ADALOGICS

Jackson Data* Security Audit

Security Audit Report of: Jackson-dataformats-binary, Jackson-dataformats-text, Jackson-dataformat-xml, Jackson-datatype-joda, Jackson-datatypes-collections

"Arthur" Sheung Chi Chan, Adam Korczynski, David Korczynski

2024-01-10

About Ada Logics

Ada Logics is a software security company founded in Oxford, UK, 2018 and is now based in London. We are a team of dedicated, pragmatic security engineers and security researchers that work hands-on with code auditing, security automation and security tooling.

We are committed open source contributors and we routinely contribute to state of the art security tooling in the fuzzing domain such as advanced fuzzing tools like Fuzz Introspector and continuous fuzzing with OSS-Fuzz. For example, we have contributed to fuzzing of hundreds of open source projects by way of OSS-Fuzz. We regularly perform security audits of open source software and make our reports publicly available with findings and fixes, and we have audited many of the most widely used cloud native applications.

Ada Logics contributes to solving the challenge of securing the software supply-chain. To this end, we develop the tooling and infrastructure needed for ensuring a secure software development lifecycle, and we deploy these tools to critical software packages. On the tooling and infrastructure side, we contribute to projects such as the OpenSSF Scorecard project as well as the Sigstore projects like SLSA and Cosign.

Ada Logics helps some of the most exposed organisations secure their software, analyse their code and increase security automation and assurance, and if you would like to consider working with us please reach out to us via our website.

We write about our work on our blog. You can also follow Ada Logics on Linkedin, Twitter and Youtube.

Ada Logics ltd
71-75 Shelton Street,
WC2H 9JQ London,
United Kingdom

Contents

Project dashboard

Contact	Role	Organisation	Email
Adam Korczynski	Auditor	Ada Logics Ltd	adam@adalogics.com
"Arthur" Sheung Chi Chan	Auditor	Ada Logics Ltd	arthur.chan@adalogics.com
David Korczynski	Auditor	Ada Logics Ltd	david@adalogics.com
Amir Montazery	Facilitator	OSTIF	amir@ostif.org
Derek Zimmer	Facilitator	OSTIF	derek@ostif.org
Helen Woeste	Facilitator	OSTIF	helen@ostif.org

Executive summary

Ada Logics conducted a security audit of Jackson at the end of November and December 2023. The goal of the audit was to perform a holistic security assessment of several Jackson projects with a particular focus on its continuous fuzzing by way of OSS-Fuzz. The audit was facilitated by the Open Source Technology Improvement Fund (OSTIF) and funded by the Sovereign Tech Fund.

The audit was focused on the Jackson projects:

- Jackson-datatypes-collections
- Jackson-datatype-joda
- Jackson-dataformat-xml
- jackson-dataformats-text
- Jackson-dataformats-binary

We performed the following tasks for each of these projects:

- Developed a threat model
- Performed a manual audit of the code
- Developed and extended the continuous fuzzing set-up

In summary, during the engagement we:

- Developed threat models for each of the five modules
- Added 1 new OSS-Fuzz project and extended 4 existing OSS-Fuzz projects
- Created 26 new fuzzers for the Jackson projects
- Performed manual auditing of each of the codebases
- Found and reported 19 issues in the Jackson projects, including 4 of moderate security severity
- Submitted patches for 11 of the issues found

Threat model

In general, the Jackson library provides serialisation of different data types to JSON (and additional data format) and vice versa. The core Jackson library only supports serialisation from general JDK objects, primitives and interfaces to JSON format string and deserialisation from JSON format string to those objects.

Jackson-datatype*

The Jackson-datatype* libraries provide additional data types on top of the general JDK objects for the serialisation and deserialisation process. The basic serialisation and deserialisation are provided by the core Jackson databind project where specific handling of those additional datatypes is done in those jackson-datatype* libraries. Serialisation and deserialisation of different objects to and from JSON string may contain invalid, corrupted or malicious contents because sometimes it is impossible to validate before the process. For example, the deserialisation of a JSON String to an Eclipse Collections object holding a bunch of String values could accidentally contain extra fields or unexpected control characters of JSON which makes the deserialisation process halt splitting fields in the wrong location. With specially crafted input, the process could result in injection or remote code execution if the specific types of the serialised JSON string are not specifically defined.

Besides injection attacks, Denial-of-Service is another possible attack that could target the serialisation and deserialisation of the supported data types. JSON strings have a standard format with certain open and closed characters. Deserialisation of those data requires matching pairs of open and closed characters and wrong order in corrupted or malicious data could create recursive loops in the process and cause Out-Of-Memory errors or result in large time and resource consumptions. These problems could crash the applications if not handled and will affect the applications that are using these libraries. In addition, some data types, like `ImmutableMap` of Eclipse Collection, may used to store an unlimited number of data. Pushing a large enough `ImmutableMap` object may consume a high amount of time and resources during the serialisation process. Also, if a long enough JSON string with too many items is used, it could cause the same effect during the deserialisation process. Attackers may target these serialisation and deserialisation processes with invalid input to attempt to crash the applications. This results in Denial-of-Service attacks.

Components

Jackson-datatypes-collection provides support on four collection-type object libraries for its core JSON (and additional data format) parsing and generating features.

Libraries	Description and origin
Eclipse Collections	This package provides parsing and serializing of Eclipse Collections objects (https://github.com/eclipse/eclipse-collections) to JSON string or other supported string or binary data types.
P Collections	This package provides parsing and serializing of P Collections objects (https://github.com/hrldcpr/pcollections) to JSON string or other supported string or binary data types.
HPPC	This package provides parsing and serializing of HPPC objects (https://github.com/carrotsearch/hppc) to JSON string or other supported string or binary data types.
Guava	This package provides parsing and serializing of Guava objects (https://github.com/google/guava) to JSON string or other supported string or binary data types.

Jackson-datatype-joda provides support on joda libraries for its core JSON (and additional data format) parsing and generating features.

Libraries	Description and origin
Joda	This package provides parsing and serializing of Joda Time objects (https://github.com/JodaOrg/joda-time) to JSON string or other supported string or binary data types.

Threat Actor

The jackson-datatype* library is aimed to add more established data type support to its core JSON (and additional data format) parsing and generating features. Thus the threat actors should consider projects that adopt jackson-datatype* libraries for datatype serialization and parsing purposes into supporting data format (JSON, XML, or else).

Actors	Description	Level of trust
Attackers targeting the applications that adopt the library	Attackers could abuse methods with invalid or malicious data on the jackson-datatype* library and affect process execution or steal information from the applications or the executing environment	Low
User of applications that adopt the library	Users that are using the applications which have adopted the library could pass in some invalid data accidentally or be affected by malicious crashing or attack redirection from attackers	Low
Admin of the running environment of applications that adopt the library	Users that can affect, manage or control the classpath and environment of the applications that adopt the library.	High
Other users of the running environment of applications that adopt the library	Other users that can access resources or other process execution of the running environment of applications that adopt the library.	Medium

Attack vectors

Jackson-datatype* is not meant to be running as a standalone application. Thus the attack vectors should consider how a threat actor could attack the applications through the jackson-datatype* library by serialising and deserialising the data types supported by these libraries to JSON (or other supported data types of the Jackson package).

Attack vectors	Description
Input contains special characters or malicious input	Some of the data types could be sensitive to special control characters which behave differently if some of them are included in the serialised input. An attacker could abuse those vulnerable data types with malicious input which are directly passed as JSON string to the Jackson-datatype* library by the applications without further checking or validating. This creates a possible integrity problem and could cause code injection problems.
Input that is too long	JSON string requires a strict set of open and closed characters and it could also take in a long list of array elements. Processing long or invalid input could take up a long time and a high amount of memory to serialise and deserialise. This could cause Denial-of-service or possibly open up a long enough window for Race Conditions or repeat attacks.
Malicious serialised input	JSON string (and other additional data types adopted from other Jackson-dataformat* libraries) are general long sets of input in string or binary format. Deserialising random string or binary without strict control of possible types and class casting could result in remote code execution if vulnerable classes exist in the execution environment of the Jackson library. As some of the data types supported by these Jackson-datatype* are meant to be a wrapper of some established formats or accept generic object types, thus they are more vulnerable to remote code execution through uncontrolled or unchecked deserialisation process because the real data stored as elements in those data types can be any object. Attackers could inject vulnerable objects as variables or members of those supported data types and point the deserialisation of those objects towards illegal command executions. This results in unexpected remote code executions.

Attacker objectives

Attackers aim to use the Jackson-datatype* as the attack vectors for attacking the applications that adopt the Jackson core library enabling those extra data types supported by those Jackson-datatype* libraries.

Injection and remote code execution The Jackson-datatype* libraries mainly provide additional data types for the core Jackson serialisation and deserialisation to and from JSON (and other data types). As those data types generally wrap some of the existing classes in the Java class paths, the deserialisation of those random JSON (and other support string and binary data types) could trigger

unexpected operations because the deserialisation types are not strictly controlled. This could affect the file system and the execution environment outside of the expected path location. That could affect other services running in the same environment or even leak information about the environment and other sensitive data that could be stored in it. Even worse, it could trigger remote code execution if the serialised input contains malicious commands and the classpath of the execution environment is polluted.

Denial-of-Service Reading or writing a large set of input or input containing invalid or unexpected characters could result in an Exception thrown. If no exception handling or data checking is enforced, these exceptions could be thrown from the library to the applications using the library which results in the crashing of applications. This creates possible Denial-of-Service if the application is designed for long-term running.

Jackson-dataformat*

The Jackson-dataformat* libraries provide additional serialised text and binary data formats as an alternative to the core JSON string format for the serialisation and deserialisation of objects or types supported by the Jackson library. The basic serialisation and deserialisation are provided by the core Jackson databind project where specific handling to create those additional data formats are supported in these libraries. Serialisation and deserialisation of different objects to and from those string and binary data may contain invalid, corrupted or malicious contents because sometimes it is impossible to validate before the process. For example, the serialisation of a CSV String to an ArrayList could accidentally contain extra fields or unexpected control characters, like commas or semi-colon which makes the deserialisation process split fields in the wrong location. Those values may also contain macro commands With specially crafted input, the process could result in injection if the resulting CSV format is opened in a macro-enabled reader.

Besides injection attacks, Denial-of-Service is another possible attack that could target the serialisation and deserialisation to and from the supported data formats. Some of the supported data formats, like YAML, follow a standard schema with certain open and closed characters. Deserialisation of those data requires matching pairs of open and closed characters and wrong order in corrupted or malicious data could create recursive loops in the process and cause Out-Of-Memory errors or result in large time and resource consumptions. These problems could crash the applications if not handled and will affect the applications that are using these libraries. In addition, some data formats, like YAML, may contain a high depth level. Deserialising those high-depth YAML strings may consume a high amount of time and resources during the deserialisation process. Attackers may target these serialisation and deserialisation processes with invalid input to attempt to crash the applications. This results in Denial-of-Service attacks.

Components

Jackson-dataformat-xml provides support for parsing and serializing different data types to XML as an alternative to JSON.

Libraries	Description and origin
XML (Woodstox) (Default)	This package provides parsing and serializing of Jackson-supported data types to Woodstox XML format (https://github.com/FasterXML/woodstox) instead of core JSON.
XML (SJSXP)	This package provides parsing and serializing of Jackson-supported data types to SJSXP XML format (https://javadoc.io/doc/com.sun.xml.stream/sjsxp/latest/index.html) instead of core JSON.

Jackson-dataformats-text provides support for parsing and serializing different data types to four different textual data formats as an alternative to JSON.

Libraries	Description and origin
CSV	This package provides parsing and serializing of Jackson-supported data types to CSV format (http://en.wikipedia.org/wiki/Comma-separated_values) instead of core JSON.
Properties	This package provides parsing and serializing of Jackson-supported data types to Java Properties format (https://en.wikipedia.org/wiki/.properties) instead of core JSON.
TOML	This package provides parsing and serializing of Jackson-supported data types to TOML format (https://github.com/toml-lang/toml) instead of core JSON.
YAML	This package provides parsing and serializing of Jackson-supported data types to Snake YAML format (https://github.com/snakeyaml/snakeyaml) instead of core JSON.

Jackson-dataformats-binary provides support for parsing and serializing different data types to five different binary data formats as an alternative to JSON.

Libraries	Description and origin
Avro	This package provides parsing and serializing of Jackson-supported data types to Apache Avro format (https://github.com/apache/avro) instead of core JSON.
CBOR	This package provides parsing and serializing of Jackson-supported data types to CBOR format (https://www.rfc-editor.org/info/rfc7049) instead of core JSON.
Ion	This package provides parsing and serializing of Jackson-supported data types to Amazon ION format (https://github.com/amazon-ion/ion-java) instead of core JSON.
Protobuf	This package provides parsing and serializing of Jackson-supported data types to Google Protobuf format (https://github.com/protocolbuffers/protobuf) instead of core JSON.
Smile	This package provides parsing and serializing of Jackson-supported data types to Smile format (https://github.com/FasterXML/smile-format-specification) instead of core JSON.

Threat Actor

The jackson-dataformat* library is aimed to provide parsing and generating jackson-supported data types to different formats as an alternative to the core JSON. Thus the threat actors should consider projects that adopt jackson-dataformat* libraries for datatype serialization and parsing purposes.

Actors	Description	Level of trust
Attackers targeting the applications that adopt the library	Attackers could abuse some vulnerable serialisation and deserialisation methods with invalid or malicious data on the jackson-dataformat* library and affect process execution or steal information from the applications or the executing environment	Low
User of applications that adopt the library	Users that are using the applications which have adopted the library could pass in some invalid data accidentally or be affected by malicious crashing or attack redirection from attackers	Low

Actors	Description	Level of trust
Admin of the running environment of applications that adopt the library	Users that can affect, manage or control the classpath and environment of the applications that adopt the library.	High
Other users of the running environment of applications that adopt the library	Other users that can access resources or other process execution of the running environment of applications that adopt the library.	Medium

Attack vectors

Jackson-dataformat* is not meant to be running as a standalone application. Thus the attack vectors should consider how a threat actor could attack the applications through the jackson-dataformat* library by serialising and deserialising the data types supported by these libraries to JSON (or other supported data types of the Jackson package).

Attack vectors	Description
Input contains special characters or malicious input	Some of the data types could be sensitive to special control characters which behave differently if some of them are included in the serialised input. An attacker could abuse those vulnerable data types with malicious input which are directly passed as Jackson-dataformat* library supported string or binary to any objects or types by the applications without further checking or validating. This creates a possible integrity problem and could cause code injection problems. This is especially vulnerable for some data formats, like CSV, which does not have strict control of special characters.

