Open in app ↗ Sign up Sign in

Medium Search Write

Programming + Java + Security + Cve + Jackson +

On Jackson CVEs: Don’t Panic — Here is what you need to know

Addendums (19-May-2020, 28-Sep-2019):



@cowtowncoder 8 min read · Dec 22, 2017

-- 5 --

Important notes:

- Jackson 2.11 added `MapperFeature.BLOCK_UNSAFE_POLYMORPHIC_BASE_TYPES` which can be enabled to prevent use of potentially unsafe 2.10-deprecated legacy methods: read more on “Jackson 2.11 Features” blog post.
- Jackson 2.10 supports new “Safe Default Typing” via `activateDefaultTyping()` (instead of deprecated `enableDefaultTyping()`): read more on “Jackson 2.10 Features” blog post.

Background

During 2017 many vulnerabilities based on so-called “serialization gadgets” (or just “gadgets”) were reported. Reports cover many different Java serialization frameworks and libraries, including Jackson.

Gadgets are classes that allow calling otherwise non-accessible methods or accessing non-accessible data, when instantiated and modified via setters and/or assignments to fields (necessary during reading of data into Objects, aka deserialization; or during writing of Objects as data, aka Serialization). With proper initialization they will perform operations as side effects: operations that may for example allow remote execution of commands or scripts, or that expose data from the file system of server handling deserialization.

As a hypothetical example let’s consider class `Bomb` with method `setTimeToDetonate(int seconds)`, calling of which shuts down JVM (or Operating System) after specific delay. If serialization system allows construction of `Bomb` and calling of `setTimeToDetonate` (for example by passing property “timeToDetonate”) this is a valid attack vector for denial-of-service attack. Most gadgets require calling of one or more setter methods in specific order, and then allow either reading of a system file, execution of a command, or sometimes execution of code specified in payload (as byte code possibly encoded as Base64).

Originally research started with focus on default JDK serialization. A good overview is this article from 2015:

- http://frohoff.github.io/appseccali-marshalling-pickles/

which covers tools that is still being developed, updated with new exploits:

- https://github.com/frohoff/ysoserial

Currently Published State of the Art Gadgets

A more recent take, covering many other libraries/frameworks, can be found from:

- https://github.com/mbechler/marshalsec

(there’s a link to pdf presentation in README: repo itself contains sample exploit(s))

This project documents about two dozen potential, commonly available gadgets over a dozen frameworks, including Jackson. Report is good reading in general so I encourage you to read (or at least skim) it if at all interested in security aspects of Object serialization.

There is also at least one existing Jackson-specific article:

- https://adamcaudill.com/2017/10/04/exploiting-jackson-rce-cve-2017-7525/

which explains specific exploit applicable against Jackson quite well (if not in deep detail); but I thought it would make sense to explain exact requirements (applicability of exploit) in bit more detail because the exploit described can only be used under specific circumstances and is NOT generally available just because Jackson is being used for serialization.

Am I Affected if I Use Jackson for JSON?

As usual, the full answer is “It Depends”. But the likely answer is “probably not”. The reason for this is that for exploit to work, multiple things must be true and commonly at least one pre-requisite is not true.

But let’s look at these pre-requisite.

What is Required for a Jackson-based “gadget” Exploit?

Simply put, exploit can work if all of these conditions are true, if your system:

1. Accepts JSON content sent by untrusted client (composed manually or by code you did not write and have no visibility or control over) — meaning that you can not constrain JSON itself that is being sent
2. Has at least one specific “gadget” class to exploit in the Java classpath: and specifically class from these that works with Jackson (most gadgets only work with specific library or libraries — most commonly reported ones for example only work with JDK serialization)
3. Enable polymorphic type handling for properties with nominal type of `java.lang.Object` (or one of small number of “permissive” tag interfaces; `java.util.Serializable`, `java.util.Comparable`)
4. Use version of Jackson that does not (yet) block “gadget” class in question (set of published exploits grows over time so it is a race between exploits and patches)

Of these, (1) means that if you control both endpoints, this exploit is not available — but this is presumed to be the starting point since without attacker there will not be attacks (although it is worth noting that in multi-phase attack internal communication may be compromised and attacker gains access).

Point (2) simply means that some of the exploits rely on libraries that your application / service has dependencies on. Since some of “gadgets” are

included in JDK itself, it is likely that you may well have at least one such type accessible.

So the main points to cover here are (3) and (4).

What Does Polymorphic Typing Even Mean?

Let’s start with the definition of “polymorphic handling”. By default, specific types used have to be declared by POJO or caller, so you use classes like:

```
public class Person {
  public String name;
  public int age;
  public PhoneNumber phone; // embedded POJO
}
public class PhoneNumber {
  public int areaCode;
  public int local;
  // ... and so forth; not including optional getters/setters
}
```

and deserializer knows how to handle values for matching JSON values. Incoming JSON might look something like:

```
{ "name" : "Bob",
  "age" : 28.
  "phone" : {
     "areaCode" : 555,
     "local" : 1234567
  }
}
```

But in Object-Oriented Programming, you often use polymorphism: certain base type has multiple alternative implementations.

In above example we could, say, define different kinds of phone numbers (international? Cell phone?). So you could have (and yes, example is contrived — but humor me):

```
abstract class PhoneNumber {
   public int areaCode, local;
}
public class InternationalNumber extends PhoneNumber {
   public int countryCode;
}
public class DomesticNumber extends PhoneNumber { }
```

Serializing (writing out) of `Person` instances would not be problematic, as type is available and be introspected for properties. But trying to read values back would give us an exception because `Person` is an abstract type and you can not create instances of abstract types; you must use a concrete subtype. And deserializer does not know which subtype to use without further information.

