Fastjson: exceptional deserialization vulnerabilities

Fastjson: exceptional deserialization vulnerabilities - Peter Stöckli, alphabot.com.

- Published: date not stated
- Original: <https://www.alphabot.com/security/blog/2020/java/Fastjson-exceptional-deserialization-vulnerabilities.html>
- Preserved from: https://www.alphabot.com/security/blog/2020/java/Fastjson-exceptional-deserialization-vulnerabilities.html (stored) on 2026-08-11
- 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.

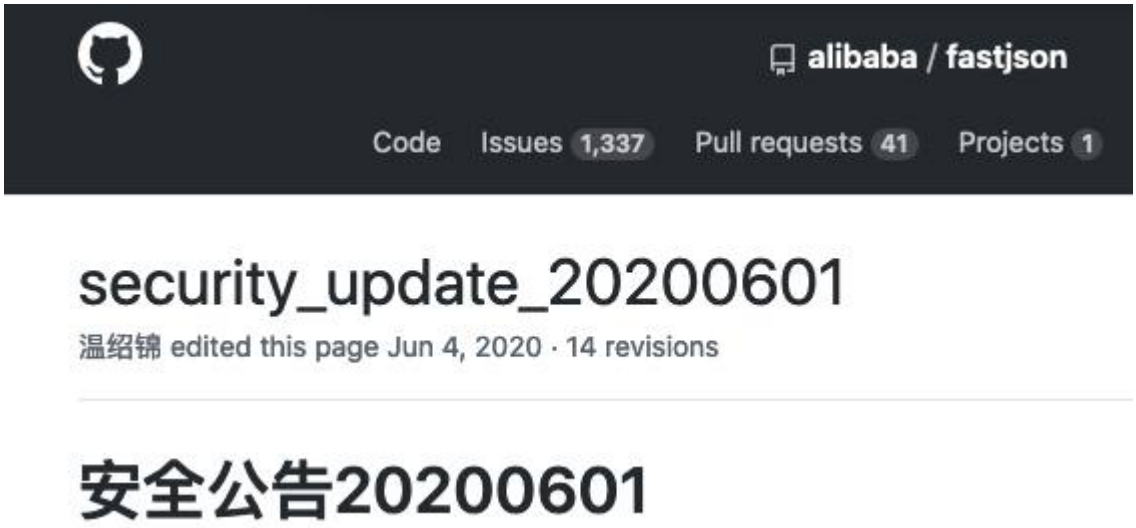
Back

15 Jul 2020 | Peter Stöckli

Fastjson: exceptional deserialization vulnerabilities

Intro

Many of you may never have heard of the Java based JSON serialization library called Fastjson, although it's quite an interesting piece of software. Fastjson is an open source project of the Chinese Internet giant Alibaba and has 22'000 stars on GitHub (and coincidentally 1337 open issues) at the time writing of this blog post.



Like Jackson(-Databind) and other JSON serialization libraries Fastjson comes with a so-called AutoType-feature, which instructs the library to deserialize JSON input using types provided by the JSON (using an extra JSON field called `@type`). Now we know at least since Muñoz/Mirosh's Friday-The-13th-JSON-Attacks and Bechler's marshalsec, that deserializing any input where the types can be provided is potentially insecure and dangerous. And that is especially true if the types can be provided from a remote user (like a JSON object or a ViewState). Now since the Fastjson developers are aware of this, autoType isn't enabled by default. And even if it is enabled by the developer using this library there is an ever-growing list of types that are not allowed at play.

Fastjson maintains deny lists to prevent classes that could potentially lead to RCE from being instantiated (so-called gadgets). To achieve this an array called `denyHashCodes` is maintained containing the hashes of forbidden packages and class names.

For example, `0xC00BE1DEBAF2808BL` is the hash for "`jdk.internal.`".

The hash function in use (TypeUtils#fnv1a_64) is a 64 bit flavor of the FNV-1a non-cryptographic hash function. The reason for this hash-based deny list seems to be some kind of security by obscurity game.

The unrelated GitHub project called fastjson-blacklist contains a list with many of the hashes and their effective package or class name and a corresponding `BreakerUtil`.

Fun fact: `Arrays.binarySearch` is used for checking the denyList for the hash of the type name. That list is only programmatically sorted in some cases. This means that the developers of Fastjson have to be extra careful when adding new entries to the denyList, because they could make parts of the `denyList` useless. (Hint: `binarySearch` requires arrays to be sorted to work as intended)

Typical Fastjson RCEs (using the autoType-feature)

Needless to say that new classes that can cause some kind of RCE are discovered all the time, which then leads to the extension of the deny list and the release of a new version of Fastjson (a similar as path Jackson-databind had taken, before they replaced their deny list with an allow list).

