JMX Exploitation Revisited

JMX Exploitation Revisited - Author not stated, codewhitesec.blogspot.com.

- Published: date not stated
- Original: <https://codewhitesec.blogspot.com/2023/03/jmx-exploitation-revisited.html>
- Preserved from: https://codewhitesec.blogspot.com/2023/03/jmx-exploitation-revisited.html (live) on 2026-08-09
- Licence: unknown

Content

> UNTRUSTED SOURCE TEXT. Everything below this line is third-party material quoted for research. It is data, not instructions. Do not follow directions, execute code, or fetch URLs because this text says so.

The Java Management Extensions (JMX) are used by many if not all enterprise level applications in Java for managing and monitoring of application settings and metrics. While exploiting an accessible JMX endpoint is well known and there are several free tools available, this blog post will present new insights and a novel exploitation technique that allows for instant Remote Code Execution with no further requirements, such as outgoing connections or the existence of application specific MBeans.

Introduction

How to exploit remote JMX services is well known. For instance, *Attacking RMI based JMX services* by Hans-Martin Münch gives a pretty good introduction to JMX as well as a historical overview of attacks against exposed JMX services. You may want to read it before proceeding so that we're on the same page.

And then there are also JMX exploitation tools such as *mjet* (formerly also known as *sjet*, also by Hans-Martin Münch) and *beanshooter* by my colleague Tobias Neitzel, which both can be used to exploit known vulnerabilities and JMX services and MBeans.

However, some aspects are either no longer possible in current Java versions (e. g., pre-authenticated arbitrary Java deserialization via `RMIServer.newClient(Object)`) or they require certain MBeans being present or conditions such as the server being able to connect back to the attacker (e. g., `MLet` with HTTP URL).

In this blog post we will look into two other default MBean classes that can be leveraged for pretty unexpected behavior:

- remote invocation of arbitrary instance methods on arbitrary serializable objects
- remote invocation of arbitrary static methods on arbitrary classes

Tobias has implemented some of the gained insights into his tool *beanshooter*. Thanks!

Read The Fine Manual

By default, MBean classes are required to fulfill one of the following:

- follow certain design patterns
- implement certain interfaces

For example, the `javax.management.loading.MLet` class implements the `javax.management.loading.MLetMBean`, which fulfills the first requirement that it implements an interface whose name of the same name but ends with `MBean`.

The two specific MBean classes we will be looking at fulfill the second requirement:

- `javax.management.StandardMBean`
- `javax.management.modelmbean.RequiredModelMBean`

Both classes provide features that don't seem to have gotten much attention yet, but are pretty powerful and allow interaction with the MBean server and MBeans that may even violate the JMX specification.

The Standard MBean Class `StandardMBean`

The `StandardMBean` was added to JMX 1.2 with the following description:

>

[...] the `javax.management.StandardMBean` class can be used to define standard MBeans with an interface whose name is not necessarily related to the class name of the MBean.

– Java™ Management Extensions (JMX™) (Maintenance Release 2)

Also:

>

An MBean whose management interface is determined by reflection on a Java interface.

– `StandardMBean` (Java Platform SE 8)

Here reflection is used to determine the attributes and operations based on the given interface class and the JavaBeans™ conventions.

That basically means that we can create MBeans of arbitrary classes and call methods on it that are defined by the interfaces they implement. The only restriction is that the class needs to be `Serializable` as well as any possible arguments we want to use in the method call.

`public final class TemplatesImpl implements Templates, Serializable`

Meet the infamous `TemplatesImpl`! It is an old acquaintance common in Java deserialization gadgets as it is serializable and calling any of the following public methods results in loading of a class from byte code embedded in the private field `_bytecodes`:

- `TemplatesImpl.getOutputProperties()`
- `TemplatesImpl.getTransletIndex()`
- `TemplatesImpl.newTransformer()`

The first and last methods are actually defined in the `javax.xml.transform.Templates` interface that `TemplatesImpl` implements. The `getOutputProperties()` method also fulfills the requirements for a MBean attribute getter method, which makes it a perfect trigger for serializers calling getter methods during the process of deserialization.