Get @cowtowncoder’s stories in your inbox

Join Medium for free to get updates from this writer.

Enter your email

Subscribe

Remember me for faster sign in

Jackson can support this if we allow/require addition of “type id”: value that allows deserializer to know which subtype to create instance of. Easiest way is to use annotation `@JsonTypeInfo`, using one of two mechanisms:

```
// enable polymorphic handling for every PhoneNumber-valued property
@JsonTypeInfo(use = Id.CLASS)
// note: there are other ways to register subtypes, too
@JsonSubTypes({ InternationalNumber.class, DomesticNumber.class })
abstract class PhoneNumber {
}

// or:
public class Person {
   @JsonTypeInfo(...) // only enable for this property!
   public PhoneNumber phone;
}
```

When doing this, serializer will add a type identifier in JSON (depending on configuration can be additional property, or array or object structure “wrapping” value); and deserializer, conversely, knows to expect this. We will see something like:

```
{ "phone" : {
   "@class" : "package.InternationalNumber",
   "areaCode" : 555,
   ...
  }
}
```

Note that this is just one of the options, regarding type id to include, and mechanism for inclusion. For more information on usage you may want to read:

- http://www.baeldung.com/jackson-annotations
- http://programmerbruce.blogspot.com/2011/05/deserialize-json-with-jackson-into.html

- http://www.studytrails.com/java/json/java-jackson-serialization-polymorphism/

The key part here is that if (and only if!) class name is used as the type id, then sender could send any class name they want to: within limits of type compatibility Jackson deserializer will (try to) create an instance of that type and initialize it by calling setter methods and/or setting values to fields.

Another Way to Enable Polymorphic Handling in Jackson

So, this is the most common way to enable polymorphic handling for specific properties. But it requires annotating either all polymorphic value types, or all properties (or some combination thereof). This may not be practical, especially with 3rd party value types (although even in those cases it is possible to use this approach, with “mix-in” annotations).

Because of this reason, there is one other way to enable polymorphic handling: “Default Typing”.
Default typing is enabled via `ObjectMapper`:

```
ObjectMapper mapper = new ObjectMapper();
mapper.enableDefaultTyping(); // or one of variants
```

and what it does is equivalent to adding:

```
@JsonTypeInfo(use = Id.CLASS, include = As.WRAPPER_ARRAY)
```

on every property that matches inclusion criteria (which, by default, is `DefaultTyping.OBJECT_AND_NON_CONCRETE`). In our case it would apply this to all properties with `abstract` type value, including `PhoneNumber`.

So Is Polymorphic Handling Always Dangerous?

So far so good. With polymorphic handling attacker can potentially produce types of values that service did not mean to allow. However, this does NOT yet mean she can use actual Gadget types — remember that type to use MUST match base type. This particular case is not yet problematic, unless you happen to find a sub-class of `PhoneNumber` that is a serialization gadget — which is unlikely (or more importantly, unlikely that attacker would know such a class is available).

So we have to make another change to declaration:

```
public class Person {
  @JsonTypeInfo(use = Id.CLASS)
  public Object phone;
}
```

Why is this important? Because all Java classes are instances of `Object`, so now attacker can select from a wide selection of documented “gadgets”. Although only from a set that works with Jackson’s setter/field assignment (some attacks only work against JDK serialization, or against other serialization frameworks).

Also note that some other “tag” types can be similarly problematic:

- `java.util.Serializable` is implemented by many types (including many Gadget classes)
- `java.util.Comparable` and `java.lang.Cloneable` are also commonly implemented by various classes

Is There Anything Jackson Does to Protect You?

So now we know the first 3 pre-requisites for successful exploit. But there is one more requirement.

Given that there is active research to find gadget types, Jackson databind module is being updated, incrementally, to prevent use of these types. This means, simply put, that deserialization of a set of “well-known” Gadget types is blocked: they can not be read in via polymorphic handling (or actually even with explicit type).

Since inclusion of checks requires releasing of new versions, it is important to try to keep your Jackson version as up-to-date as possible. As of February 2019, latest versions are:

- `2.8.11`
- `2.9.8`

and these have the most recent updates.

To learn more about active development, you may also want to check out issues reported:

- https://github.com/FasterXML/jackson-databind/labels/CVE

(note: you may want to include `closed` issues as well as open ones)

What to do to Protect My System?

Knowing all of above, we know something about general applicability.
So here are some suggestions for further reduce likelihood of successful attacks:

1. Try to keep up with updated versions of Jackson (`jackson-databind`): it should always be safe to upgrade to the latest patch version of given minor version (safest in the sense they should be no breaking changes to functionality)
2. If possible, AVOID enabling default typing (since it is usually class name based). It is better to be explicit about specifying where polymorphism is needed.
3. AVOID using `java.lang.Object` (or, `java.util.Serializable`) as the nominal type of polymorphic values, regardless of whether you use per-type, per-property, or Default Typing
4. If possible USE “type name” and NOT classname as type id: `@JsonTypeInfo(use = Id.NAME)` — this may require annotation of type name (see `@JsonTypeName` and `@JsonSubTypes`)

Especially consider the fact that if you can do either (3) or (4), you will prevent use of this class of exploits.

Programming + Java + Security + Cve + Jackson +



Written by @cowtowncoder

612 followers · 26 following

Open Source developer, most known for Jackson data processor (nee “JSON library”), author of many, many other OSS libraries for Java, from ClassMate to Woodstox