An example of such a gadget would be the JDK class `javax.swing.JEditorPane`, that worked until Fastjson 1.2.68 (released in March of 2020). It can be used for remote detection of older Fastjson

versions with autoType enbabled or alternatively to exploit a blind SSRF vulnerability.

A simple payload using that gadget would look like this:

```
{"@type":"javax.swing.JEditorPane","page": "https://sectests.net/canary/sample"}
```

If Fastjson before the version 1.2.69 with autoType enabled is in use and the payload above is parsed it instantiates the JDK class `javax.swing.JEditorPane` and calls its `setPage` method, which in turn makes a simple HTTP `GET` request to the URL specified. (As said before this gadget is mostly interesting for the remote detection of a vulnerable application using Fastjson.)

Now it gets interesting...

Most people that know of the dangers of deserialization vulnerabilities won't be surprised so far. However, while looking at the library I found some interesting things:

The global Fastjson instance

One of these interesting things is that there is a global Fastjson instance, that allows to change its serialization settings. So, it might happen that one developer enables the `autoType` feature on the global instance for storing some serialized values into a Redis datastore (which in itself is not that dangerous yet):

```
ParserConfig.getGlobalInstance().setAutoTypeSupport(true);
```

Whilst another developer parses JSON from a remote data source in another part of the same codebase, with the same global instance. So, a new RCE was created:

```
JSON.parse(payload);
```

It looks so harmless, doesn't it?

How many autoType checks?

In Fastjson 1.2.70 the `JSONException` with the message `"autoType is not support."` (sic) is thrown at *nine* different places in the `ParserConfig` class. One could argue about the reasons why the authors deem this necessary. In my simplified, naïve view of the world I would expect one place of code where such an exception is thrown ("No really, autoType is not supported, we won't instantiate your stupid class.). End of discussion. But in this library, there are nine of those autoType checks, enabling people to find all kinds of unintended bypasses.

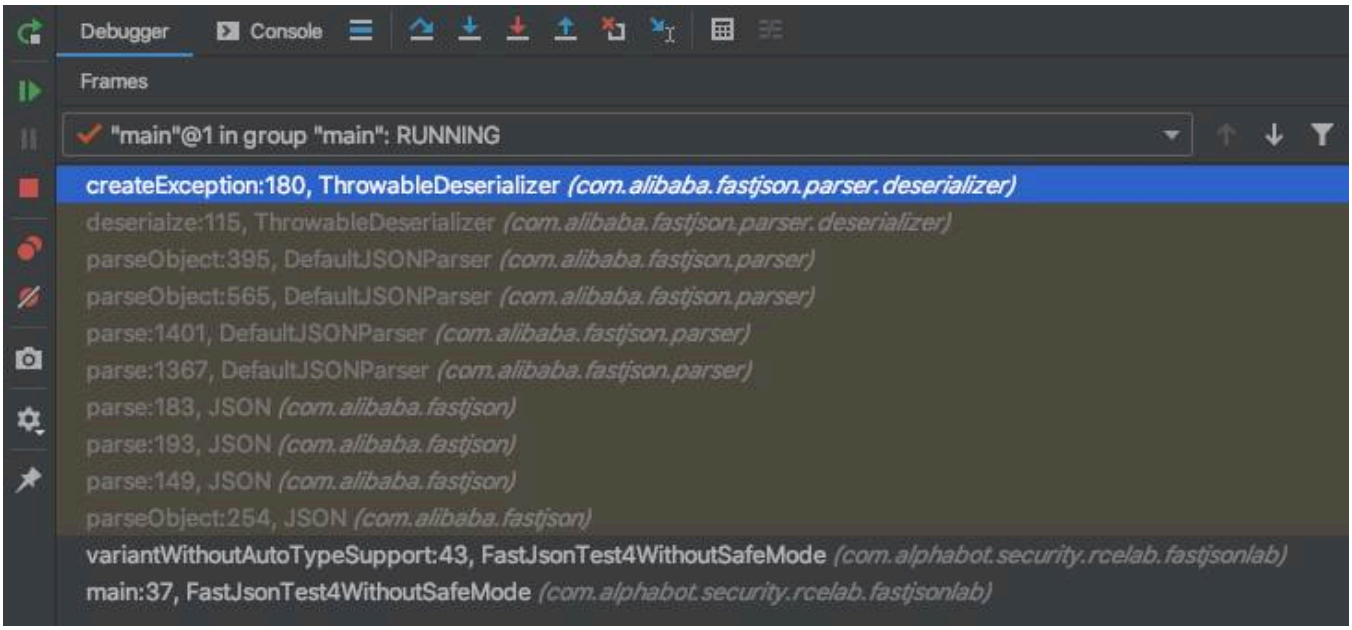
But can you make an Exception (instance), please?

So, let's assume we have autoType disabled. Nine different checks should be enough? Right?

Well...