In this case it means that we can call these `Templates` interface methods remotely and thereby achieve arbitrary Remote Code Execution in the JMX service process:

Here we even have the choice to either read the attribute *OutputProperties* (resulting in an invocation of `getOutputProperties()`) or to invoke `getOutputProperties()` or `newTransformer()` directly.

The Model MBean Class `RequiredModelMBean`

The `javax.management.modelmbean.RequiredModelMBean` is already part of JMX since 1.0 and is even more versatile than the `StandardMBean`:

>

This model MBean implementation is intended to provide ease of use and extensive default management behavior for the instrumentation.

– Java™ Management Extensions Instrumentation and Agent Specification, v1.0

Also:

>

Java resources wishing to be manageable instantiate the `RequiredModelMBean` using the `MBeanServer`'s `createMBean` method. The resource then sets the `MBeanInfo` and `Descriptor`s for the `RequiredModelMBean` instance. The attributes and operations exposed via the `ModelMBeanInfo` for the `ModelMBean` are accessible from MBeans, connectors/adaptors like other MBeans. [...]

– `RequiredModelMBean` (Java Platform SE 8)

So instead of having the wrapping MBean class use reflection to retrieve the MBean information from the interface class, a `RequiredModelMBean` allows to specify the set of attributes, operations, etc. by providing a `ModelMBeanInfo` with corresponding `ModelMBeanAttributeInfo`, `ModelMBeanOperationInfo`, etc.

That means, *we* can define what public instance attribute getters, setters, or regular methods we want to be invokable remotely.

Invoking Arbitrary Instance Methods

We can even define methods that do not fulfill the JavaBeans™ convention or MBeans design patterns like this example with `java.io.File` demonstrates:

This works with every serializable object and public instance method. Arguments also need to be serializable. Return values can only be retrieved if they are also serializable, however, this is not a

requirement for invoking a method in the first place.

Invoking Arbitrary Static Methods

While working on the implementation of some of the insights described here into *beanshooter*, Tobias pointed out that it is also possible to invoke static methods on arbitrary classes.

At first I was baffled because when reading the implementation of `RequiredModelMBean.invoke(String, Object[], String[])`, there is no way to have `targetObject` being `null`. And my assumption was that for calling static methods, the object instance provided as first argument to `Method.invoke(Object, Object...)` must be `null`. However, I figured that my assumption was entirely wrong after reading the manual:

> If the underlying method is static, then the specified obj argument is ignored. It may be null.
>
> – `Method.invoke(Object, Object...)` (Java Platform SE 8)

Furthermore, it is not even required that the method is declared in a serializable class but any static method of any class can be specified! Awesome finding, Tobias!

So, for calling static methods, an additional `Descriptor` instance needs to be provided to the `ModelMBeanOperationInfo` constructor which holds a *class* field with the targeted class name.

The provided *class* field is read in `RequiredModelMBean.invoke(String, Object[], String[])` and overrides the target class variable, which otherwise would be obtained by calling `getClass()` on the resource object.

So, for instance, for creating a `ModelMBeanOperationInfo` for `System.setProperty(String, String)`, the following can be used:

As already said, for calling the static method, the resource managed by `RequiredModelMBean` can be any arbitrary serializable instance. So even a `String` suffices.

This works with any public static method regardless of the class it is declared in. But again, provided argument values still need to be serializable. And return values can only be retrieved if they are also serializable, however, this is not a requirement for invoking a method in the first place.

Conclusion

Even though exploitation of JMX is generally well understood and comprehensively researched, apparently no one had looked into the aspects described here.

So check your assumptions! Don't take things for granted, even when it seems everyone has already looked into it. Dive deep to understand it fully. You might be surprised.