Attack vectors	Description
Input that is too long	Some string or binary input of Jackson-dataformat* supported format requires a strict set of open and closed characters and it could also take in a long input or input with a high depth level. Processing long or invalid input could take up a long time and a high amount of memory to serialise or deserialise. This could cause Denial-of-service or possibly open up a long enough window for Race Conditions or repeat attacks.
Malicious serialised input	JSON string (and other additional data types adopted from other Jackson-dataformat* libraries) are general long sets of input in string or binary format. Deserialising random string or binary without strict control of possible types and class casting could result in remote code execution if vulnerable classes exist in the execution environment of the Jackson library. As some of the data types supported by Jackson are meant to be a wrapper of some established formats or accept generic object types (Java Collections objects), thus they are more vulnerable to remote code execution through uncontrolled or unchecked deserialisation processes because the real data stored as elements in those data types can be any object. Attackers could inject vulnerable objects as variables or members of those supported data types and point the deserialisation of those objects towards illegal command executions. This results in unexpected remote code executions.

Attacker objectives

Attackers aim to use the Jackson-dataformat* as the attack vectors for attacking the applications that adopt the Jackson core library enabling serialising from Java objects to those supported text or binary data formats.

Injection and remote code execution: The Jackson-dataformat* libraries mainly provide additional data serialisation formats as an alternative to the core JSON string format. As there exist some Java data types, like Java Collections objects, take in generic types of objects in the Java class paths, the deserialisation of those random text and binary input) could trigger unexpected operations because the deserialisation types are not strictly controlled. Some support text and binary input formats, like CSV, don't have strict control of illegal or controlled characters. That could affect other services running in the same environment or even leak information about the environment and other sensitive data that could be stored in it. Even worse, it could trigger remote code execution if the serialised input contains malicious commands and the classpath of the execution environment is polluted.

Denial-of-Service: Reading or writing a large set of input or input containing invalid or unexpected characters could result in an Exception thrown. If no exception handling or data checking is enforced, these exceptions could be thrown from the library to the applications using the library which results in the crashing of applications. This creates possible Denial-of-Service if the application is designed for long-term running.

Manual audit and static analysis

A manual code review has been done for all five target projects. The Maven build file `pom.xml` configuration for each project has also been gone through to check for vulnerabilities in dependencies and configuration settings. Most of the live and non-deprecated Java code in the base /src/main directory has been gone through. Those unit test classes in the /src/test directory have been ignored. The following list shows a generic list of items that have been looked for in Java code during the manual code auditing process.

Issues found by manual audit

#	ID	Title	Severity	Fixed
2	ADA-JACKSON-BINARY-2023-2	Vulnerable version of the Avro dependency is used	Moderate	No
12	ADA-JACKSON-COLLECTIONS-2023-2	Infinite recursive loop in GuavaOptionalDeserializer	Moderate	No
13	ADA-JACKSON-COLLECTIONS-2023-3	Vulnerable version of the Guava dependency is used	Informational	No
14	ADA-JACKSON-JODA-2023-1	Direct comparison of Boolean object in JacksonJodaDateFormat	Low	No
15	ADA-JACKSON-JODA-2023-2	Unnecessary auto-boxing/unboxing in IntervalDeserializer	Informational	No
16	ADA-JACKSON-TEXT-2023-1	Unused conditional check in CsvDecoder	Informational	No

Besides manual audit, we also have run three static analysis tools, `infer` (https://github.com/facebook/infer), `findsecbug` (https://find-sec-bugs.github.io/) and `semgrep` (https://semgrep.dev/) and they are run on all three projects.

`infer` generates around 50 possible vulnerabilities. After a detailed analysis of the items, it is found that more than 40 of them are located in the unit testing package, thus they are ignored as they do not affect the main functionality. Only 8 issues is found for the source package of the five projects, and most of them are classified as possible null referencing problems. Although some of them could be triggered if invalid data has been provided, it is believed that those invalid inputs are all checked,

handled or filtered in different locations before reaching the problematic statement that could cause a null dereferencing problem. Thus they are all considered as False Positive cases.

For `semgrep`, it does generates around 10 possible vulnerabilities. After a detailed analysis of the items, it is believed that all of them are false positive or informational items, thus they are ignored.

For `findsecbug`, it does discover one issue and it is summarized in the issue list.

Issues found by findsecbug

#	ID	Title	Severity	Fixed
19	ADA-JACKSON-XML-2023-2	XML External Entity vulnerability in XMLFactory	Moderate	No

Fuzzers

Jackson-datatypes-collections

The Jackson Datatypes Collections library adds more established data type support to its core JSON (and additional data format) parsing and generating features.

Fuzzers

Each of the fuzzers targets one of the four established data types supported by the Jackson Datatypes Collections library and performs serialisation or deserialisation of that type. The fuzzers provide random string, byte array and other primitives and collections objects as input for creating serialised objects of the chosen type for fuzzing the deserialisation methods or creating an object of the chosen type for fuzzing the serialisation methods. The fuzzers can be found in https://github.com/google/oss-fuzz/tree/bcb9400cf88be8ee660feeeca6416a8f3b043d96/projects/jackson-datatypes-collections.

Newly added fuzzers	Description
EclipseCollectionsDeserializerFuzzer	This fuzzer creates random inputs and invokes the deserialisation method to fuzz the deserialisation process from the random input to different objects in the EclipseCollections datatype package.
EclipseCollectionsSerializerFuzzer	This fuzzer creates different objects in the EclipseCollections datatype package with random data and invokes the serialisation method to fuzz the serialisation process from different objects in the EclipseCollections datatype package to JSON format.
GuavaDeserializerFuzzer	This fuzzer creates random inputs and invokes the deserialisation method to fuzz the deserialisation process from the random input to different objects in the Guava datatype package.
GuavaSerializerFuzzer	This fuzzer creates different objects in the Guava datatype package with random data and invokes the serialisation method to fuzz the serialisation process from different objects in the Guava datatype package to JSON format.

Newly added fuzzers	Description
HppcDeserializerFuzzer	This fuzzer creates random inputs and invokes the deserialisation method to fuzz the deserialisation process from the random input to different objects in the HPPC datatype package.
HppcSerializerFuzzer	This fuzzer creates different objects in the HPPC datatype package with random data and invokes the serialisation method to fuzz the serialisation process from different objects in the HPPC datatype package to JSON format.
PCollectionsFuzzer	This fuzzer creates random inputs and different objects in the PCollections datatype package with random data. The fuzzer then invokes serialisation and deserialisation methods to fuzz the serialisation and deserialisation process between random inputs (assumed to be JSON) and different objects in the PCollections datatype package.

Coverage

This project is a new implementation in OSS-Fuzz, figure 1 shows the Jacoco fuzzers coverage report for the Jackson Datatypes Collections project for the new implementation of OSS-Fuzz.

Most of the classes and methods are covered, with exceptions for those methods in abstract classes and interfaces and those helper methods which does not take any input, including getters and setters methods.

Jackson Datatypes Collections library provides additional support for serialisation and deserialisation of four different groups of established data types to and from JSON. The supports for these datatypes are built on top of the base Jackson Databind module. Thus many of the serialisation and deserialisation processes are wrappers for the existing Jackson Databinding module. For example, the Eclipse Collections are an extension of the General Java collections package which could store generic objects like primitives and String. The serialisation and deserialisation for those underlying generic objects are done by the base Jackson Databind modules and thus the `eclipsecollections` package (https://storage.googleapis.com/oss-fuzz-coverage/jackson-datatypes-collections/reports/20231219/linux/com.fasterxml.jackson.datatype.eclipsecollections.deser.map/index.html) in this library contains many classes with low cyclomatic complexity, many of them contain many one-liner wrappers for invoking different superclasses methods in the base Jackson Databind module. These methods and classes are therefore not fuzzworthy.

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.datatype.eclipsecollections.ser.map		40%		40%	220	302	665	956	179	257	15	76
com.fasterxml.jackson.datatype.guava.ser		43%		30%	188	230	284	567	50	84	6	11
com.fasterxml.jackson.datatype.hppc.ser		41%		28%	98	142	162	261	60	94	4	19
com.fasterxml.jackson.datatype.eclipsecollections.deser.map		82%		46%	92	445	116	848	73	417	2	88
com.fasterxml.jackson.datatype.guava.deser		64%		45%	102	190	132	377	40	111	1	22
com.fasterxml.jackson.datatype.eclipsecollections.deser.pair		69%		52%	58	135	87	285	38	114	5	8
com.fasterxml.jackson.datatype.eclipsecollections		70%		37%	59	88	59	258	11	24	2	7
com.fasterxml.jackson.datatype.guava.deser.multimap		47%		43%	28	42	56	116	4	12	0	1
com.fasterxml.jackson.datatype.guava		73%		57%	54	114	46	194	9	35	0	7
com.fasterxml.jackson.datatype.eclipsecollections.deser		65%		57%	18	48	34	105	8	28	1	10
com.fasterxml.jackson.datatype.eclipsecollections.deser.set		58%		n/a	32	80	36	100	32	80	4	20
com.fasterxml.jackson.datatype.eclipsecollections.deser.bag		58%		n/a	32	80	36	100	32	80	4	20
com.fasterxml.jackson.datatype.guava.deser.util		38%		50%	15	23	23	43	14	22	0	3
com.fasterxml.jackson.datatype.pcollections.deser		74%		52%	23	57	26	116	5	32	0	8
com.fasterxml.jackson.datatype.eclipsecollections.deser.list		62%		n/a	28	76	31	95	28	76	3	19
com.fasterxml.jackson.datatype.hppc.deser		65%		52%	17	41	23	77	6	22	0	7
com.fasterxml.jackson.datatype.eclipsecollections.ser		83%		70%	18	57	17	113	12	42	0	9
com.fasterxml.jackson.datatype.guava.deser.cache		73%		54%	11	19	10	49	1	7	0	1
default		98%		95%	10	124	11	358	7	73	0	54
com.fasterxml.jackson.datatype.hppc		62%		50%	5	10	6	18	4	9	1	4
com.fasterxml.jackson.datatype.pcollections		84%		66%	9	21	6	31	4	12	0	3
com.fasterxml.jackson.datatype.guava.deser.multimap.set		100%		n/a	0	8	0	12	0	8	0	2
com.fasterxml.jackson.datatype.guava.deser.multimap.list		100%		n/a	0	8	0	12	0	8	0	2
Total	7,251 of 20,062	63%	731 of 1,346	45%	1,117	2,340	1,866	5,091	617	1,647	48	401

Figure 1: Fuzzer Coverage for Jackson Datatypes Collections as at 9th January 2024

As a whole, there is an estimated 10% of methods have very low cyclomatic complexity (5 or less) which is therefore not worth to fuzz.

Upstream fixes

https://github.com/FasterXML/jackson-datatypes-collections/pull/125

https://github.com/FasterXML/jackson-datatypes-collections/pull/139

Issues found by fuzzers

#	ID	Title	Severity	Fixed
11	ADA-JACKSON-COLLECTIONS-2023-1	Unexpected NullPointerException when deserializing	Low	Yes

Jackson-datatype-joda

The Jackson Datatype Joda library adds established Joda data type support to its core JSON (and additional data format) parsing and generating features.

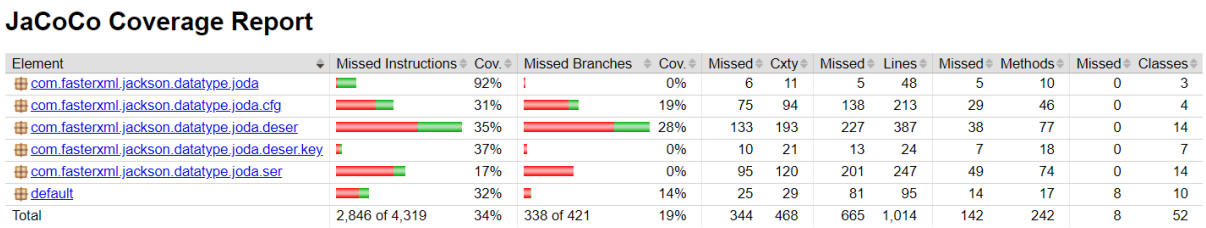
Fuzzers

The fuzzers target the established Joda data type and perform serialisation or deserialisation of it. The fuzzers provide random string, byte array and other primitives and collections objects as input for creating serialised objects of the Joda type for fuzzing the deserialisation methods or creating an object of the Joda type for fuzzing the serialisation methods. The fuzzers can be found in https://github.com/google/oss-fuzz/tree/bcb9400cf88be8ee660feeeca6416a8f3b043d96/projects/jackson-datatype-joda.

Newly added fuzzers	Description
JodaDeserializerFuzzer	This fuzzer creates random inputs and invokes the deserialisation method to fuzz the deserialisation process from the random input to different objects in the Joda datatype package.
JodaSerializerFuzzer	This fuzzer creates different objects in the Joda datatype package with random data and invokes the serialisation method to fuzz the serialisation process from different objects in the Joda datatype package to JSON format.

Coverage

Figure 2 shows the Jacoco fuzzers coverage report for the Jackson Datatype Joda project before the new fuzzers implementation to OSS-Fuzz.



JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.datatype.joda		92%		0%	6	11	5	48	5	10	0	3
com.fasterxml.jackson.datatype.joda.cfg		31%		19%	75	94	138	213	29	46	0	4
com.fasterxml.jackson.datatype.joda.deser		35%		28%	133	193	227	387	38	77	0	14
com.fasterxml.jackson.datatype.joda.deser.key		37%		0%	10	21	13	24	7	18	0	7
com.fasterxml.jackson.datatype.joda.ser		17%		0%	95	120	201	247	49	74	0	14
default		32%		14%	25	29	81	95	14	17	8	10
Total	2,846 of 4,319	34%	338 of 421	19%	344	468	665	1,014	142	242	8	52

Figure 2: Fuzzer Coverage for Jackson Datatype Joda as of 1st December 2023

Figure 3 shows the Jacoco fuzzers coverage report for the Jackson Datatype Joda project after the new fuzzers implementation to OSS-Fuzz.

Figure 4 shows the coverage and fuzzer difference during the audit period from the Fuzz-Introspector report. Fuzz-Intorspector is a tool that aids fuzzer developers in understanding the fuzzer's performance and identifying any potential blockers for fuzzer enhancement.

Most of the classes and methods are covered, with exceptions for those methods in abstract classes and interfaces and those helper methods which does not take any input, including getters and setters methods.

