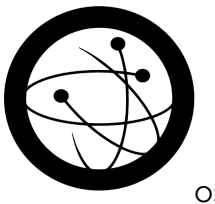
PRESENTS

Jackson-Core and Jackson-Databind Security Audit

In collaboration with the Jackson projects maintainers and The Open Source Technology Improvement Fund, Inc (OSTIF).



Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>
Date: 1st November 2022

Table of contents

Executive summary

In august and september 2022, OSTIF solicited Ada Logics to carry out a security audit for the Jackson-core and Jackson-databind projects. Jackson-databind and Jackson-core are libraries used for data parsing and data binding. The primary use case is for JSON data, but other types of data such as XML, CSV and protobuf can be parsed as well. The goal of the engagement was to conduct a holistic security assessment of both projects, which for each project involved four high-level tasks:

- Threat model formalisation.
- Manual code auditing.
- Building a comprehensive and substantial fuzzing suite.
- Integrate the fuzzing suite into Jacksons CI integration.

Throughout the audit, Ada Logics' auditing team met with Tatu Saloranta from the Jackson projects to discuss the process as well as findings.

Key findings

The main assessment from this audit is that Jackson-core and Jackson-databind have a strong security posture. Some issues were found during the engagement, but the overall conclusion is that the libraries are well-written and safe to use. The audit found 12 issues that could affect Jackson users' security. The below table overview illustrates the severity of the findings.

Severity

Severity	# of findings
High	**3**
Medium	**5**
Informational	**4**

Issue categorization

Type	# of findings
Missing bounds checks	4
Stack overflow DoS issues	4
Usability risks	4

Notable issues

The primary security findings of this audit are 2 high-severity CVEs in Jackson-databind that can lead to denial of service, and both of these issues were found by way of fuzzing:

- CVE-2022-42003 | CVSS: 7.5 **HIGH**
- CVE-2022-42004 | CVSS: 7.5 **HIGH**

Project Summary

The auditors of Ada Logics were:

Name	Title	Email
Adam Korczynski	Security Engineer	Adam@adalogics.com
David Korczynski	Security Researcher	David@adalogics.com

The maintainers of Jackson-core and Jackson-databind that were involved were:

Name	Title	Email
Tatu Saloranta	Jackson maintainer	Tatu@fasterxml.com

The following facilitators of OSTIF were engaged in the audit:

Name	Title	Email
Derek Zimmer	Executive Director, OSTIF	Derek@ostif.org
Amir Montazery	Managing Director, OSTIF	Amir@ostif.org

Project Timeline

Events and milestones of the audit.

August 15 2022	Kick-off meeting
September 6 2022	Status meeting #1
September 18 2022	Status meeting #2
September 31 2022	CVEs applied for for issue ADA-Jackson-10 and ADA-Jackson-11
October 2 2022	CVEs assigned by MITRE
1st November 2022	The audit report is published

Audit scope

The following assets were in scope of the audit.

Jackson-core

Repository	https://github.com/FasterXML/jackson-core
Language	Java

Jackson-databind

Repository	https://github.com/FasterXML/jackson-databind
Language	Java

Initial assessment

As part of the audit, Ada Logics made an initial assessment of the existing security efforts of Jackson-core and Jackson-databind. The two projects in scope were found to have a strong security posture in place. Much work around the security of the projects had been carried out by maintainers and contributors who visibly had been auditing the projects ad hoc in recent years. Besides an XXE vulnerability[1], most of the previous findings were related to deserialization attacks leading to remote code execution or SSRF attacks. As of ultimo October 2022, 70 reviewed advisories are registered for Jackson-databind of which the vast majority are related to deserialization attacks[2]. As such, the vulnerable code being exploitable in deserialization attacks via Jackson-databind has been in other classes not related to Jackson-core and Jackson-databind. Dozens of CVE's assigned to Jackson-databind over the last years have been due to vulnerable code in other projects that could be triggered via deserialization into these classes. At the time of the audit, deserialization attacks were deemed out of scope, because so much effort had been dedicated to them in the past few years.

From our point of view, Jackson-core and Jackson-databind are secure projects with positive security habits in place. Few security issues had been found in the implementation of data parsing and databinding, and most found security issues had come from insecure classes.

One area the projects were lacking was fuzzing. Parsing and data binding functionality is especially well suited to test with fuzzing due to the complexity involved. Two areas where fuzzing could be improved was test coverage and CI integration.