In Mai of 2020 someone discovered that despite autoType being disabled it was possible to instance Exceptions... and leak some data using them. Let's look at how that was possible:

Just because autoType was not enabled didn't mean that no classes could be instantiated. It just meant you couldn't instantiate most classes...



So, with a simplified payload like:

```
{"@type":"java.lang.Exception","@type":"java.lang.RuntimeException"}
```

it was possible to instantiate a simple RuntimeException, despite autoType being disabled.

When we look at what happens after we call `parse` on the `com.alibaba.fastjson.JSON` class we see the following behavior: At one time both our types `java.lang.Exception` and `java.lang.RuntimeException` have to go through the `checkAutoType` in the `ParserConfig` class, where the over 200(!) lines long `checkAutoType` method checks following things (excerpt):

- whether `safeMode` is enabled (it is not)
- whether the type name is shorter or equal to 192 chars and at least 3 characters long
- whether the fnv1a_64-hash of our type name is in the `INTERNAL_WHITELIST_HASHCODES` array (it is not)
- whether the fnv1a_64-hash of our type name is in the `internalDenyHashCodes` array (it is not)
- Depending on which configuration flags were enabled it would also check against `denyHashCodes` array

Remember: These steps above happen, despite not having `autoType` enabled.

Going forward the `createException` method of the `ThrowableDeserializer` class tries to instantiate the exception using three different constructors in this order:

```
if (causeConstructor != null) {
        return (Throwable) causeConstructor.newInstance(message, cause);
}

if (messageConstructor != null) {
        return (Throwable) messageConstructor.newInstance(message);
}

if (defaultConstructor != null) {
        return (Throwable) defaultConstructor.newInstance();
}
```

At the end with have an instantiated exception on that we can call getters and setters.

The Selenium gadget

Another exception gadget payload, that can be found in the learnjavabug GitHub repository of threedr3am is using the `WebDriverException` of Selenium. This payload could be used to leak some system information. But not that easily: First of all, it needs a web application that somehow reflects the input data and secondly it requires Selenium on the classpath, which should rarely be the case. (Selenium is mostly used for integration tests and should only be on the classpath that is used for testing.)

A simple version of that payload looks like this (note the $ref):

```
{"content": {"$ref":"$x.systemInformation"}, "x": {"@type":"java.lang.Exception","@type":"org.openqa.selenium.WebDriv
```

If the web application reflects the value of the `content` property somewhere, system information such as the following could be “leaked”:

```
host: 'detonation-chamber-20', ip: '127.0.1.1', os.name: 'Linux', os.arch: 'amd64', os.version: '5.4.0-40-generic', java.vers
```

Not that interesting information to be honest, but might be interesting if you want to detect if older versions of Fastjson are in use (requires the vulnerable web application to have Selenium on their classpath).

Searching for more gadgets using CodeQL and LGTM

Instead of searching for vulnerabilities using GitHub Security Lab’s amazing CodeQL, I thought it would be interesting to leverage CodeQL on LGTM as a tool to find additional Exception gadgets using a CodeQL query similar to this:

```
import java

from Class clazz, Method method
where
    clazz.getASourceSupertype*() instanceof TypeException
    and
    method = clazz.getAMethod() and
    method.getName().matches("get%")
    // and ...
select clazz, method
```

I ran an extended version of this query against some popular Java libraries, but did not find any interesting gadgets (due to the nature of these gadgets this was expected). However, CodeQL in combination with LGTM is definitively a comfortable way of searching for gadget candidates.

Detection with a simple gadget

It turns out that for detection it might be enough to misuse the `getStackTrace` method implemented on `Throwable`. In that case any Exception that somehow inherits from `Throwable` would do (e.g. java.lang.RuntimeException):

```
{"content": {"$ref":"$x.stackTrace"}, "x": {"@type":"java.lang.Exception","@type":"java.lang.RuntimeException"}}
```

This detection method also requires that an attribute (like `content`) is reflected somewhere.

The safe mode

With version 1.2.68 the Fastjson developers introduced the so-called safe mode. One way to turn on safe mode is to call `setSafeMode` with `true` on the global config instance:

```
ParserConfig.getGlobalInstance().setSafeMode(true);
```

The check for the safe mode flag takes place almost at the top of the already mentioned `checkAutoType` method in the `ParserConfig` class and throws an exception when Fastjson wants to instantiate an arbitrary type. However, safe mode is not enabled by default...

Closing thoughts

Basically Fastjson looks like a gift to the information security world that will keep on giving...

I didn't even have the time to look into other interesting features of Fastjson like reference or JSONPath/Regex support, so there's probably much more interesting stuff in the hiding. Now while Fastjson might seem like an extremely powerful library, it's probably too powerful for its own good. (Or at least for the developers using it.)

Should you use this library for handling user input? Probably not. Consider using JSON libraries with less features, that prevent you from shooting into your own foot. Like Gson.