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.datatype.joda.deser		70%		64%	72	193	94	387	15	77	0	14
com.fasterxml.jackson.datatype.joda.ser		47%		30%	76	120	128	247	37	74	0	14
com.fasterxml.jackson.datatype.joda.cfg		54%		38%	62	94	88	213	20	46	0	4
com.fasterxml.jackson.datatype.joda.deser.key		50%		33%	9	21	10	24	6	18	0	7
default		91%		85%	9	32	13	101	7	20	4	12
com.fasterxml.jackson.datatype.joda		92%		0%	6	11	5	48	5	10	0	3
Total	1,490 of 4,332	65%	204 of 421	51%	234	471	338	1,020	90	245	4	54

Figure 3: Fuzzer Coverage for Jackson Datatype Joda as at 9th January 2024

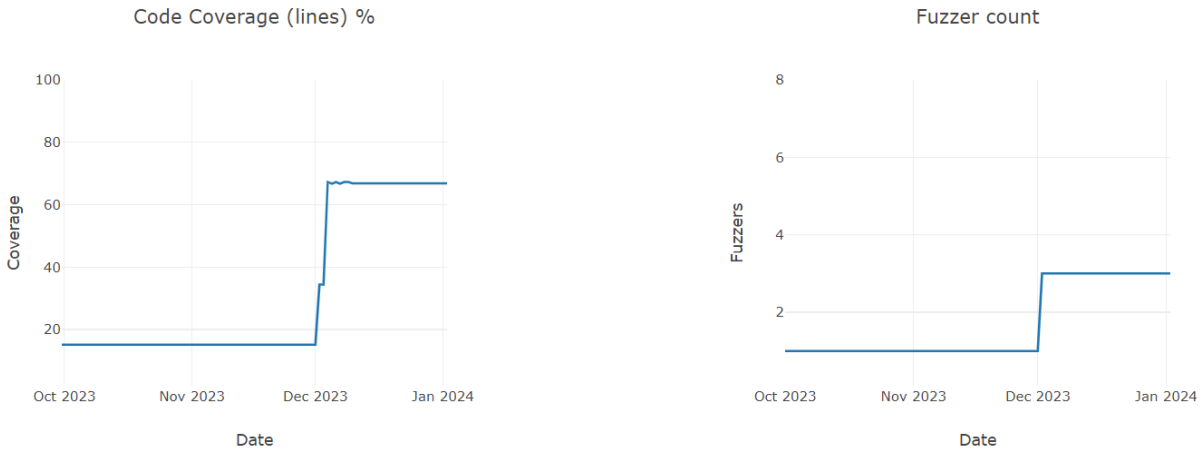


Figure 4: Fuzz-Introspector report for Jackson Datatype Joda

Jackson Datatype Joda library provides additional support for serialisation and deserialisation between Joda type and JSON. The supports for the Joda type are built on top of the base Jackson Databind module. For example, if the Joda type is wrapped in Java bean or collection objects. Thus many of the serialisation and deserialisation processes are wrappers for the existing Jackson Databinding module. The serialisation and deserialisation for those underlying generic objects are done by the base Jackson Databind modules and thus this library (https://storage.googleapis.com/oss-fuzz-coverage/jackson-datatype-joda/reports/20231219/linux/com.fasterxml.jackson.datatype.joda.deser/index.html) contains many classes with low cyclomatic complexity, many of them contain many one-liner wrappers for invoking different superclasses methods in the base Jackson Databind module. These methods and classes are therefore not fuzzworthy.

As a whole, there is an estimated 10% of methods have very low cyclomatic complexity (5 or less) which is therefore not worth to fuzz.

Upstream fixes

No upstream fixes.

Jackson-dataformat-xml

The Jackson Dataformat XML library adds support to the XML data format for the Jackson library. It allows serialising and deserialising between all supported data types and XML format in addition to the core JSON format.

Fuzzers

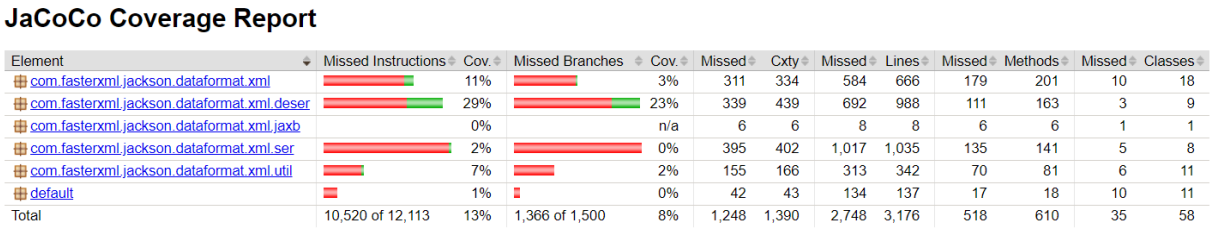
Each of the fuzzers targets random data types and performs serialisation or deserialisation of those types to XML format. The fuzzers provide random string, byte array and other primitives and collections objects as input fuzzing the deserialisation methods or creating random objects supported by the Jackson library for fuzzing the serialisation methods. The fuzzers can be found in https://github.com/google/oss-fuzz/tree/bcb9400cf88be8ee660feeeca6416a8f3b043d96/projects/jackson-dataformat-xml.

Newly added fuzzers	Description
XmlDeserializerFuzzer	This fuzzer creates random inputs and invokes the deserialisation method to fuzz the deserialisation process from the random input (assumed to be XML) to different objects supported by the Jackson Databind library.

Newly added fuzzers	Description
XmlSerializerFuzzer	This fuzzer creates different objects supported by the Jackson Databind library with random data and invokes the serialisation method to fuzz the serialisation process from different objects supported by the Jackson Databind library to XML format.
ToXmlGeneratorFuzzer	This fuzzer creates random input to invoke and fuzz XML entity generating methods in the ToXmlGenerator class methods for the XML serialisation process.
FromXmlParserFuzzer	This fuzzer creates random input to invoke and fuzz XML entity parsing methods in the FromXmlParser class methods for the XML deserialisation process.

Coverage

Figure 5 shows the Jacoco fuzzers coverage report for the Jackson Dataformat XML project before the new fuzzers implementation to OSS-Fuzz.

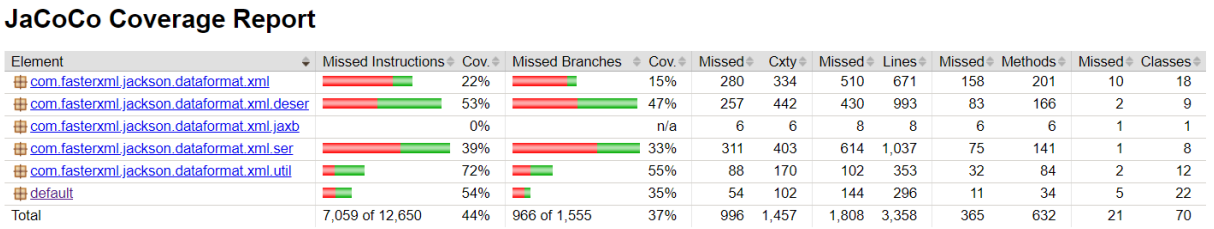


JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.dataformat.xml		11%		3%	311	334	584	666	179	201	10	18
com.fasterxml.jackson.dataformat.xml.deser		29%		23%	339	439	692	988	111	163	3	9
com.fasterxml.jackson.dataformat.xml.jaxb		0%		n/a	6	6	8	8	6	6	1	1
com.fasterxml.jackson.dataformat.xml.ser		2%		0%	395	402	1,017	1,035	135	141	5	8
com.fasterxml.jackson.dataformat.xml.util		7%		2%	155	166	313	342	70	81	6	11
default		1%		0%	42	43	134	137	17	18	10	11
Total	10,520 of 12,113	13%	1,366 of 1,500	8%	1,248	1,390	2,748	3,176	518	610	35	58

Figure 5: Fuzzer Coverage for Jackson Dataformat XML as of 1st December 2023

Figure 6 shows the Jacoco fuzzers coverage report for the Jackson Dataformat XML project after the new fuzzers implementation to OSS-Fuzz.



JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.dataformat.xml		22%		15%	280	334	510	671	158	201	10	18
com.fasterxml.jackson.dataformat.xml.deser		53%		47%	257	442	430	993	83	166	2	9
com.fasterxml.jackson.dataformat.xml.jaxb		0%		n/a	6	6	8	8	6	6	1	1
com.fasterxml.jackson.dataformat.xml.ser		39%		33%	311	403	614	1,037	75	141	1	8
com.fasterxml.jackson.dataformat.xml.util		72%		55%	88	170	102	353	32	84	2	12
default		54%		35%	54	102	144	296	11	34	5	22
Total	7,059 of 12,650	44%	966 of 1,555	37%	996	1,457	1,808	3,358	365	632	21	70

Figure 6: Fuzzer Coverage for Jackson Dataformat XML as at 9th January 2024

Figure 7 shows the coverage and fuzzer difference during the audit period from the Fuzz-Introspector

report. Fuzz-Intorspector is a tool that aids fuzzer developers in understanding the fuzzer's performance and identifying any potential blockers for fuzzer enhancement.

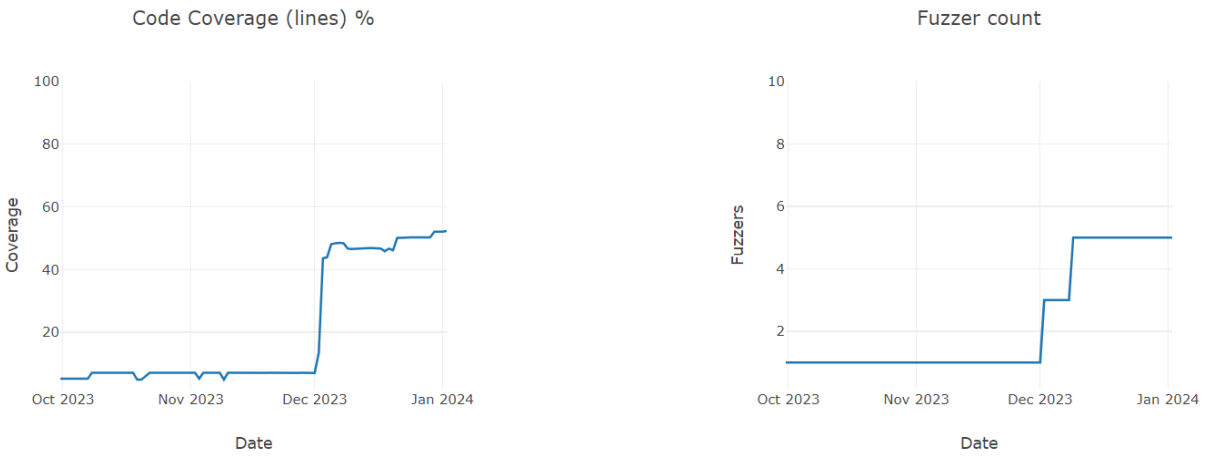


Figure 7: Fuzz-Introspector report for Jackson Dataformat XML

Most of the classes and methods are covered, with exceptions for those methods in abstract classes and interfaces and those helper methods which does not take any input, including getters and setters methods.

Jackson Dataformat XML library provides an additional serialised format of XML in addition to the core JSON format. It mainly provides serialisation and deserialisation between Jackson-supported type and XML data and is built on top of the base Jackson Databind module as additional serialised modules. Most of the serialisation and deserialisation processes are wrappers for the existing Jackson Databinding module except for those steps transforming to and from XML string. The serialisation and deserialisation for those underlying generic objects are done by the base Jackson Databind modules and thus this library (https://storage.googleapis.com/oss-fuzz-coverage/jackson-dataformat-xml/reports/20231219/linux/com.fasterxml.jackson.dataformat.xml.deser/index.html and https://storage.googleapis.com/oss-fuzz-coverage/jackson-dataformat-xml/reports/20231219/linux/com.fasterxml.jackson.dataformat.xml.ser/index.html) contains many classes with low cyclomatic complexity, many of them contain many one-liner wrappers for invoking different superclasses methods in the base Jackson Databind module. These methods and classes are therefore not fuzzworthy.

As a whole, there is an estimated 15% of methods have very low cyclomatic complexity (5 or less) which is therefore not worth to fuzz.

Last but not least, the upstream libraries (Woodstox, SJSXP or else) enforce strict input checkers and thus the fuzzers need more time to explore different branches because many of the random inputs are denied those input checkers with exceptions thrown. The coverage is assumed to be increasing in the

coming months.

Upstream fixes

https://github.com/FasterXML/jackson-dataformat-xml/pull/619

Issues found by fuzzers

#	ID	Title	Severity	Fixed
18	ADA-JACKSON-XML-2023-1	Unexpected ArrayIndexOutOfBoundsException in XMLTokenStream with SJSXP	Low	Yes

Jackson-dataformats-text

The Jackson Dataformats Text library adds support to four different text (YAML/Java Properties/TOML/CSV) data formats for the Jackson library. It allows serialising and deserialising between all supported data types and those four text formats in addition to the core JSON format.

Fuzzers

Each of the fuzzers targets random data types and performs serialisation or deserialisation of those types to either of the four text data formats supported by this library. The fuzzers provide random string, byte array and other primitives and collections objects as input fuzzing the deserialisation methods or creating random objects supported by the Jackson library for fuzzing the serialisation methods. The fuzzers can be found in https://github.com/google/oss-fuzz/tree/bcb9400cf88be8ee660feeeca6416a8f3b043d96/projects/jackson-dataformats-text.

Newly added fuzzers	Description
DeserializerFuzzer	This fuzzer creates random inputs and invokes the deserialisation method to fuzz the deserialisation process from the random input (assumed to be one of the four supported text formats) to different objects supported by the Jackson Databind library.
SerializerFuzzer	This fuzzer creates different objects supported by the Jackson Databind library with random data and invokes the serialisation method to fuzz the serialisation process from different objects supported by the Jackson Databind library to either one of the four text formats supported by this library.

Coverage

Figure 8 shows the Jacoco fuzzers coverage report for the Jackson Dataformats Text project before the new fuzzers implementation to OSS-Fuzz.

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.dataformat.csv		9%		4%	727	776	1,331	1,495	367	409	11	18
com.fasterxml.jackson.dataformat.csv.impl		25%		24%	596	750	1,354	1,836	173	213	13	17
com.fasterxml.jackson.dataformat.javaprop		13%		9%	242	270	445	523	167	192	4	8
com.fasterxml.jackson.dataformat.javaprop.impl		0%		0%	53	53	151	151	35	35	2	2
com.fasterxml.jackson.dataformat.javaprop.io		36%		38%	64	100	165	268	27	42	2	6
com.fasterxml.jackson.dataformat.javaprop.util		69%		68%	37	97	63	230	11	36	3	9
com.fasterxml.jackson.dataformat.toml		54%		45%	411	695	728	1,547	154	255	7	22
com.fasterxml.jackson.dataformat.yaml		35%		41%	427	616	880	1,410	197	256	5	12
com.fasterxml.jackson.dataformat.yaml.snakeyaml.error		24%		0%	14	17	15	21	13	16	1	3
com.fasterxml.jackson.dataformat.yaml.util		45%		0%	30	35	37	42	8	13	0	2
default		6%		0%	59	63	199	219	19	23	10	14
Total	22,621 of 32,247	29%	2,703 of 3,784	28%	2,660	3,472	5,368	7,742	1,171	1,490	58	113

Figure 8: Fuzzer Coverage for Jackson Dataformats Text as of 1st December 2023

Figure 9 shows the Jacoco fuzzers coverage report for the Jackson Dataformats Text project after the new fuzzers implementation to OSS-Fuzz.