As part of the initial assessment, Ada Logics also perused the documentation of Jackson-core and Jackson-databind and found that a great deal of effort had been put into documenting usage, releases and security of the project. For example, a blog post had been released to cover deserialization vulnerabilities in depth[3], and well-defined policy on the criteria for assigning CVEs to polymorphic deserialization[4].
The auditors also found that both projects had plenty of examples on common usage as well as well-defined Wikis describing optionals features. This proved tremendously helpful during the audit, and we believe the investment is worthwhile for users and auditors; A well-defined documentation is a contributing factor to improved security work in that it allows auditors clear guidelines to get started with reviewing the project.

---

[1] https://github.com/advisories/GHSA-288c-cq4h-88gq

[2] https://github.com/advisories?page=1&query=type%3Areviewed+ecosystem%3Amaven+jackson-databind

[3] https://cowtowncoder.medium.com/on-jackson-cves-dont-panic-here-is-what-you-need-to-know-54cd0d6e8062

[4] https://github.com/FasterXML/jackson/wiki/Jackson-Polymorphic-Deserialization-CVE-Criteria

Fuzzing continuously with OSS-Fuzz

Fuzzing is a vital component when ensuring the security of software. A goal of this security audit was to contribute to Jackson-core and Jackson-databinds testing infrastructure by improving their fuzzing suites. A big part of the engagement was dedicated to this part which resulted in several found issues. When improving the fuzz testing of a software project, it is critical to do it in such a manner that the fuzzers will run continuously - even after the initial work of writing the fuzzers. To do this, the work done on fuzzing Jackson-core and Jackson-databind in this audit evolved around OSS-Fuzz. OSS-Fuzz is an open source service that helps assist in this context by handling the infrastructure of running fuzzers and reporting issues for important open source projects. Jackson-core and Jackson-databind were both integrated into OSS-Fuzz prior to this engagement start. However, there was only a single fuzzer implemented for each project and the goal was, therefore, to expand the fuzzing suite for each project. A key benefit from integrating the fuzzers by way of OSS-Fuzz is that the fuzzers run continuously, both during and after the audit.

Fuzzing is particularly useful for data processing and parsing which makes Jackson-core and Jackson-databind optimal projects to fuzz. The fuzzers found several issues of varying nature: Some crashes revealed lack of bounds checks across many similar class methods in both Jackson-core and Jackson-databind. ADA-Jackson-1, ADA-Jackson-2, ADA-Jackson-3, ADA-Jackson-4 are examples of such cases where similar methods had similar lack of sufficient data validation. Many crashes were reported for each of these four issues, and in the report they are presented based on their root cause.

The issues ADA-Jackson-9, ADA-Jackson-10, ADA-Jackson-11 and ADA-Jackson-12 were also found by fuzzers. We consider these to be hard-to-find edge cases that existed in a single part of the given software project. Of these, 2 were assigned CVEs, whereas 1 was fixed without CVE and 1 was left unfixed. The issue that was not assigned CVE was considered such a rare use case that it was expected that no users were affected by it.

Ada Logics wrote a total of 10 new fuzzers targeting both Jackson-core and Jackson-databind. The fuzzers can each be found in the respective directories in the OSS-Fuzz repository, namely:

- Jackson-core: https://github.com/google/oss-fuzz/tree/master/projects/jackson-core
- Jackson-databind: https://github.com/google/oss-fuzz/tree/master/projects/jackson-databind

The following table lists the fuzzers, the project they target as well as links to the source code of the given fuzzer:

Jackson library	Fuzzer	Source code
core	DataInputFuzzer	OSS-Fuzz link
core	ParseNextTokenFuzzer	OSS-Fuzz link
core	UTF8GeneratorFuzzer	OSS-Fuzz link
core	WriterBasedJsonGeneratorFuzzer	OSS-Fuzz link

databind	ConvertValueFuzzer	OSS-Fuzz link
databind	ObjectReader2Fuzzer	OSS-Fuzz link
databind	AdaLObjectRead3Fuzzer	OSS-Fuzz link
databind	ObjectReaderRandomClassFuzzer	OSS-Fuzz link
databind	ObjectWriterFuzzer	OSS-Fuzz link
databind	ReadTreeFuzzer	OSS-Fuzz link

