActiveMQInitialContextFactory");
        env.setProperty("java.naming.provider.url", "tcp://localhost:61616");
        env.setProperty("connectionFactoryNames", cfName);

        // Create Initial Context
        ctx = new InitialContext(env);
        cFact = (ConnectionFactory) (ctx.lookup(cfName));
```

```
try {
    conn = cFact.createConnection(); // Attempts anonymous connection
    conn.start();

    //Control reaches here only if the connection is successful
    System.out.println("Anonymous authentication supported");

    System.exit(0);
}
catch(JMSEException ex) {
    System.out.println(ex.getMessage());
}
br.close();
}
```

JMS API Based Credential Brute Force Code

```
package com.mcafee;

import java.io.BufferedReader;
import java.io.FileReader;
import javax.jms.JMSSecurityException;
import javax.jms.Connection;
import javax.jms.ConnectionFactory;
import javax.naming.InitialContext;
import javax.jms.JMSEException;
import java.util.Properties;

public class Bruteforce {

    public static void main(String... args) throws Exception{
        InitialContext ctx;

        // File with 1000 different passwords (each password in a new line)
        String filename = "1000Passwords.txt";
        String password;
        String cfName = "ConnectionFactory";
        Connection conn = null;
        ConnectionFactory cFact = null;
```

```
BufferedReader br = new BufferedReader(new FileReader(filename));
// Prepare the environment
Properties env = new Properties();
// Edit the environment properties for different brokers
env.setProperty("java.naming.factory.initial", "org.apache.activemq.
jndi.ActiveMQInitialContextFactory");
env.setProperty("java.naming.provider.url", "tcp://localhost:61616");
env.setProperty("connectionFactoryNames", cfName);

// Create Initial Context
ctx = new InitialContext(env);
cFact = (ConnectionFactory) (ctx.lookup(cfName));

while((password = br.readLine()) != null) {
    System.out.println("Trying password => " + password);
    try {
        conn = cFact.createConnection("system", password);
        // conn = cFact.createConnection(); Attempts anonymous
connection
        conn.start();
        //Control reaches here only if the connection is successful
        System.out.println("Password found => " + line);
        System.exit(0);
    }
    catch(JMSEException ex) {
        System.out.println(ex.getMessage());
    }
}
br.close();
}
```

Example Password and Configuration File Decryption Code

The example Java class below accepts a list of passwords and one encrypted string from the configuration file. It then tries to decrypt the encrypted string until it identifies the correct password and returns it. Otherwise null is returned. All credentials can be extracted from the configuration file once a correct password is guessed.

```
package com.mcafee;

import java.util.ArrayList;
import org.jasypt.encryption.pbe.StandardPBEStrngEncryptor;
import org.jasypt.exceptions.EncryptionOperationNotPossibleException;

public class JmsPasswordOps {
    private ArrayList<String> passwords = new ArrayList<String>();
    private StandardPBEStrngEncryptor encryptor = new
StandardPBEStrngEncryptor();

    public void addPassword(String password) {
        if(password == null)
            throw new IllegalArgumentException("Password cannot be null");
        this.passwords.add(password);
    }

    public void addPasswordList(ArrayList<String> passwords) {
        if(passwords == null || passwords.size() == 0)
            throw new IllegalArgumentException("Password ArrayList cannot be null or
of zero length");

        for(String pass: passwords) {
            if(pass != null)
                this.passwords.add(pass);
        }
    }
}
```



```
public void clearPasswords() {
    passwords.clear();
}

public String decrypt(String encryptedText) throws JmsDiggerException {
    String result = null;
    if(encryptedText == null)
        throw new IllegalArgumentException("Encrypted text cannot be null");

    if(passwords.size() == 0)
        throw new IllegalArgumentException("No password provided");
    String pass;
    for(pass : passwords) {
        //New object is required for each decryption attempt
        encryptor = new StandardPBEStrngEncryptor();
        try {
            encryptor.setPassword(pass);
            result = encryptor.decrypt(encryptedText);
        } catch (EncryptionOperationNotPossibleException ex) {
            //Absorb this to be able to run through a large number of passwords
        }
    }
    if(result == null)
        return null;
    return pass; // returns null if password could not be decrypted
}
}
```

About The Author

Gursev Singh Kalra serves as a Senior Principal with McAfee Foundstone Professional Services, a division of McAfee. Gursev has authored several security-related white papers and his research has been voted among the top ten web hacking techniques of 2011 and 2012. He loves to code and has authored several free security tools like JMSDigger, TesserCap, Oyedata, SSLSmart, and clipcaptcha. He has spoken at conferences such as BlackHat, ToorCon, OWASP, NullCon, Infosec Southwest, and more.

About McAfee Foundstone Professional Services

McAfee Foundstone Professional Services, a division of McAfee, offers expert services and education to help organizations continuously and measurably protect their most important assets from the most critical threats. Through a strategic approach to security, McAfee Foundstone identifies and implements the right balance of technology, people, and process to manage digital risk and leverage security investments more effectively. The company's professional services team consists of recognized security experts and authors with broad security experience with multinational corporations, the public sector, and the US military. <http://www.mcafee.com/us/services/mcafee-foundstone-practice.aspx>

About McAfee

McAfee is now part of Intel® Security. With its Security Connected strategy, innovative approach to hardware-enhanced security, and unique Global Threat Intelligence, Intel Security is intensely focused on developing proactive, proven security solutions and services that protect systems, networks, and mobile devices for business and personal use around the world. Intel Security combines the experience and expertise of McAfee with the innovation and proven performance of Intel to make security an essential ingredient in every architecture and on every computing platform. Intel Security's mission is to give everyone the confidence to live and work safely and securely in the digital world. www.intelsecurity.com.

1. <http://activemq.apache.org/>
2. http://en.wikipedia.org/wiki/Java_Message_Service
3. <http://jcp.org/aboutJava/communityprocess/final/jsr914/index.html>
4. <https://github.com/OpenSecurityResearch/jmsdigger>
5. Java Messaging Service, O'Reilly Media
6. <http://www.manning.com/snyder/>
7. <https://www.jcp.org/en/jsr/detail?id=914>
8. <http://en.wikipedia.org/wiki/SOAP>
9. <http://en.wikipedia.org/wiki/JSON>
10. http://en.wikipedia.org/wiki/Java_Authentication_and_Authorization_Service
11. <http://www.eclipse.org/jetty/>
12. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-3060>
13. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1880>
14. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-1879>
15. <http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-6092>
16. http://en.wikipedia.org/wiki/Man-in-the-middle_attack
17. <http://activemq.apache.org/how-do-i-use-ssl.html>
18. <http://www.wireshark.org/>
19. <http://www.jasypt.org/>
20. <http://www.openwall.com/john/>
21. [https://www.owasp.org/index.php/Testing_for_XML_Injection_\(OWASP-DV-008\)](https://www.owasp.org/index.php/Testing_for_XML_Injection_(OWASP-DV-008))
22. http://projects.webappsec.org/w/page/13247003/XML_External_Entities
23. <http://activemq.apache.org/statisticsplugin.html>
24. http://en.wikipedia.org/wiki/Java_Naming_and_Directory_Interface