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.dataformat.csv		21%		13%	659	779	1,121	1,498	305	412	5	18
com.fasterxml.jackson.dataformat.csv.impl		49%		42%	490	750	946	1,836	124	213	10	17
com.fasterxml.jackson.dataformat.javaprop		35%		30%	188	273	325	526	122	195	1	8
com.fasterxml.jackson.dataformat.javaprop.impl		67%		58%	32	53	52	151	21	35	1	2
com.fasterxml.jackson.dataformat.javaprop.io		76%		72%	33	100	64	268	11	42	0	6
com.fasterxml.jackson.dataformat.javaprop.util		69%		68%	37	97	63	230	11	36	3	9
com.fasterxml.jackson.dataformat.toml		72%		64%	292	696	412	1,549	92	255	1	22
com.fasterxml.jackson.dataformat.yaml		62%		63%	306	623	490	1,421	126	259	1	12
com.fasterxml.jackson.dataformat.yaml.snakeyaml.error		24%		0%	14	17	15	21	13	16	1	3
com.fasterxml.jackson.dataformat.yaml.util		96%		82%	7	35	4	42	0	13	0	2
default		94%		83%	17	70	17	239	8	28	1	19
Total	15,369 of 32,372	52%	1,981 of 3,798	47%	2,075	3,493	3,509	7,781	833	1,504	24	118

Figure 9: Fuzzer Coverage for Jackson Dataformats Text as at 9th January 2024

Figure 10 shows the coverage and fuzzer difference during the audit period from the Fuzz-Introspector report. Fuzz-Intorspector is a tool that aids fuzzer developers in understanding the fuzzer's performance and identifying any potential blockers for fuzzer enhancement.

Most of the classes and methods are covered, with exceptions for those methods in abstract classes and interfaces and those helper methods which does not take any input, including getters and setters methods.

Jackson Dataformats Text library provides four additional serialised text data formats in addition to the core JSON format. It mainly provides serialisation and deserialisation between Jackson supported type and those four different text data formats and is built on top of the base Jackson Databind module as additional serialised modules. Most of the serialisation and deserialisation processes are wrappers

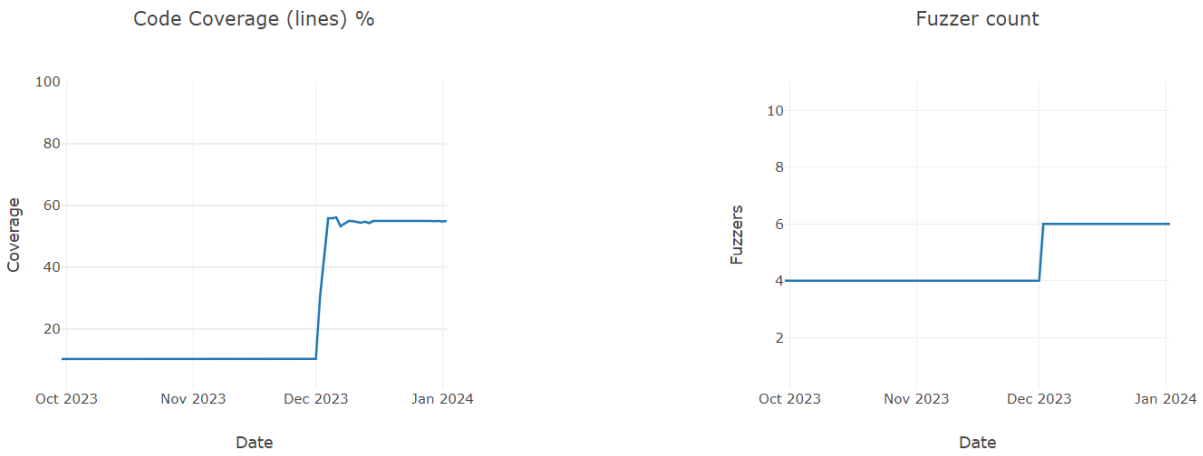


Figure 10: Fuzz-Introspector report for Jackson Dataformats Text

for the existing Jackson Databinding module except for those steps transforming to and from those supported text formats. The serialisation and deserialisation for those underlying generic objects are done by the base Jackson Databind modules and thus this library contains many classes with low cyclomatic complexity, many of them contain many one-liner wrappers for invoking different superclasses methods in the base Jackson Databind module. These methods and classes are therefore not fuzzworthy.

As a whole, there is an estimated 10% of methods have very low cyclomatic complexity (5 or less) which is therefore not worth to fuzz.

Last but not least, the upstream libraries of those text formats enforce strict input checkers and thus the fuzzers need more time to explore different branches because many of the random inputs are denied those input checkers with exceptions thrown. The coverage is assumed to be increasing in the coming months.

Upstream fixes

https://github.com/FasterXML/jackson-dataformats-text/pull/446

Issues found by fuzzers

#	ID	Title	Severity	Fixed
10	ADA-JACKSON-BINARY-2023-10	Stack out of memory in Jackson standard ThrowableDeserializer	Moderate	No

#	ID	Title	Severity	Fixed
17	ADA-JACKSON-TEXT-2023-2	Unexpected NullPointerException in YAMLParser	Low	Yes

Jackson-dataformats-binary

The Jackson Dataformats Binary library adds support to five different binary (Avro/CBOR/Ion/Protobuf/Smile) data formats for the Jackson library. It allows serialising and deserialising between all supported data types and those five binary formats in addition to the core JSON format.

Fuzzers

Each of the fuzzers targets random data types and performs serialisation or deserialisation of those types to either of the five binary data formats supported by this library. The fuzzers provide random string, byte array and other primitives and collections objects as input fuzzing the deserialisation methods or creating random objects supported by the Jackson library for fuzzing the serialisation methods. The fuzzers can be found in https://github.com/google/oss-fuzz/tree/bcb9400cf88be8ee660feeeca6416a8f3b043d96/projects/jackson-dataformats-binary.

Newly added fuzzers	Description
DeserializerFuzzer	This fuzzer creates random inputs and invokes the deserialisation method to fuzz the deserialisation process from the random input (assumed to be one of the five supported binary formats) to different objects supported by the Jackson Databind library.
SerializerFuzzer	This fuzzer creates different objects supported by the Jackson Databind library with random data and invokes the serialisation method to fuzz the serialisation process from different objects supported by the Jackson Databind library to either one of the five binary formats supported by this library.
AvroGeneratorFuzzer	This fuzzer creates random input to invoke and fuzz Avro entity generation methods in the AvroGenerator class methods for the Avro serialisation process.
AvroParserFuzzer	This fuzzer creates random input to invoke and fuzz Avro entity parsing methods in the AvroParser class methods for the Avro deserialisation process.

Newly added fuzzers	Description
CborGeneratorFuzzer	This fuzzer creates random input to invoke and fuzz CBOR entity generating methods in the CborGenerator class methods for the CBOR serialisation process.
CborParserFuzzer	This fuzzer creates random input to invoke and fuzz CBOR entity parsing methods in the CborParser class methods for the CBOR deserialisation process.
IonGeneratorFuzzer	This fuzzer creates random input to invoke and fuzz Ion entity generating methods in the IonGenerator class methods for the Ion serialisation process.
IonParserFuzzer	This fuzzer creates random input to invoke and fuzz Ion entity parsing methods in the IonParser class methods for the Ion deserialisation process.
ProtobufParserFuzzer	This fuzzer creates random input to invoke and fuzz both Protobuf entity generating methods and entity parsing methods in the ProtobufParser class for the Protobuf serialisation and deserialisation process.
SmileGeneratorFuzzer	This fuzzer creates random input to invoke and fuzz Smile entity generating methods in the SmileGenerator class methods for the Smile serialisation process.
SmileParserFuzzer	This fuzzer creates random input to invoke and fuzz Smile entity parsing methods in the SmileParser class methods for the Smile deserialisation process.

Coverage

Figure 11 shows the Jacoco fuzzers coverage report for the Jackson Dataformats Binary project before the new fuzzers implementation to OSS-Fuzz.

Figure 12 shows the Jacoco fuzzers coverage report for the Jackson Dataformats Binary project after the new fuzzers implementation to OSS-Fuzz.

Figure 13 shows the coverage and fuzzer difference during the audit period from the Fuzz-Introspector report. Fuzz-Intorspector is a tool that aids fuzzer developers in understanding the fuzzer's performance and identifying any potential blockers for fuzzer enhancement.

Most of the classes and methods are covered, with exceptions for those methods in abstract classes and interfaces and those helper methods which does not take any input, including getters and setters methods.

Jackson Dataformats Binary library provides four additional serialised text data formats in addition to the core JSON format. It mainly provides serialisation and deserialisation between Jackson supported type and those four different text data formats and is built on top of the base Jackson Databind module as additional serialised modules. Most of the serialisation and deserialisation processes are

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.dataformat.avro		0%		0%	369	369	700	700	250	250	19	19
com.fasterxml.jackson.dataformat.avro.apacheimpl		0%		0%	156	156	279	279	116	116	11	11
com.fasterxml.jackson.dataformat.avro.deser		0%		0%	761	761	1,672	1,672	362	362	60	60
com.fasterxml.jackson.dataformat.avro.jsr310		0%		n/a	1	1	14	14	1	1	1	1
com.fasterxml.jackson.dataformat.avro.jsr310.deser		0%		0%	17	17	27	27	16	16	5	5
com.fasterxml.jackson.dataformat.avro.jsr310.ser		0%		0%	20	20	43	43	16	16	4	4
com.fasterxml.jackson.dataformat.avro.schema		0%		0%	235	235	482	482	104	104	14	14
com.fasterxml.jackson.dataformat.avro.ser		0%		0%	205	205	443	443	98	98	11	11
com.fasterxml.jackson.dataformat.cbor		39%		40%	761	1,191	1,785	2,977	241	366	5	15
com.fasterxml.jackson.dataformat.cbor.databind		8%		0%	13	15	21	25	10	12	1	2
com.fasterxml.jackson.dataformat.ion		0%		0%	361	361	692	692	236	236	19	19
com.fasterxml.jackson.dataformat.ion.ionvalue		0%		0%	31	31	64	64	19	19	7	7
com.fasterxml.jackson.dataformat.ion.jsr310		0%		0%	51	51	124	124	29	29	4	4
com.fasterxml.jackson.dataformat.ion.polymorphism		0%		0%	51	51	91	91	29	29	4	4
com.fasterxml.jackson.dataformat.ion.util		0%		0%	11	11	27	27	9	9	1	1
com.fasterxml.jackson.dataformat.protobuf		0%		0%	926	926	2,424	2,424	293	293	12	12
com.fasterxml.jackson.dataformat.protobuf.protoparser.protoparser		0%		0%	829	829	1,771	1,771	321	321	43	43
com.fasterxml.jackson.dataformat.protobuf.schema		0%		0%	397	397	845	845	202	202	51	51
com.fasterxml.jackson.dataformat.protobuf.schemagen		0%		0%	102	102	202	202	65	65	10	10
com.fasterxml.jackson.dataformat.smile		32%		30%	879	1,238	2,335	3,395	238	352	5	13
com.fasterxml.jackson.dataformat.smile.async		0%		0%	406	406	1,214	1,214	88	88	2	2
com.fasterxml.jackson.dataformat.smile.databind		5%		0%	19	21	31	35	13	15	1	2
default		3%		0%	71	73	239	249	18	20	11	13
Total	67,914 of 78,477	13%	7,334 of 8,500	13%	6,672	7,467	15,525	17,795	2,774	3,019	301	323

Figure 11: Fuzzer Coverage for Jackson Dataformats Binary as of 1st December 2023

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
com.fasterxml.jackson.dataformat.avro		16%		4%	314	372	587	703	198	253	7	19
com.fasterxml.jackson.dataformat.avro.apacheimpl		1%		0%	153	156	275	279	113	116	9	11
com.fasterxml.jackson.dataformat.avro.deser		3%		3%	737	761	1,601	1,672	344	362	55	60
com.fasterxml.jackson.dataformat.avro.jsr310		0%		n/a	1	1	14	14	1	1	1	1
com.fasterxml.jackson.dataformat.avro.jsr310.deser		0%		0%	17	17	27	27	16	16	5	5
com.fasterxml.jackson.dataformat.avro.jsr310.ser		0%		0%	20	20	43	43	16	16	4	4
com.fasterxml.jackson.dataformat.avro.schema		0%		0%	235	235	482	482	104	104	14	14
com.fasterxml.jackson.dataformat.avro.ser		0%		0%	205	205	443	443	98	98	11	11
com.fasterxml.jackson.dataformat.cbor		46%		45%	687	1,195	1,575	2,981	208	369	2	15
com.fasterxml.jackson.dataformat.cbor.databind		26%		0%	10	15	16	25	7	12	0	2
com.fasterxml.jackson.dataformat.ion		43%		45%	227	369	378	713	141	240	5	19
com.fasterxml.jackson.dataformat.ion.ionvalue		0%		0%	31	31	64	64	19	19	7	7
com.fasterxml.jackson.dataformat.ion.jsr310		0%		0%	51	51	124	124	29	29	4	4
com.fasterxml.jackson.dataformat.ion.polymorphism		0%		0%	51	51	91	91	29	29	4	4
com.fasterxml.jackson.dataformat.ion.util		0%		0%	11	11	27	27	9	9	1	1
com.fasterxml.jackson.dataformat.protobuf		2%		1%	901	930	2,339	2,428	271	296	5	12
com.fasterxml.jackson.dataformat.protobuf.protoparser.protoparser		0%		0%	829	829	1,771	1,771	321	321	43	43
com.fasterxml.jackson.dataformat.protobuf.schema		0%		0%	395	397	842	845	200	202	50	51
com.fasterxml.jackson.dataformat.protobuf.schemagen		0%		0%	102	102	202	202	65	65	10	10
com.fasterxml.jackson.dataformat.smile		47%		52%	628	1,249	1,745	3,412	169	356	2	13
com.fasterxml.jackson.dataformat.smile.async		0%		0%	407	407	1,217	1,217	88	88	2	2
com.fasterxml.jackson.dataformat.smile.databind		17%		0%	16	21	26	35	10	15	0	2
default		51%		41%	179	290	420	828	24	46	9	29
Total	62,852 of 80,407	21%	6,848 of 8,755	21%	6,207	7,715	14,309	18,426	2,480	3,062	250	339

Figure 12: Fuzzer Coverage for Jackson Dataformats Binary as at 9th January 2024

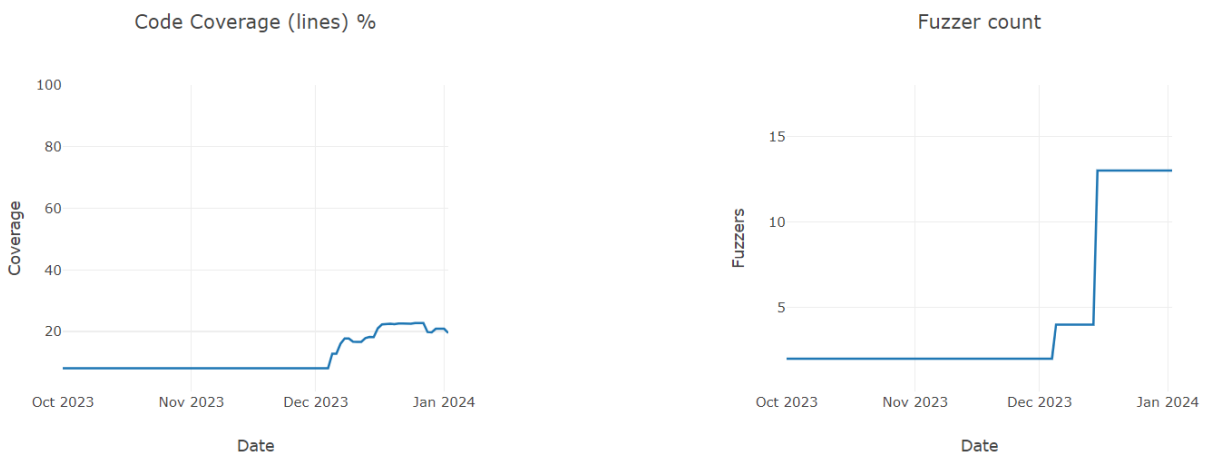


Figure 13: Fuzz-Introspector report for Jackson Dataformats Binary

wrappers for the existing Jackson Databinding module except for those steps transforming to and from support binary formats. The serialisation and deserialisation for those underlying generic objects are done by the base Jackson Databind modules and thus this library contains many classes with low cyclomatic complexity, many of them contain many one-liner wrappers for invoking different superclasses methods in the base Jackson Databind module. These methods and classes are therefore not fuzzworthy.