All development of the fuzzers was carried out in open source; When modifications and/or additions were developed, they were added to the OSS-Fuzz build system so that the improvements would have immediate effect. This allowed the auditors to iterate multiple times in the environment in which the fuzzers were meant to run. During that process, Tatu Saloranta fixed issues when they were found, which meant that the fuzzers would be unblocked to test for more bugs.
By the end of the audit, all fuzzers are running on OSS-Fuzz and will keep testing the code base. The following overview gives an insight into the compute power behind the fuzzers:

Jackson library	Fuzzer	Total executions at end of audit	Total hours of fuzzing
core	DataInputFuzzer	43,214,654,970	2,110.5
core	ParseNextTokenFuzzer	483,205,790,590	2,318.9
core	UTF8GeneratorFuzzer	10,205,677,340	2,155.2
core	WriterBasedJsonGeneratorFuzzer	10,674,031,329	2,246.5
databind	ConvertValueFuzzer	26,953,577,839	1,617.7
databind	ObjectReader2Fuzzer	369,515,044	1,158.1
databind	AdaLObjectRead3Fuzzer	9,748,470,799	1,184.6
databind	ObjectReaderRandomClassFuzzer	15,399,814	935.6
databind	ObjectWriterFuzzer	6,034,945,982	1,707.8
databind	ReadTreeFuzzer	21,047,488,812	1,582.5

The fuzzers were written based on the threat model that we created. In essence, the threat model was used to delineate the attack surface of the Jackson libraries and where to focus when attacking the project. This was used when developing the fuzzers such that the fuzzers focus on testing code in Jackson that accepts untrusted input, or have particularly sensitive operations. The threat model also found that Jackson should maintain a safe state given the usage of its classes and methods. As such, we created a fuzzer that focused on fuzzing jackson as a stateful application by calling a sequence of Jackson's API in a manner

determined by the data from the fuzzer. This stateful fuzzing led to finding several security and reliability issues.
For example, the 2 issues leading to CVE's were found not only by passing test data to the target API, but also by randomizing the state in which Jackson-databind should run. The fuzzer, `ObjectReader2Fuzzer`, does this by declaring an array of all possible deserialization features that Jackson-databind offers:

```
DeserializationFeature[] deserializationfeatures = new
DeserializationFeature[]{DeserializationFeature.USE_BIG_DECIMAL_FOR_FLOATS,
DeserializationFeature.USE_BIG_INTEGER_FOR_INTS,
DeserializationFeature.USE_JAVA_ARRAY_FOR_JSON_ARRAY,
DeserializationFeature.READ_ENUMS_USING_TO_STRING,
DeserializationFeature.ACCEPT_SINGLE_VALUE_AS_ARRAY,
DeserializationFeature.UNWRAP_ROOT_VALUE,
DeserializationFeature.UNWRAP_SINGLE_VALUE_ARRAYS,
DeserializationFeature.ACCEPT_EMPTY_ARRAY_AS_NULL_OBJECT,
DeserializationFeature.ACCEPT_EMPTY_STRING_AS_NULL_OBJECT,
DeserializationFeature.ACCEPT_FLOAT_AS_INT,
DeserializationFeature.ADJUST_DATES_TO_CONTEXT_TIME_ZONE,
DeserializationFeature.READ_DATE_TIMESTAMPS_AS_NANOSECONDS,
DeserializationFeature.READ_UNKNOWN_ENUM_VALUES_AS_NULL,
DeserializationFeature.READ_UNKNOWN_ENUM_VALUES_USING_DEFAULT_VALUE,
DeserializationFeature.FAIL_ON_IGNORED_PROPERTIES,
DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
DeserializationFeature.FAIL_ON_INVALID_SUBTYPE,
DeserializationFeature.FAIL_ON_NULL_FOR_PRIMITIVES,
DeserializationFeature.FAIL_ON_NUMBERS_FOR_ENUMS,
DeserializationFeature.FAIL_ON_READING_DUP_TREE_KEY,
DeserializationFeature.FAIL_ON_UNRESOLVED_OBJECT_IDS,
DeserializationFeature.FAIL_ON_MISSING_CREATOR_PROPERTIES,
DeserializationFeature.WRAP_EXCEPTIONS,
DeserializationFeature.FAIL_ON_TRAILING_TOKENS,
DeserializationFeature.EAGER_DESERIALIZER_FETCH};
```

Each feature can either be enabled or disabled, and depending on its settings, Jackson-databind will take certain execution paths that are only available with certain features enabled or disabled. Therefore, `ObjectReader2Fuzzer` either enables or disables all features in each fuzz iteration:

```
for (int i = 0; i < deserializationfeatures.length; i++) {
    if (data.consumeBoolean()) {
        r = r.with(deserializationfeatures[i]);
    } else {
        r = r.without(deserializationfeatures[i]);
    }
}
```

Upon discovering the two issues leading to CVE's, an extended fuzzer for the ObjectReader and ObjectMapper classes was added - `AdaLObjectReader3Fuzzer` - which also includes

randomization of `SerializationFeatures` for the `ObjectReader` as well as `MapperFeatures` and `DefaultTypings` for the `ObjectMapper` from which the `ObjectReader` gets created. At the end of the audit, this fuzzer has not been running long enough to determine whether any low-hanging issues are to be found under these settings, but any issues will be reported directly to the project maintainers. Both `ObjectReader2Fuzzer` and `AdaLObjectReader3Fuzzer` both proceed to invoke a number of methods of each class with data from the fuzz test case.

The fuzzing was the most fruitful task of the engagement in terms of finding security vulnerabilities. This is likely due to fuzzing as a technique being particularly good when faced with logic exposed by Jackson, e.g. parsing and deserialisation, as well as the Jackson libraries having previously been exposed to limited fuzzing.

Threat model formalisation

In this section we outline our threat model of Jackson-databind and Jackson-core. The goal of this effort was two-fold. First, to create a formalisation that can be used by Jackson maintainers, security researchers and consumers of Jackson to understand the security setting of Jackson. Second, we use the threat model in this engagement to help guide and direct where and what to focus on when assessing the security of Jackson.

Audience

The audience for the threat model is the three target groups:

1. Security researchers who wish to contribute to the security posture of the Jackson ecosystem.
2. The maintainers of Jackson-databind and Jackson-core.
3. Users of the Jackson projects.

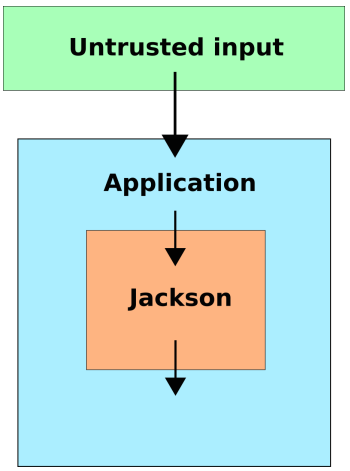
For researchers and the Jackson maintainers, the threat model offers a definition of what security is for Jackson-databind and Jackson-core. The model helps to distinguish security-relevant issues from non-security-relevant issues.

For users of Jackson, the threat model can be used to assist deployment and use of Jackson. This includes following best practices and ensuring proper guards when using Jackson, such as when using APIs that accept trusted input then proper guards must be in place.

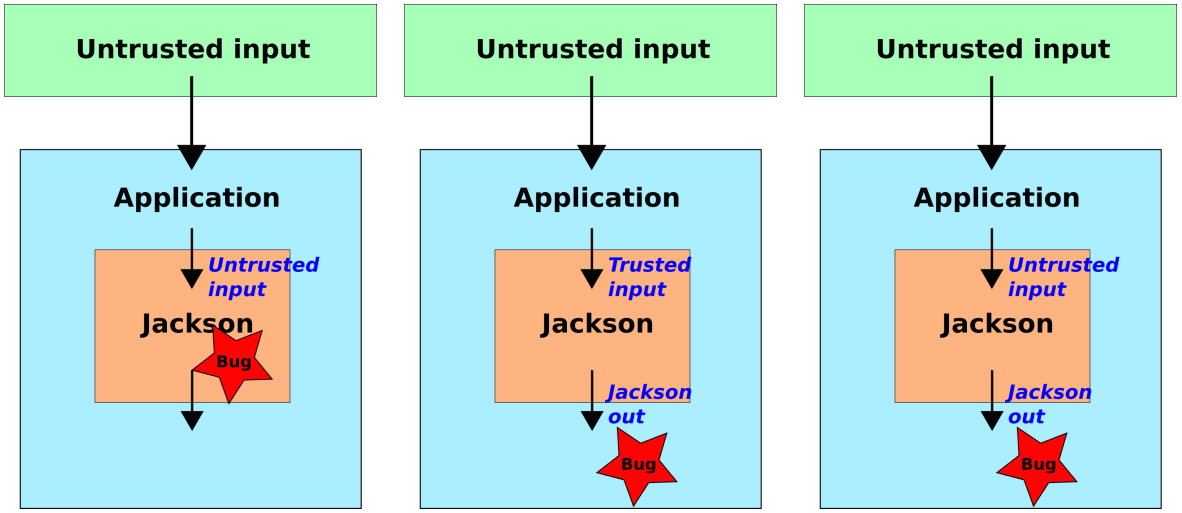
It is expected that the threat model develops slightly over time. New security issues and evolving use cases will be empirical factors that will modify the security boundaries of the threat model. The community is encouraged to suggest changes to the threat model.