As a whole, there is an estimated 20% of methods have very low cyclomatic complexity (5 or less) which is therefore not worth to fuzz.

Last but not least, the upstream libraries of those binary formats enforce strict input checkers and thus the fuzzers need more time to explore different branches because many of the random inputs are denied those input checkers with exceptions thrown. Also, some of the newest coverage is not reflected in the coverage report and the coverage is assumed to be increasing in the coming months.

Upstream fixes

https://github.com/FasterXML/jackson-dataformats-binary/pull/418

https://github.com/FasterXML/jackson-dataformats-binary/pull/421

https://github.com/FasterXML/jackson-dataformats-binary/pull/425

https://github.com/FasterXML/jackson-dataformats-binary/pull/427

https://github.com/FasterXML/jackson-dataformats-binary/pull/435

Issues found by fuzzers

#	ID	Title	Severity	Fixed
1	ADA-JACKSON-BINARY-2023-1	Unexpected IndexOutOf-BoundsException in JacksonAvroParserImpl	Low	Yes
3	ADA-JACKSON-BINARY-2023-3	Unexpected IndexOutOf-BoundsException in CBORParser	Low	Yes
4	ADA-JACKSON-BINARY-2023-4	Unexpected IndexOutOf-BoundsException in IonParser	Low	Yes
5	ADA-JACKSON-BINARY-2023-5	Unexpected IndexOutOf-BoundsException in IonReader implementations	Low	Yes
6	ADA-JACKSON-BINARY-2023-6	Unexpected NullPointerException in IonParser	Low	Yes
7	ADA-JACKSON-BINARY-2023-7	Unexpected NullPointerException in Ion-Parser::getNumberType()	Low	Yes
8	ADA-JACKSON-BINARY-2023-8	Unexpected AssertionError in IonParser	Low	Yes
9	ADA-JACKSON-BINARY-2023-9	Unexpected IndexOutOf-BoundsException in SmileParser	Low	Yes
10	ADA-JACKSON-BINARY-2023-10	Stack out of memory in Jackson standard ThrowableDeserializer	Moderate	No

Remark for Jacoco coverage report

The Jacoco fuzzer coverage report shows the instructions and branches covered/missed of each existing package in the project by the fuzzers. It means that after fuzzing for some time until the report

generation, the number of instructions and branches of the project has been reached by the fuzzers. Sometimes some instructions and branches are not covered simply because they are not reachable directly by fuzzers. This could happen if some methods or classes have protected or private modifiers, or they are some unused code located in abstract classes or interfaces. It could also be that the fuzzers explicitly skipped some methods which is not fuzzworthy or it requires some special input to reach some of the branches which are not yet used for fuzzing. In conclusion, the Jacoco coverage report provides an objective understanding of the code that has been covered by fuzzers.

Issues found

Here we present the issues that we identified during the audit.

#	ID	Title	Severity	Fixed
1	ADA-JACKSON-BINARY-2023-1	Unexpected IndexOutOf-BoundsException in JacksonAvroParserImpl	Low	Yes
2	ADA-JACKSON-BINARY-2023-2	Vulnerable version of the Avro dependency is used	Moderate	No
3	ADA-JACKSON-BINARY-2023-3	Unexpected IndexOutOf-BoundsException in CBORParser	Low	Yes
4	ADA-JACKSON-BINARY-2023-4	Unexpected IndexOutOf-BoundsException in IonParser	Low	Yes
5	ADA-JACKSON-BINARY-2023-5	Unexpected IndexOutOf-BoundsException in IonReader implementations	Low	Yes
6	ADA-JACKSON-BINARY-2023-6	Unexpected NullPointerException in IonParser	Low	Yes
7	ADA-JACKSON-BINARY-2023-7	Unexpected NullPointerException in Ion-Parser::getNumberType()	Low	Yes
8	ADA-JACKSON-BINARY-2023-8	Unexpected AssertionError in IonParser	Low	Yes
9	ADA-JACKSON-BINARY-2023-9	Unexpected IndexOutOf-BoundsException in SmileParser	Low	Yes
10	ADA-JACKSON-BINARY-2023-10	Stack out of memory in Jackson standard ThrowableDeserializer	Moderate	No

#	ID	Title	Severity	Fixed
11	ADA-JACKSON-COLLECTIONS-2023-1	Unexpected NullPointerException when deserializing	Low	Yes
12	ADA-JACKSON-COLLECTIONS-2023-2	Infinite recursive loop in GuavaOptionalDeserializer	Moderate	No
13	ADA-JACKSON-COLLECTIONS-2023-3	Vulnerable version of the Guava dependency is used	Informational	No
14	ADA-JACKSON-JODA-2023-1	Direct comparison of Boolean object in JacksonJodaDateFormat	Low	No
15	ADA-JACKSON-JODA-2023-2	Unnecessary auto-boxing/unboxing in IntervalDeserializer	Informational	No
16	ADA-JACKSON-TEXT-2023-1	Unused conditional check in CsvDecoder	Informational	No
17	ADA-JACKSON-TEXT-2023-2	Unexpected NullPointerException in YAMLParser	Low	Yes
18	ADA-JACKSON-XML-2023-1	Unexpected ArrayIndexOutOfBoundsException in XMLTokenStream with SJSXP	Low	Yes
19	ADA-JACKSON-XML-2023-2	XML External Entity vulnerability in XMLFactory	Moderate	No

[Dataformats-Binary-Ion] Unexpected IndexOutOfBoundsException in JacksonAvroParserImpl

Severity	Low
Status	Fixed
id	ADA-JACKSON-BINARY-2023-1
Component	JacksonAvroParserImpl

In the `JacksonAvroParserImpl::_finishShortText(int)` method and the `JacksonAvroParserImpl::_finishLongText(int)` method, there are missing bound checks during value reading from the byte array `inputBuf` and could cause unexpected `IndexOutOfBoundsException` if the provided input is not correctly ended.

In the first line of the **`do..while`** loop for the `JacksonAvroParserImpl::_finishShortText(int)` method, the current index pointed by the `inPtr` variable is retrieved, processed and stored. Then the `inPtr` value is increased by one. From the study of the code, the value of `inPtr` must be within the range of the byte array `inputBuf`, but since it has increased by one in this line, the value of `inPtr` may be larger or equals to the length of `inputBuf` on some invalid input. This causes the subsequent `inputBuf` value access with the out-of-bound `inPtr` to throw an unexpected `IndexOutOfBoundsException`. Similar situation is found in the `JacksonAvroParserImpl::_finishLongText(int)` method

Source direct link:

https://github.com/FasterXML/jackson-dataformats-binary/blob/041d61919d1afa8db4b474d73ece0450707a3e25/avro/src/main/java/com/fasterxml/jackson/dataformat/avro/deser/JacksonAvroParserImpl.java#L628-L657

```
628         final int[] codes = sUtf8UnitLengths;
629         do {
630             i = inputBuf[inPtr++] & 0xFF;
631             switch (codes[i]) {
632             case 0:
633                 break;
634             case 1:
635                 i = ((i & 0x1F) << 6) | (inputBuf[inPtr++] & 0x3F);
636                 break;
637             case 2:
638                 i = ((i & 0x0F) << 12)
639                     | ((inputBuf[inPtr++] & 0x3F) << 6)
```

```
640                     | (inputBuf[inPtr++] & 0x3F);
641                 break;
642             case 3:
643                 i = ((i & 0x07) << 18)
644                  | ((inputBuf[inPtr++] & 0x3F) << 12)
645                  | ((inputBuf[inPtr++] & 0x3F) << 6)
646                  | (inputBuf[inPtr++] & 0x3F);
647                 // note: this is the codepoint value; need to split,
                       too
648                 i -= 0x10000;
649                 outBuf[outPtr++] = (char) (0xD800 | (i >> 10));
650                 i = 0xDC00 | (i & 0x3FF);
651                 break;
652             default: // invalid
653                 _reportError("Invalid byte "+Integer.toHexString(i)+"
                       in Unicode text block");
654             }
655             outBuf[outPtr++] = (char) i;
656         } while (inPtr < end);
657         return _textBuffer.setCurrentAndReturn(outPtr);
```

https://github.com/FasterXML/jackson-dataformats-binary/blob/041d61919d1afa8db4b474d73ece0450707a3e25/avro/src/main/java/com/fasterxml/jackson/dataformat/avro/deser/JacksonAvroParserImpl.java#L693-L696

```
693             case 3: // 4-byte UTF
694                 c = _decodeUTF8_4(c);
695                 // Let's add first part right away:
696                 outBuf[outPtr++] = (char) (0xD800 | (c >> 10));
```

Mitigation

The suggested fix is to add a bound checking in the `do..while` loop after the `inPtr++` process to ensure the `inPtr` is still within bound and throws an error if not for identifying invalid input.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65618

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65649

Upstream fix

https://github.com/FasterXML/jackson-dataformats-binary/pull/450

https://github.com/FasterXML/jackson-dataformats-binary/pull/453

[Dataformats-Binary-Avro] Vulnerable version of the Avro dependency is used

Severity	Moderate
Status	Reported
id	ADA-JACKSON-BINARY-2023-2
Component	avro/pom.xml

Avro version `1.11.2` or before are found to be vulnerable to out-of-bound memory read and cause Out of Memory. That has been reported in `CVE-2023-39410` (https://www.cve.org/CVERecord?id=CVE-2023-39410) and fixed in Avro version 1.11.3. It is found that the `pom.xml` in the Avro module of the Jackson-data formats-binary library is still using Avro version `1.8.2` which makes the module vulnerable to the possible Out-of-Memory crashing problem documented in `CVE-2023-39410`. As the modules in the Jackson-data formats-binary library are meant to be used as module objects for serialising and deserialising from and to Jackson-supported data types to the Avro format, thus the code is vulnerable if malicious data are being passed to the library for serialisation and deserialisation purposes.

Source direct link:

https://github.com/FasterXML/jackson-dataformats-binary/blob/896dd7f8193bc71a84208022a10203cf31fe9bb0/avro/pom.xml#L47-L51

```
47      <dependency>
48        <groupId>org.apache.avro</groupId>
49        <artifactId>avro</artifactId>
50        <version>1.8.2</version>
51      </dependency>
```

Mitigation

Avro maintainers have fixed this specific CVE in version 1.11.3 which has already been published. Thus the suggested fix is to update the Avro version from the used `1.8.2` to `1.11.3` to avoid the problem.

[Dataformats-Binary-Ion] Unexpected IndexOutOfBoundsException in CBORParser

Severity	Low
Status	Fixed
id	ADA-JACKSON-BINARY-2023-3
Component	CBORParser

The `CBORParser::nextToken()` method relies on the integer index `_inputPtr` to read the next character from the provided input byte array. In some cases, if the provided input byte array is malformed and contains negative bytes, that negative could be used as the new value for the `_inputPtr`. If the negative `_inputPtr` is used as an index for later access to the byte array, an unexpected `IndexOutOfBoundsException` is thrown because a negative index is used.

Source direct link:

https://github.com/FasterXML/jackson-dataformats-binary/blob/041d61919d1afa8db4b474d73ece0450707a3e25/cbo
L816

```
811         if (_inputPtr >= _inputEnd) {
812             if (!loadMore()) {
813                 return _eofAsNextToken();
814             }
815         }
816         int ch = _inputBuffer[_inputPtr++] & 0xFF;
```

Mitigation

The suggested fix is to add a negative checking before the use of `_inputPtr`. It is shown that there is already a check in the method to ensure `_inputPtr` is not larger than or equal to the `_inputEnd`, but there is no check to confirm that `_inputPtr` is not negative. The suggested fix is to add a negative check to ensure the retrieved `_inputPtr` is not negative before use.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65617

Upstream fix

https://github.com/FasterXML/jackson-dataformats-binary/pull/452

[Dataformats-Binary-Ion] Unexpected IndexOutOfBoundsException in IonParser

Severity	Low
Status	Fixed
id	ADA-JACKSON-BINARY-2023-4
Component	IonParser

In the `IonParser` class, there are multiple methods to retrieve the `BigInteger` or `BigDecimal` type of objects. The upstream Ion-Java library does not ensure the retrieval of those types of objects must be successful if invalid data is provided. The upstream library always assumes that the provided byte buffer has enough bytes remaining for reading a `BigInteger` or `BigDecial` (or related object like a timestamp). Thus if the remaining bytes are not enough, the upstream Ion-Java library could throw an unexpected `IndexOutOfBoundsException`.

The following code could get an unexpected `IndexOutOfBoundsException` if the remaining buffer is not long enough.

Source code location:

https://github.com/FasterXML/jackson-dataformats-binary/blob/84371784f0b45e56fc0fbea2e6b069221d512012/ion/src/main/java/com/fasterxml/jackson/dataformat/ion/IonParser.java#L337

```
337                 return _reader.bigIntegerValue();
```

https://github.com/FasterXML/jackson-dataformats-binary/blob/84371784f0b45e56fc0fbea2e6b069221d512012/ion/src/main/java/com/fasterxml/jackson/dataformat/ion/IonParser.java#L348

```
348                 return _reader.bigDecimalValue();
```

https://github.com/FasterXML/jackson-dataformats-binary/blob/84371784f0b45e56fc0fbea2e6b069221d512012/ion/s

```
298                 Timestamp ts = _reader.timestampValue();
```

Mitigation

The suggested fix is to wrap the `IndexOutOfBoundsException` with `JsonParseException` mentioning that the input is invalid.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65513

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65628

Upstream fix

https://github.com/FasterXML/jackson-dataformats-binary/pull/440

[Dataformats-Binary-Ion] Unexpected IndexOutOfBoundsException in IonReader implementations

Severity	Low
Status	Fixed
id	ADA-JACKSON-BINARY-2023-5
Component	IonParser

Attackers can crash the application that adopts the Jackson-dataformats-binary library which does not handle the unexpected `IndexOutOfBoundsException`. It will create Denial-of-Service if the vulnerable application is meant to be running as a web service, this cause legitimate users of the vulnerable applications becomes a victim of Denial-of-Service.