Jackson use case and possible issues

Jackson-core and Jackson-data bind are libraries, and they are each composed of a single logical component. In this sense, the usage of Jackson is simple in that the libraries are used within applications. The key goal of Jackson security is that when applications use Jackson following the intended guidelines then Jackson must ensure it does not violate the confidentiality, integrator or availability of the application or environment in which the library is used. We can visualise this is in a simple manner as follows:



There are two trust boundaries to Jackson, the input it takes from the application and the output returned. In essence there is a single high-level type of security issue in Jackson. This is shown below in the form of issue type 1.



Issue type 1.
Jackson handles untrusted input and a bug occurs. This is the main type of security issue Jackson is concerned with as Jackson is meant to be hardened against untrusted input.
An issue of this type is a clear security vulnerability in Jackson.

Issue type 2.
Jackson handles trusted input and no bug occurs in Jackson. The output of jackson is then used further by the application, and a bug occurs.
An issue of this type is not a security vulnerability in Jackson.

Issue type 3.
Jackson handles untrusted input and no bug occur in Jackson's handling of the input. The application uses the return value of Jackson, and a bug occurs. This is not an issue in Jackson.
An issue of this type is not a security vulnerability in Jackson.

In addition to this, there is also the case where Jackson exhibits a failure after receiving trusted input, however, this is also not a security vulnerability in Jackson as the trusted input is expected to be non-malevolent. The main issue of concern is issue type 1. This is where a

security vulnerability exists in Jackson that can be triggered and exploited in a context where Jackson is meant to be hardened against untrusted input.
Issue type 2 and issue type 3 can lead to all sorts of security issues in a given application. However, this is not a consequence of a security vulnerability in Jackson itself, but rather of insecure handling of Jackson. For this reason, it's important that users of Jackson ensure proper usage of Jackson as wrong assumptions lead to potential security issues. In the list of issues we have given several examples of potential problems that may arise in a context like this.

As the Jackson libraries are single-entity components the key to the threat model is identifying the APIs that accept untrusted input.

Jackson input classification

In this section we capture the specific API entry points of Jackson to identify which are trusted and which are untrusted.

- **Trusted input**: This is data passed to Jackson by a trusted user such as an admin user. Trusted input can have security implications but will usually not constitute a security risk, and, most importantly is not a security risk that Jackson cares about as Jackson considers this to be a missing security barrier in the User's code, and not in Jackson's code. There will be exceptions to this, and the defining question is whether *only* an admin user can carry out an attack or if there is *any* way a non-admin can perform it as well. In the case of the former, a given vulnerability does not represent a true security risk.
- **Untrusted input**: Untrusted input is input from non-admin users. All vulnerabilities that can be triggered by way of untrusted input constitute a security risk and should be reported to the Jackson security team.

Below we list out endpoints that accept untrusted and trusted input.

Identifying the untrusted input

A common use case of JSON parsing and databinding is to accept untrusted input. In this section we particularise how untrusted input is handled in Jackson and what the security implications are.

Jackson-databind has the following methods that accept untrusted data:

- `ObjectMapper.readValue();`
- `ObjectMapper.readValues();`
- `ObjectMapper.readTree();`
- `ObjectMapper.createParser();`
- `ObjectReader.createParser();`
- `ObjectReader.readTree();`
- `ObjectReader.readValue();`
- `ObjectReader.readValues();`

Jackson-core considers input to the following APIs untrusted:

- `JsonFactory.createParser();`
- `JsonFactory.createJsonParser();`

These 10 APIs will ordinarily accept untrusted input in the form of a `String`, `byte[]`, `char[]` or another format that carry raw data.
The `createParser()` methods will instantiate and return an implementation of the `com.fasterxml.jackson.core.JsonParser` object depending on the raw input.
Subsequently, the Jackson user will invoke one or several of the JsonParsers methods in their application.
Because the `JsonParser` object is created from untrusted, raw data, it in turn will always be untrusted as well. As such, its methods fall within the scope of the attack surface of untrusted input, in that an attacker might be able to first instantiate a `com.fasterxml.jackson.core.JsonParser` from malicious input and then invoke one or several of its methods to cause harm.

Dataflow of untrusted input

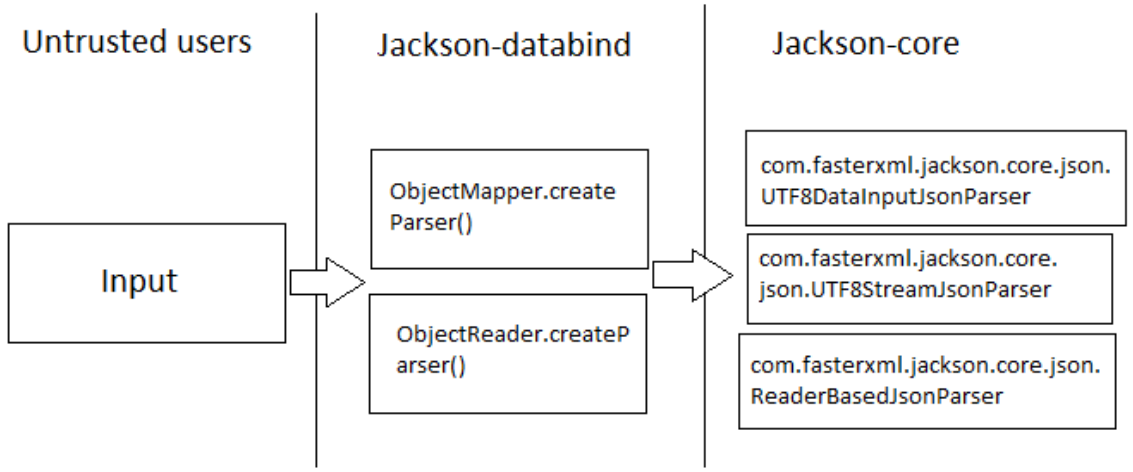
Databind

`ObjectMapper.createParser()` and `ObjectReader.createParser()` methods are wrappers to create parsers from Jackson-core. Different parsers are created from different input types:

ObjectMapper.createParser() input type	**Resulting Jackson-core parser**
DataInput	`UTF8DataInputJsonParser`
`byte[]`	`UTF8StreamJsonParser` or `ReaderBasedJsonParser`
`byte[], int, int`	`UTF8StreamJsonParser` or `ReaderBasedJsonParser`
`InputStream`	`UTF8StreamJsonParser` or `ReaderBasedJsonParser`
`File`	`UTF8StreamJsonParser` or `ReaderBasedJsonParser`
`URL`	`UTF8StreamJsonParser` or `ReaderBasedJsonParser`
`Reader`	`ReaderBasedJsonParser`
`char[], int, int`	`ReaderBasedJsonParser`

`char[]`	`ReaderBasedJsonParser`
`String`	`ReaderBasedJsonParser`

Untrusted data of this method flows as follows:



JsonParser

All APIs that accept a `JsonParser` as an argument can be passed untrusted data. This includes methods that untrusted users are not meant to interact with as a first point of contact. A `JsonParser` object may be passed through the calltree to deep APIs without being sanitized or checked for insecure data. Security issues could exist in methods that untrusted users are not meant to interact directly with, however, if the issue is invoked from any values from a `JsonParser`, it is likely that it can be produced from untrusted input.

Found issues

The following table shows an overview of the security issues found during the audit.