The `IonParser::nextToken()` method relies on the IonReader implementations to retrieve the next token. Those IonReader implementations are provided by the upstream Amazon Java-Ion package and some of the code in those IonReader implementations does mention that if the provided data is malformed, it could throw `IndexOutOfBoundsException` and that is not handled because it would sacrifice performance. And `IonParser::nextToken()` fails to handle them nor check if the input is malformed. This results in an unexpected `IndexOutOfBoundsException` being thrown to the user.

Source direct link:

https://github.com/FasterXML/jackson-dataformats-binary/blob/896dd7f8193bc71a84208022a10203cf31fe9bb0/ion/src/main/java/com/fasterxml/jackson/dataformat/ion/IonParser.java#L531-L583

```
531     public JsonToken nextToken() throws IOException
532     {
533         // special case: if we return field name, we know value type,
                return it:
534         if (_currToken == JsonToken.FIELD_NAME) {
535             return (_currToken = _valueToken);
536         }
537         // also, when starting array/object, need to create new context
538         if (_currToken == JsonToken.START_OBJECT) {
539             _parsingContext = _parsingContext.createChildObjectContext
                    (-1, -1);
540             _reader.stepIn();
541         } else if (_currToken == JsonToken.START_ARRAY) {
```

```
542             _parsingContext = _parsingContext.createChildArrayContext
                    (-1, -1);
543             _reader.stepIn();
544         }
545
546         // any more tokens in this scope?
547         IonType type = null;
548         try {
549             type = _reader.next();
550         } catch (IonException e) {
551             _wrapError(e.getMessage(), e);
552         }
553         if (type == null) {
554             if (_parsingContext.inRoot()) { // EOF?
555                 close();
556                 _currToken = null;
557             } else {
558                 _parsingContext = _parsingContext.getParent();
559                 _currToken = _reader.isInStruct() ? JsonToken.
                        END_OBJECT : JsonToken.END_ARRAY;
560                 _reader.stepOut();
561             }
562             return _currToken;
563         }
564         // Structs have field names; need to keep track:
565         boolean inStruct = !_parsingContext.inRoot() && _reader.
                isInStruct();
566         // (isInStruct can return true for the first value read if the
                reader
567         // was created from an IonValue that has a parent container)
568         try {
569             // getFieldName() can throw an UnknownSymbolException if
                    the text of the
570             // field name symbol cannot be resolved.
571             _parsingContext.setCurrentName(inStruct ? _reader.
                    getFieldName() : null);
572         } catch (UnknownSymbolException e) {
573             _wrapError(e.getMessage(), e);
574         }
575         JsonToken t = _tokenFromType(type);
576         // and return either field name first
577         if (inStruct) {
578             _valueToken = t;
579             return (_currToken = JsonToken.FIELD_NAME);
580         }
581         // or just the value (for lists, root value)
582         return (_currToken = t);
583     }
```

Below are two sample code that mentioned the possible throwing of `IndexOutOfBoundsException` from the upstream `IonCursorBinary::uncheckedReadVarUInt_1_0(byte)` method and

`IonReaderContinuableCoreBinary::readVarInt_1_0` method.

Source direct link:

https://github.com/amazon-ion/ion-java/blob/b0d3dcc141774a60705adc2b0bda68026987b17f/src/main/java/com/amazon/ion/impl/IonReaderContinuableCoreBinary.java#L195-L210

```
195     private int readVarInt_1_0(int firstByte) {
196         int currentByte = firstByte;
197         int sign = (currentByte & VAR_INT_SIGN_BITMASK) == 0 ? 1 : -1;
198         int result = currentByte & LOWER_SIX_BITS_BITMASK;
199         while ((currentByte & HIGHEST_BIT_BITMASK) == 0) {
200             // Note: if the varInt is malformed such that it extends
                    beyond the declared length of the value *and*
201             // beyond the end of the buffer, this will result in
                    IndexOutOfBoundsException because only the declared
202             // value length has been filled. Preventing this is simple:
                     if (peekIndex >= valueMarker.endIndex) throw
203             // new IonException(); However, we choose not to perform
                    that check here because it is not worth sacrificing
204             // performance in this inner-loop code in order to throw
                    one type of exception over another in case of
205             // malformed data.
206             currentByte = buffer[(int)(peekIndex++)];
207             result = (result << VALUE_BITS_PER_VARUINT_BYTE) | (
                    currentByte & LOWER_SEVEN_BITS_BITMASK);
208         }
209         return result * sign;
210     }
```

Source direct link:

https://github.com/amazon-ion/ion-java/blob/b0d3dcc141774a60705adc2b0bda68026987b17f/src/main/java/com/amazon/ion/impl/IonCursorBinary.java#L727-L740

```
727     private long uncheckedReadVarUInt_1_0(byte currentByte) {
728         long result = currentByte & LOWER_SEVEN_BITS_BITMASK;
729         do {
730             // Note: if the varUInt  is malformed such that it extends
                    beyond the declared length of the value *and*
731             // beyond the end of the buffer, this will result in
                    IndexOutOfBoundsException because only the declared
732             // value length has been filled. Preventing this is simple:
                     if (peekIndex >= limit) throw
733             // new IonException(); However, we choose not to perform
                    that check here because it is not worth sacrificing
734             // performance in this inner-loop code in order to throw
                    one type of exception over another in case of
735             // malformed data.
736             currentByte = buffer[(int) (peekIndex++)];
737             result = (result << VALUE_BITS_PER_VARUINT_BYTE) | (
```

```
                    currentByte & LOWER_SEVEN_BITS_BITMASK);
738         } while (currentByte >= 0);
739         return result;
740     }
```

Mitigation

The simplest fix is to catch the `IndexOutOfBoundsException` and wrap it with the `JsonParseException`. A better way may be adding some checking before the upstream call to ensure malformed data is detected and exit before calling those upstream methods.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65062

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65083

Upstream fix

https://github.com/FasterXML/jackson-dataformats-binary/pull/421

Code behaviour after the fix

The unexpected `IndexOutOfBoundsException` is wrapped and an expected `StreamReadException` is thrown instead.

[Dataformats-Binary-Ion] Unexpected NullPointerException in IonParser

Severity	Low
Status	Fixed
id	ADA-JACKSON-BINARY-2023-6
Component	IonParser

In the `IonParser::getText()` method, there is a call to the `IonReader::stringValue()`. Also, in `IonParser::getXXXValue()` for retrieving different number values from the `IonReader` calls to underlying `IonReader` for retrieving string or number value. According to the Javadoc of `IonReader`, each of the APIs requires a special `IonType` and `IllegalStateException` could be thrown if the wrong type is passed. But there is a special case when there is no more input, the `IonType` will be null and continuing calling those methods will result in `NullPointerException`. In some cases of `IonParser::getXXXValue()`, if there is no buffer configuration assigned from malformed input, the call to `getBufferConfiguration()` which is required in retrieving some number type would return **`null`** and make the subsequent call throw an unexpected `NullPointerException`.

Source direct link:

https://github.com/FasterXML/jackson-dataformats-binary/blob/db12a6571842887d5a4c83f1a0b45b5f3514ba43/ion/src/main/java/com/fasterxml/jackson/dataformat/ion/IonParser.java#L273-L307

```
273             case VALUE_STRING:
274                 try {
275                     return _reader.stringValue();
276                 } catch (UnknownSymbolException e) {
277                     // stringValue() will throw an
                        UnknownSymbolException if we're
278                     // trying to get the text for a symbol id that
                        cannot be resolved.
279                     // stringValue() has an assert statement which
                        could throw an
280                     throw _constructError(e.getMessage(), e);
281                 } catch (AssertionError e) {
282                     // AssertionError if we're trying to get the text
                        with a symbol
283                     // id less than or equals to 0.
284                     String msg = e.getMessage();
```

```
285                     if (msg == null) {
286                         msg = "UNKNOWN ROOT CAUSE";
287                     }
288                     throw _constructError("Internal `IonReader` error:
                            "+msg, e);
289                 }
```

Source direct link:

https://github.com/FasterXML/jackson-dataformats-binary/blob/db12a6571842887d5a4c83f1a0b45b5f3514ba43/ion/src/main/java/com/fasterxml/jackson/dataformat/ion/IonParser.java#L332-L360

```
332     @Override
333     public BigInteger getBigIntegerValue() throws IOException {
334         return _reader.bigIntegerValue();
335     }
336
337     @Override
338     public BigDecimal getDecimalValue() throws IOException {
339         return _reader.bigDecimalValue();
340     }
341
342     @Override
343     public double getDoubleValue() throws IOException {
344         return _reader.doubleValue();
345     }
346
347     @Override
348     public float getFloatValue() throws IOException {
349         return (float) _reader.doubleValue();
350     }
351
352     @Override
353     public int getIntValue() throws IOException {
354         return _reader.intValue();
355     }
356
357     @Override
358     public long getLongValue() throws IOException {
359         return _reader.longValue();
360     }
```

It is found that in the `IonParser::getNumberValue()` method, there is a null check to ensure the `IonType` (and `NumberType`) of the current token is not null before calling the corresponding data retrieving method in the `IonReader` implementation. But these null checks are missing from the above method which could cause unexpected `NullPointerException`.

https://github.com/FasterXML/jackson-dataformats-binary/blob/db12a6571842887d5a4c83f1a0

b45b5f3514ba43/ion/src/main/java/com/fasterxml/jackson/dataformat/ion/IonParser.java#L391-L411

```
391     @Override
392     public Number getNumberValue() throws IOException {
393         NumberType nt = getNumberType();
394         if (nt != null) {
395             switch (nt) {
396             case INT:
397                 return _reader.intValue();
398             case LONG:
399                 return _reader.longValue();
400             case FLOAT:
401                 return (float) _reader.doubleValue();
402             case DOUBLE:
403                 return _reader.doubleValue();
404             case BIG_DECIMAL:
405                 return _reader.bigDecimalValue();
406             case BIG_INTEGER:
407                 return getBigIntegerValue();
408             }
409         }
410         return null;
411     }
```

Mitigation

The simplest fix is to add a null check similar to the one done in the `IonParser::getNumberValue()` method and wrap some of the `NullPointerException` with the `JsonStreamException` to avoid unexpected `NullPointerException` thrown directly to the users.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65065

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65106

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65274

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65452

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65479

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65557

Upstream fix

https://github.com/FasterXML/jackson-dataformats-binary/pull/425

https://github.com/FasterXML/jackson-dataformats-binary/commit/0e2a81a78dbfa6583bee7520c2d441dbb38e2f5b

[Dataformats-Binary-Ion] Unexpected NullPointerException in IonParser::getNumberType()

Severity	Low
Status	Fixed
id	ADA-JACKSON-BINARY-2023-7
Component	IonParser

In the `IonParser::getNumberType()method`, there is an invocation of the `IonReader.getIntegerSize()` method which could return a **`null`** value in some cases with invalid data. If the result is null, the code will throw a NullPointerException in the next line when the value is used for the switch condition.

Also, the `IonReader.getIntegerSize()` method will throw `NullPointerException` in some cases, thus it is also necessary to wrap around the method invocation to ensure `NullPointerException` is caught.

Source code location:

https://github.com/FasterXML/jackson-dataformats-binary/blob/84371784f0b45e56fc0fbea2e6b069221d512012/ion/src/main/java/com/fasterxml/jackson/dataformat/ion/IonParser.java#L389-L415

```
389     public NumberType getNumberType() throws IOException
390     {
391         IonType type = _reader.getType();
392         if (type != null) {
393             // Hmmh. Looks like Ion gives little bit looser definition here;
394             // harder to pin down exact type. But let's try some checks still.
395             switch (type) {
396             case DECIMAL:
397                 //Ion decimals can be arbitrary precision, need to read as big decimal
398                 return NumberType.BIG_DECIMAL;
399             case INT:
400                 IntegerSize size = _reader.getIntegerSize();
401                 switch (size) {
```

Mitigation

The suggested fix is to add a null checking after the invocation of the `IonReader.getIntegerSize()` method and throw an exception if the return value stored in `size` is indeed null. Also, temporary wrap the `IonReader.getIntegerSize()` method invocation with a try-catch block to catch the possible `NullPointerException` until that has been fixed from the upstream Amazon-Ion-Java library in https://github.com/amazon-ion/ion-java/issues/685.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65268

Upstream fix

https://github.com/FasterXML/jackson-dataformats-binary/pull/435

https://github.com/amazon-ion/ion-java/issues/685

[Dataformats-Binary-Ion] Unexpected AssertionError in IonParser

Severity	Low
Status	Fixed
id	ADA-JACKSON-BINARY-2023-8
Component	IonParser

Attackers can crash the application that consumes the Jackson-dataformats-binary library which does not handle the unexpected `AssertionError` thrown from `IonReader`. It will create Denial-of-Service if the vulnerable application is meant to be running as a web service, this cause legitimate users of the vulnerable applications becomes a victim of Denial-of-Service.

In the `IonParser::getText()` method, there is a call to the `IonReader::stringValue()` which is served by an Amazon implementation of `IonReaderTextSystemX`. The method does throw `UnknownSymbolException` if the symbol id cannot be resolved. But it also contains some assert statements which throw `AssertionError` when the resolved symbol id is 0 or negative. The AssertionError is not handled and is thrown to the users unexpectedly.

Source direct link:

https://github.com/FasterXML/jackson-dataformats-binary/blob/0e76830aceed2b2f208743614d34ad37994d7682/ion/src/main/java/com/fasterxml/jackson/dataformat/ion/IonParser.java#L273-L298

```
273             case VALUE_STRING:
274                 try {
275                     // stringValue() will throw an
                           UnknownSymbolException if we're
276                     // trying to get the text for a symbol id that
                           cannot be resolved.
277                     return _reader.stringValue();
278                 } catch (UnknownSymbolException e) {
279                     throw _constructError(e.getMessage(), e);
280                 }
```

Mitigation

The simplest fix is to also catch the `AssertionError`, the same as the `UnkonwnSymbolException`. In general, AssertionError should be internal use only and should be wrapped and avoided by throwing directly to the users. Not sure if it is meant to not handle it in this situation.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64721

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64917

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65273

Upstream fix

https://github.com/FasterXML/jackson-dataformats-binary/pull/418

https://github.com/FasterXML/jackson-dataformats-binary/pull/433

[Dataformats-Binary-Smile] Unexpected IndexOutOfBoundsException in SmileParser

Severity	Low
Status	Fixed
id	ADA-JACKSON-BINARY-2023-9
Component	SmileParser

In the `SmileParser::nextTextValue()` method, there is a line that uses the Integer `ptr` as an index to retrieve a byte from the `_inputBuffer`. But it is found that with some invalid input and repeat calls to the `SmileParser::nextTextValue()` method, it could cause ptr to be negative and trigger an unexpected `ArrayIndexOutOfBoundsException`.

Source direct link:

https://github.com/FasterXML/jackson-dataformats-binary/blob/db12a6571842887d5a4c83f1a0b45b5f3514ba43/smile/src/main/java/com/fasterxml/jackson/dataformat/smile/SmileParser.java#L908-L1014

```
908             int ptr = _inputPtr;
909             if (ptr >= _inputEnd) {
910                 if (!_loadMore()) {
911                     _eofAsNextToken();
912                     return null;
913                 }
914                 ptr = _inputPtr;
915             }
916             _tokenOffsetForTotal = ptr;
917 //          _tokenInputTotal = _currInputProcessed + _inputPtr;
918             int ch = _inputBuffer[ptr++] & 0xFF;
```

Mitigation

Add a bound check for the ptr before using it as the array index.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65126

Upstream fix

https://github.com/FasterXML/jackson-dataformats-binary/pull/427

[Dataformats-Binary / Dataformats-Text] Stack out of memory in Jackson standard ThrowableDeserializer

Severity	Moderate
Status	Report
id	ADA-JACKSON-BINARY-2023-10
Component	ThrowableDeserializer

Attackers can crash the application that adopts the Jackson-dataformats-text / Jackson-dataformats-binary library which does not handle those large data inputs. Attackers can also exhaust the memory of the JVM that is running the vulnerable application. These situations will create Denial-of-Service if the vulnerable application is meant to be running as a web service, this cause legitimate users of the vulnerable applications becomes a victim of Denial-of-Service.

This is a possible stack out-of-memory problem in IonParser when parsing an Exception type object with a high depth level. `IonParser` depends on the `ThrowableDeserializer::deserializeFromObject()` method to deserialize `Throwable` object before transforming to Ion format. But if the provided deserialized data contains a high depth level, the recursive logic could cause a Stack out-of-memory problem.

Source direct link:

https://github.com/FasterXML/jackson-databind/blob/15fa6ec14608790664f214ab53688b68aad23dbd/src/main/java/com/fasterxml/jackson/databind/deser/std/ThrowableDeserializer.java#L164-L165

```
164                     suppressed = ctxt.readValue(p,
165                             ctxt.constructType(Throwable[].class));
```

Proof of concept using 'IonParser' from Jackson-dataformats-binary

```
1  import com.fasterxml.jackson.dataformat.ion.*;
2
3  public class ProofOfConcept {
4    public static void main(String[] args) throws Exception {
5      IonFactory f = IonFactory.builderForTextualWriters().enable(
           IonParser.Feature.USE_NATIVE_TYPE_ID).build();
6      IonObjectMapper mapper = IonObjectMapper.builder(f).build();
7      String open = "{suppressed:rr::(";
8      String close = ")}";
```

```
9       mapper.readValue(open.repeat(10000) + close.repeat(10000),
            Exception.class);
10    }
11  }
```

Proof of concept using ‘YamlParser‘ from jackson-dataformats-text

```
1   import com.fasterxml.jackson.dataformat.yaml.*;
2
3   public class ProofOfConcept {
4     public static void main(String[] args) throws Exception {
5       YAMLMapper mapper = YAMLMapper.builder(YAMLFactory.builder().build
            ()).build();
6       String open = "suppressed:\n [ \\U000,   \n";
7       String close = "";
8       mapper.readValue(open.repeat(10000) + close.repeat(10000),
            Exception.class);
9     }
10  }
```

Mitigation

Since `Throwable` objects generally don’t have that much depth. Thus adding limitations to the deserialization of the `Throwable` object (i.e. 256) could avoid much reduced Stack out-of-memory error. But it can also be argued that this issue is out of scope because it happened in the ThrowableDeserializer which is not within the five projects. It is more like a general issue for the Jackson Library in general when deserializing an invalid or valid but contains high-depth-level serialized data. As the general ThrowableDeserializer is used by different JsonParser implementations of different Jackson-supported formats, thus to trigger it through different serialized data formats could be different as shown above.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65000

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65084

[Datatypes-Collections] Unexpected NullPointerException when deserializing

Severity	Low
Status	Fixed
id	ADA-JACKSON-COLLECTIONS-2023-1
Component	EclipseCollection.PrimitiveKVHandler / Guava-CollectionDeserializer

Attackers can crash the application that adopts the Jackson-datatypes-collections library which does not handle the unexpected `NullPointerException`. It will create Denial-of-Service if the vulnerable application is meant to be running as a web service, this cause legitimate users of the vulnerable applications becomes a victim of Denial-of-Service.

Some methods in the project fail to handle invalid input and throw unexpected `NullPointerExcetption`. For example, the `PrimitiveKVHandler.Char::value()` method retrieves a string return from `parser.getValueAsString()`. If the input provided in the parser is invalid and cannot be converted to a string, it will return **`null`**. But the next conditional check calls the length method directly without a null check which could cause an unexpected `NullPointerException` thrown.

Source direct link:

https://github.com/FasterXML/jackson-datatypes-collections/blob/56ce944dcd0f97371a3a3aa9d53461a73a80fbec/eclipse-collections/src/main/java/com/fasterxml/jackson/datatype/eclipsecollections/deser/map/PrimitiveKVHandler.java#L71-L80

```
71          public char value(DeserializationContext ctx, JsonParser parser
                ) throws IOException {
72              String valueAsString = parser.getValueAsString();
73              if (valueAsString.length() != 1) {
74                  ctx.reportInputMismatch(char.class,
75                                          "Cannot convert a JSON String
                                               of length %d into a char
                                               element of map",
76                                          valueAsString.length());
77              }
78              return valueAsString.charAt(0);
79          }
80      }
```

In `GuavaCollectionDeserializer::deserialize()` method, it deserialises the provided input and eventually creates a `GuavaImmutableCollection` object by the upstream `GuavaImmutableCollection` Builder. In the documentation of Guava, it does mention that in some cases (where the provided input is invalid), `NullPointerException` can be thrown but it is not specifically handled in the `GuavaCollectionDeserializer::deserialize()` method and causes unexpected `NullPointerException` thrown to the user.

Source direct link:

https://github.com/FasterXML/jackson-datatypes-collections/blob/56ce944dcd0f97371a3a3aa9d53461a73a80fbec/guava/src/main/java/com/fasterxml/jackson/datatype/guava/deser/GuavaCollectionDeserializer.java#L132-L144

```
132     public T deserialize(JsonParser p, DeserializationContext ctxt)
133             throws IOException
134     {
135         // Should usually point to START_ARRAY
136         if (p.isExpectedStartArrayToken()) {
137             return _deserializeContents(p, ctxt);
138         }
139         // But may support implicit arrays from single values?
140         if (ctxt.isEnabled(DeserializationFeature.
                ACCEPT_SINGLE_VALUE_AS_ARRAY)) {
141             return _deserializeFromSingleValue(p, ctxt);
142         }
143         return (T) ctxt.handleUnexpectedToken(_valueClass, p);
144     }
```

Mitigation

Add null checking and throw JsonProcessingException to indicate possible invalid data.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64610

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64629

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64936

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65117

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65142

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=65183

Upstream fix

https://github.com/FasterXML/jackson-datatypes-collections/pull/125

https://github.com/FasterXML/jackson-datatypes-collections/pull/139

Also inspire https://github.com/FasterXML/jackson-datatypes-collections/pull/141 (not fixed by us)

[Datatypes-Collections-Guava] Infinite recursive loop in GuavaOptionalDeserializer

Severity	Moderate
Status	Reported
id	ADA-JACKSON-COLLECTIONS-2023-2
Component	GuavaOptionalDeserializer

`GuavaOptionalDeserializer` is a subclass extending from the core Jackson `ReferenceTypeDeserializer` class. It inherits the 'getEmptyValue(DeserializationContext) method from the superclass and provides a self-implementation of the method. But the method simply calls itself with the provided parameters without doing anything, this creates an infinite recursive loop as there are no stopping criteria to stop the recursion call nor exiting the recursive loop. The method will continue the recursive call until either stack-overflow or out-of-memory and crash.

Source direct link:

https://github.com/FasterXML/jackson-datatypes-collections/blob/d6ec5337657eaee575969360a169e74bef555bcc/guava/src/main/java/com/fasterxml/jackson/datatype/guava/deser/GuavaOptionalDeserializer.java#L48-L51

```
48      @Override
49      public Object getEmptyValue(DeserializationContext ctxt) throws
            JsonMappingException {
50          return getEmptyValue(ctxt);
51      }
```

Mitigation

It is unknown the purpose of this method, it is assumed that it may want to pass the provided `DeserializationContext` context to the same method from the superclass to process. Thus it is assumed that a **`super`** keyword is missing from the code, causing the infinite recursive loop bug. If this subclass simply does not support this method call, throws an `UnsupportedOperationException` or simply returns and exits the method directly is a suggested fix.

[Datatypes-Collections-Guava] Vulnerable version of the Guava dependency is used

Severity	Informational
Status	Reported
id	ADA-JACKSON-COLLECTIONS-2023-3
Component	guava/pom.xml

Guava versions before `32.0.0` are found to be vulnerable to information leakage of temporary files created in the default Java temporary directory. That has been documented in `CVE-2023-2976` (https://www.cve.org/CVERecord?id=CVE-2023-2976). This CVE is known vulnerable if the `FileBackedOutputStream` in the Guava library has been used. As the Jackson-datatypes-collections package only provides add-on datatype support for Jackson that handles object serialisation and deserialisation to and from Guava base collection objects, thus direct use or support of `FileBackedOutputStream` is not found. Thus this issue reported remains informational to notify the existence of such CVE vulnerability and if future support of `FileBackedOutputStream` in Jackson is needed, it is recommended to update the Guava version to at least `32.0.1` to avoid being affected by those vulnerable versions of Guava library.

Source direct link:

https://github.com/FasterXML/jackson-datatypes-collections/blob/d6ec5337657eaee575969360a169e74bef555bcc/guava/pom.xml#L40

```
40      <version.guava>25.1-jre</version.guava>
```

Mitigation

No changes are needed if `FileBackedOutputStream` is confirmed as not used in the library. An update to version `32.0.1` or above is suggested for safety, and it is strongly recommended to update to version `32.0.1` if `FileBackedOutputStream` is meant to be supported in the future.

[Datatype-Joda] Direct comparison of Boolean object in JacksonJodaDateFormat

Severity	Low
Status	Reported
id	ADA-JACKSON-JODA-2023-1
Component	JacksonJodaDateFormat

Multiple locations in the `JacksonJodaDateFormat` class compare Boolean object directly with the == operator which means an identity equality check of the object instead of a value equality check is done. It is because `Boolean` is the object wrapper of the primitive type **`boolean`**. This could cause wrong checking results and possible `NullPointerException`.

Source direct link:

https://github.com/FasterXML/jackson-datatype-joda/blob/f8478ccc610e43f72304531e905d3477355a10f6/src/main/java/com/fasterxml/jackson/datatype/joda/cfg/JacksonJodaDateFormat.java#L134-L135

```
47          if ((adjustTZ != _adjustToContextTZOverride)
48                  || (writeZoneId != _writeZoneId)) {
```

Source direct link:

https://github.com/FasterXML/jackson-datatype-joda/blob/f8478ccc610e43f72304531e905d3477355a10f6/src/main/java/com/fasterxml/jackson/datatype/joda/cfg/JacksonJodaDateFormat.java#L184

```
47          if (adjustToContextTZOverride == _adjustToContextTZOverride) {
```

Source direct link:

https://github.com/FasterXML/jackson-datatype-joda/blob/f8478ccc610e43f72304531e905d3477355a10f6/src/main/java/com/fasterxml/jackson/datatype/joda/cfg/JacksonJodaDateFormat.java#L195

```
47          if (writeZoneId == _writeZoneId) {
```

In Java, the == sign is used for checking equality between two objects or values. If it is used with primitive values, the == sign means checking the equality of values, if it is used for objects, it means

checking for identity not values. Thus using == on primitive **boolean** values and Boolean objects could get different results.

For example, the following code snippet would print **true**.

```
1 boolean a = true;
2 boolean b = true;
3 System.out.println(a == b);
```

The following code snippet would print **false**, even if the value of a and b are both **true**. This is because they are two different Boolean objects and thus their identity is different.

```
1 Boolean a = new Boolean(true);
2 Boolean b = new Boolean(true);
3 System.out.println(a == b);
```

But Java does have an auto-boxing/unboxing mechanism that automatically transforms between primitive values and its object wrapper. (For example, automatically transfer between Boolean and **boolean** if necessary). Thus there are some cases where `a == b` could still return **true** even if they are both objects.

For example, the following code snippet would print **true**, even if a and b are both objects, not primitive values because the assignment of primitive value **true** to the object makes Java recognize that it is still a primitive comparison when using ==, thus the == in this case is comparing value equality instead of object identity.

```
1 Boolean a = true;
2 Boolean b = true;
3 System.out.println(a == b);
```

The following code snippet would also print **true**, even if b is an object. It is because a is a primitive value and Java assumes automatically that the == operation is a value equality check.

```
1 boolean a = true;
2 Boolean b = new Boolean(true);
3 System.out.println(a == b);
```

There is also a special case, because **null** is a valid value for an object, thus a Boolean object could have null as a value. As shown above, it is possible to compare a primitive **boolean** value with a Boolean object thanks to auto-unboxing. But this also causes problems. If the Boolean object is null, the auto-unboxing process will throw NullPointerException when auto-unboxing is performed before the comparison. That unexpected NullPointerException could crash the program if unhandled. This is a special case because no **boolean** to Boolean == operation is found in the code. All of the above-shown vulnerable locations are just two Boolean object comparisons using the == operator.

The following code snippet will throw `NullPointerException` when executed.

```
1 boolean a = true;
2 Boolean b = null;
3 System.out.println(a == b);
```

So in conclusion, using == for comparing 2 `Boolean` objects has the possibility that it is not comparing the boolean values, and it also has the chance to throw unexpected `NullPointerException`, thus it may be a possible problem.

Mitigation

It is suggested to use the primitive **`boolean`** directly if possible. If the use of `Boolean` is needed, null checking is suggested if the source of the `Boolean` values is untrusted and the `equals()` method of the `Boolean` object should be used instead of the == operator to ensure it is value equality checking instead of identity equality checking. Sometimes, null checking may not be necessary if the value is from a trusted source.

The following code snippet correctly compares the value of the two `Boolean` objects a and b and prints **`true`**.

```
1 Boolean a = new Boolean(true);
2 Boolean b = new Boolean(true);
3 System.out.println(a.equals(b));
```

[Datatype-Joda] Unnecessary auto-boxing/unboxing in IntervalDeserializer

Severity	Informational
Status	Reported
id	ADA-JACKSON-JODA-2023-2
Component	IntervalDeserializer

In `IntervalDeserializer`, there is a need to parse the starting and ending interval from given `String` to **`long`** values. It uses the `Long.valueOf(String)` method to parse the given string and store it into two primitive **`long`** variables `start` and `end`. As the `Long.valueOf(String)` method returns a `Long` wrapper object instead of the primitive type **`long`** value, an auto-unboxing process has been done before the storing actions. This could affect performance when this method is invoked many times.

In Java, all primitive types have their object wrapper class. For example, the primitive type **`long`** has its object wrapper class `Long`. Java provides an auto-boxing/unboxing mechanism to transform between the primitive value and its wrapper objects when necessary. For example, if a method returns a `Long` object and is stored in a primitive **`long`** variable, auto-unboxing is done on the returned `Long` object before storing the operation. This automatic process takes an extra step and could take time. A single auto-boxing/unboxing process may less insignificant time, but a large amount of those processes could be a possible performance issue.

Source direct link:

https://github.com/FasterXML/jackson-datatype-joda/blob/f8478ccc610e43f72304531e905d3477355a10f6/src/main/java/com/fasterxml/jackson/datatype/joda/deser/IntervalDeserializer.java#L69-L80

```
69          long start, end;
70          String str = value.substring(0, index);
71          Interval result;
72
73          try {
74              // !!! TODO: configurable formats...
75              if (hasSlash) {
76                  result = Interval.parseWithOffset(value);
77              } else {
78                  start = Long.valueOf(str);
79                  str = value.substring(index + 1);
80                  end = Long.valueOf(str);
```

Mitigation

As the type of the `start` and `end` variables are both primitive **`long`**, it is suggested to use the `Long.parseLong(String)` method instead which directly returns a primitive **`long`** value instead of wrapping them in the `Long` class. That could slightly increase the performance by eliminating an auto-boxing/unboxing roundtrip operation.

[Dataformats-Text-Yaml] Unused conditional check in CsvDecoder

Severity	Informational
Status	Reported
id	ADA-JACKSON-TEXT-2023-1
Component	CsvDecoder

In the `CsvDecoder::_nextQuotedString()` method, there is an empty control flow with an `if` conditional check. That check simply checks the `checkLF` boolean value and does nothing for either case. Although this does not affect the code, it is suggested not to have this kind of dangling and unused control flow in the code. It is assumed that this conditional branch exists because of testing or future enhancement, but it is better to remove or comment it out until it is really necessary.

Source direct link:

https://github.com/FasterXML/jackson-dataformats-text/blob/fc40a6371660379cd805cb12afc16b9948c2779f/csv/src/main/java/com/fasterxml/jackson/dataformat/csv/impl/CsvDecoder.java#L840-L841

```
840  if (checkLF) { // had a "hanging" CR in parse loop; check now
841  }
```

Mitigation

It is suggested to remove that unnecessary conditional check or comment them out if there is logic planned for this conditional check in future development.

[Dataformats-Text-Yaml] Unexpected NullPointerException in YAMLParser

Severity	Low
Status	Fixed
id	ADA-JACKSON-TEXT-2023-2
Component	YAMLParser

Attackers can crash the application that adopts the Jackson-dataformats-text library which does not handle the unexpected `NullPointerException`. It will create Denial-of-Service if the vulnerable application is meant to be running as a web service, this cause legitimate users of the vulnerable applications becomes a victim of Denial-of-Service.

In `YAMLParse::getNumberValueDeferred()` / `YAMLParse::_parseNumericValue()` / `YAMLParse::_parseIntValuev()` methods, the `lenght()` method of the String object `_cleanedTextValue` is called. This could cause an unexpected `NullPointerException` when the previous steps make `_cleanedTextValue` become null with an invalid input value. Some reasons for the null value in `_cleanedTextValue` include the case that the input triggered buffering and the code was not using previously decoded int/long value but assuming the buffered number is a String.

Source direct link:

https://github.com/FasterXML/jackson-dataformats-text/blob/755a907e8a0d6fe0d0e43ef964565ec7e306c331/yaml/src/main/java/com/fasterxml/jackson/dataformat/yaml/YAMLParser.java#L1025-L1054

```
1025      @Override
1026      public Object getNumberValueDeferred() throws IOException {
1027          // 01-Feb-2023, tatu: ParserBase implementation does not quite
                work
1028          //   due to refactoring. So let's try to cobble something
                together
1029
1030          if (_currToken == JsonToken.VALUE_NUMBER_INT) {
1031              // For integrals, use eager decoding for all ints, longs (
                    and
1032              // some cheaper BigIntegers)
1033              if (_cleanedTextValue.length() <= 18) {
1034                  return getNumberValue();
1035              }
1036              return _cleanedTextValue;
```

```
1037            }
1038            if (_currToken != JsonToken.VALUE_NUMBER_FLOAT) {
1039                _reportError("Current token ("+_currToken+") not numeric,
                        can not use numeric value accessors");
1040            }
1041
1042            // For FP, see if we might have decoded values already
1043            if ((_numTypesValid & NR_BIGDECIMAL) != 0) {
1044                return _getBigDecimal();
1045            }
1046            if ((_numTypesValid & NR_DOUBLE) != 0) {
1047                return _getNumberDouble();
1048            }
1049            if ((_numTypesValid & NR_FLOAT) != 0) {
1050                return _getNumberFloat();
1051            }
1052
1053            // But if not, same as BigInteger, let lazy/deferred handling
                    be done
1054            return _cleanedTextValue;
1055        }
```

Mitigation

Add some checking for the buffered number and return the buffered int/long value if found. Then add a null check at the end and report invalid `_cleanedTextValue`.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64662

Upstream fix

https://github.com/FasterXML/jackson-dataformats-text/pull/446

[Dataformat-XML] Unexpected ArrayIndexOutOfBoundsException in XMLTokenStream with SJSXP

Severity	Low
Status	Fixed
id	ADA-JACKSON-XML-2023-1
Component	XMLTokenStream

Attackers can crash the application that adopts the Jackson-dataformat-xml library which does not handle the unexpected `ArrayIndexOutOfBoundsException`. It will create Denial-of-Service if the vulnerable application is meant to be running as a web service, this cause legitimate users of the vulnerable applications becomes a victim of Denial-of-Service.

In `XmlTokenStream::_collectUntilTag()` method, there is an infinite while loop to loop through the provided XML string (through _xmlReader) token by token. The loop only exits by return statements when a valid character (`XMLStreamConstants.START_ELEMENT`, `XMLStreamConstants.END_ELEMENT` or `XMLStreamConstants.END_DOCUMENT`) is found. If the provided XML string is invalid without those characters, it will continue to loop through the whole XML String and eventually throw `ArrayIndexOutOfBoundsException` when `_xmlReader` has no more characters that can be returned by the `next()` method. Besides, there are also some other methods that depend on those `END_ELEMENT` to stop looping out-of-bound.

REMARK: This only happens when the JDK default Stax XML parser is used.

Source direct link:

https://github.com/FasterXML/jackson-dataformat-xml/blob/f5cc10a910b153ec9f162c6d212212b39bfc889d/src/main/java/com/fasterxml/jackson/dataformat/xml/deser/XmlTokenStream.java#L548-L589

```
548         CharSequence chars = null;
549         while (true) {
550             switch (_xmlReader.next()) {
551             case XMLStreamConstants.START_ELEMENT:
552                 return (chars == null) ? "" : chars.toString();
553 
554             case XMLStreamConstants.END_ELEMENT:
555             case XMLStreamConstants.END_DOCUMENT:
556                 return (chars == null) ? "" : chars.toString();
557 
```

```
558                 // note: SPACE is ignorable (and seldom seen), not to be
                        included
559                 case XMLStreamConstants.CHARACTERS:
560                 case XMLStreamConstants.CDATA:
561                     // 17-Jul-2017, tatu: as per [dataformat-xml#236], need
                            to try to...
562                     {
563                         String str = _getText(_xmlReader);
564                         if (chars == null) {
565                             chars = str;
566                         } else  {
567                             if (chars instanceof String) {
568                                 chars = new StringBuilder(chars);
569                             }
570                             ((StringBuilder)chars).append(str);
571                         }
572                     }
573                     break;
574                 default:
575                     // any other type (proc instr, comment etc) is just
                            ignored
576                 }
577             }
578         }
```

Mitigation

Wrapping the `ArrayIndexOutOfBoundsException` with the `JsonParseException` and also changing the while loop criteria to ensure there is more token left with the `hasToken()` method.

Reported Issues

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64655

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64659

https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=64967

Upstream fix

https://github.com/FasterXML/jackson-dataformat-xml/pull/619

[Dataformats-XML] XML External Entity vulnerability in XMLFactory

Severity	Moderate
Status	Reported
id	ADA-JACKSON-XML-2023-2
Component	XMLFactory

Attackers can perform an XXE attack through the Jackson-dataformat-XML library with a malicious serialized XML when a victim uses an application that adopts the jackson-dataformat-xml library and is wrongly configured with the XMLInputFactory. This could result in illegal information leaking from the victim's computer where the vulnerable application is launched.

Traditionally, the JDK XMLParser is vulnerable to XML External Entity attack (XXE attack) if the XMLInputFactory of the parser is configured to support External Entities and is used to parse untrusted XML input. XML External Entity is an XML feature to allows reading data from a URL, which means that it can read local files with the `file:///` tag. An example is given below. When an XMLInputFactory is configured to support External Entities and its parser is used to parse the following XML, the data of the `/etc/passwd` will be included in the `foo` tag after parsing.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE foo [
3    <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
4 <foo>&xxe;</foo>
```

By default, if no XMLInputFactory is specified, the `Jackson XMLFactory` will create a default XMLInputFactory and then disable the support for both DTD and external entities. These actions make the default XMLInputFactory not vulnerable to XXE attack.

Source direct link:

https://github.com/FasterXML/jackson-dataformat-xml/blob/d7ce61f43370dac3b1c144b72eb953303a91f6db/src/main/java/com/fasterxml/jackson/dataformat/xml/XmlFactory.java#L122-L128

```
122         if (xmlIn == null) {
123             xmlIn = StaxUtil.defaultInputFactory(getClass().
                    getClassLoader());
124             // as per [dataformat-xml#190], disable external entity
                    expansion by default
125             xmlIn.setProperty(XMLInputFactory.
                    IS_SUPPORTING_EXTERNAL_ENTITIES, Boolean.FALSE);
```

```
126             // and ditto wrt [dataformat-xml#211], SUPPORT_DTD
127             xmlIn.setProperty(XMLInputFactory.SUPPORT_DTD, Boolean.
                    FALSE);
128         }
```

However the disable of DTD and external entities are only performed when no `XMLInputFactory` is provided to the `XMLFactory` constructor. If the user generates a custom `XMLInputFactory` by calling `javax.xml.stream.XMLInputFactory.newInstance()` and passes it to the `XMLFactory` contructor, the disable of DTD and external entities support is not executed and thus the XML parsing will be vulnerable to XXE attacks.

Proof of concept for the XXE attack

Sample file for the proof of concept

1. test.xml

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <!DOCTYPE foo [
3    <!ENTITY xxe SYSTEM "file:///etc/passwd" > ]>
4 <foo>&xxe;</foo>
```

2. TestXXE.java

```
1  import java.io.*;
2  import java.nio.file.*;
3  import com.fasterxml.jackson.core.*;
4  import com.fasterxml.jackson.databind.*;
5  import com.fasterxml.jackson.dataformat.xml.*;
6  import javax.xml.stream.XMLInputFactory;
7
8  public class TestXXE {
9      public static void main(String[] args) throws Exception {
10         String XML = new String(Files.readAllBytes(Paths.get("test.xml"
               )));
11         XMLInputFactory factory = XMLInputFactory.newInstance();
12
13         try (JsonParser p = new XmlMapper(factory).createParser(XML)) {
14             while (p.nextToken() != null) {
15                 try {
16                     System.out.println(p.getText());
17                 } catch (Exception e) {}
18             }
19         }
20     }
21 }
```

3. TestNoXXE.java

```java
import java.io.*;
import java.nio.file.*;
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import com.fasterxml.jackson.dataformat.xml.*;
import javax.xml.stream.XMLInputFactory;

public class TestNoXXE {
    public static void main(String[] args) throws Exception {
        String XML = new String(Files.readAllBytes(Paths.get("test.xml"
            )));

        try (JsonParser p = new XmlMapper().createParser(XML)) {
            while (p.nextToken() != null) {
                try {
                    System.out.println(p.getText());
                } catch (Exception e) {}
            }
        }
    }
}
```

Steps for the proof of concept

```
# Create temporary directory
mkdir xxe
cd xxe

# Retrieve maven
curl -L https://archive.apache.org/dist/maven/maven-3/3.6.3/binaries/
    apache-maven-3.6.3-bin.zip -o maven.zip
unzip maven.zip -d ./
rm -rf maven.zip

# Clone and build the jackson-dataformat-xml library
git clone https://github.com/FasterXML/jackson-dataformat-xml
cd jackson-dataformat-xml
git checkout d7ce61f43370dac3b1c144b72eb953303a91f6db
../apache-maven-3.6.3/bin/mvn clean package shade:shade

# Compile the PoC Code
cd ../
javac -cp jackson-dataformat-xml/target/jackson-dataformat-xml-2.17.0-
    SNAPSHOT.jar TestXXE.java
javac -cp jackson-dataformat-xml/target/jackson-dataformat-xml-2.17.0-
    SNAPSHOT.jar TestNoXXE.java

# Run the PoC and exploit the XXE
## Should print out the content of /etc/passwd
java -classpath .:jackson-dataformat-xml/target/jackson-dataformat-xml
```

```
    -2.17.0-SNAPSHOT.jar TestXXE
## Should throw a WstxParsingException wrapped by a JsonParseException
java -classpath .:jackson-dataformat-xml/target/jackson-dataformat-xml
    -2.17.0-SNAPSHOT.jar TestNoXXE
```

The `TestXXE` will create an `XMLInputFactory` object with `XMLInputFactory.newInstance()` and by default, it supports both DTD and external entities, then it is passed to the `XMLFactory` to create an `XMLMapper` object. While `TestNoXXE` use the default constructor of the `XMLFactory` to create an `XMLMapper` object thus will create a default `XMLInputFactory` and disable both DTD and external entities. Both of them use the created `XMLMapper` object to create a JsonParser and parse `test.xml` which contains XXE attacks to display the content of `/etc/passwd`. Since external entities have been enabled for `TestXXE`, the content of `/etc/passwd` is printed on screen while `TestNoXXE` will throw an Exception because external entities are not supported and thus variable `$xxe` is not set and result in `WstxParsingException`.

Mitigation

It could be arguable that if the user of the jackson-dataformat-xml decided to use their XMLInputFactory, they should be configuring the XMLInputFactory correctly before passing to the jackson-dataformat-xml library and in some cases, the user of the library may want to enable external entities support. Thus it is not sure if jackson-dataformat-xml library wants to handle this. The easier way to allow flexibility and increase security is to turn off the support for DTD and external entities at the constructor by default, no matter if it is provided by the user or created lively. Then provide additional helper methods to allow configuring these properties. This may decrease the chance of possible XXE vulnerabilities because of the wrongly configured XMLInputFactory object passed to the jackson-dataformat-xml library.

Reported Issues

Reported by findsecbug.