#	ID	Name	Severity
1	ADA-Jackson-1	Missing bounds check for methods that accept byte[]/char[]-with-offsets input	Medium
2	ADA-Jackson-2	Missing bounds check when looping through tokens	Medium
3	ADA-Jackson-3	Missing bounds check when looping through field names	Medium
4	ADA-Jackson-4	Missing bounds check when returning parser value and integer	Medium
5	ADA-Jackson-5	File reads in Jackson projects follow symlinks per default	Informational
6	ADA-Jackson-6	ObjectMapper.writeValue() allows arbitrary file writes	Informational
7	ADA-Jackson-7	Arbitrary file lookup when parsing map key of type URI with StdKeyDeserializer	Informational
8	ADA-Jackson-8	Regex Denial of Service (ReDos) in FromStringDeserializer	Informational
9	ADA-Jackson-9	DoS from stack exhaustion in Jackson-databind	High
10	ADA-Jackson-10	DoS from stack exhaustion in Jackson-databind	High
11	ADA-Jackson-11	DoS from stack exhaustion in Jackson-databind	High
12	ADA-Jackson-12	DoS from stack exhaustion in Jackson-databind	Medium

This section enumerates the issues found during the audit.

Some notes should be made about issue 5, 6, 7 and 8. These issues do imply possible security issues for users if the relevant Jackson project is used insecurely, however, they fall outside of the scope of the security model of the related Jackson project. They have been included on the list to inform users of potential risks and mitigations, but adopters should not expect fixes for these issues in the related Jackson project. From a high level, the solution to these issues is to perform sufficient validation of untrusted data before serializing or deserializing it in Jackson.
Issue 5, 6, 7 and 8 are edge cases that most users would not be exposed to.

ADA-Jackson-1: Missing bounds check for methods that accept byte[]/char[]-with-offsets input

Severity: Medium	Difficulty: High
Fixed: Yes	Vector: CWE-248: Uncaught Exception, CWE-129: Improper Validation of Array Index
URLs: Fixed in: https://github.com/FasterXML/jackson-core/commit/fb3baee23f924a816bb71f8db5ed521d88b93130 Monorail Issue: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=50003	

Description

The fuzzers found an issue in both Jackson-core and Jackson-databind in APIs where an offset and a length were passed as well. Due to missing bounds checks, Jackson-core and Jackson-databind would crash from uncaught exceptions if the passed length did not correspond to the actual length of the data. The issue was identified by the fuzzers to exist in the following methods:

`com.fasterxml.jackson.databind.ObjectReader.readValues(byte[] src, int offset, int length)`
`com.fasterxml.jackson.core.json.UTF8JsonGenerator.writeRaw(String text, int offset, int len)`
`com.fasterxml.jackson.core.json.UTF8JsonGenerator.writeBinary(Base64Variant b64variant, byte[] data, int offset, int len)`

Upon further investigation, Jackson maintainer Tatu, found that every method of this type was missing a bounds check which was then added to every method across the Jackson-core and Jackson-databind code bases. The public issue for that is: https://github.com/FasterXML/jackson-core/issues/811

ADA-Jackson-2: Missing bounds check when looping through tokens

Severity: Medium	Difficulty: Medium
Fixed: Yes	Vector: CWE-248: Uncaught Exception, CWE-129: Improper Validation of Array Index
URLs: Monorail issues: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=49805 https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=50064 https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=50377	

Description
An issue was found by the `DataInputFuzzer` whereby a well-crafted string could cause Jackson-core to crash with an uncaught exception when the string was used to create a parser, and the user would subsequently loop through all the tokens.

This was the snippet that found the issue:

```
JsonFactory jf = new JsonFactory();
try {
    JsonParser jp = jf.createParser(new
MockFuzzDataInput(data.consumeRemainingAsString()));
    while (jp.nextToken() != null) {
        ;
    }
} catch (IOException ignored) {
}
```

Which produced the following stack trace:

```
== Java Exception: java.lang.ArrayIndexOutOfBoundsException: Index 200 out of
bounds for length 200
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser._parseSignedNumber(UTF8D
ataInputJsonParser.java:1139)
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser._parseNegNumber(UTF8Data
InputJsonParser.java:1103)
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser._nextTokenNotInObject(UT
F8DataInputJsonParser.java:734)
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser.nextToken(UTF8DataInputJ
sonParser.java:642)
  at DataInputFuzzer.fuzzerTestOneInput(DataInputFuzzer.java:126)
```

ADA-Jackson-3: Missing bounds check when looping through field names

Severity: Medium	Difficulty: High
Fixed: Yes	Vector: CWE-248: Uncaught Exception, CWE-129: Improper Validation of Array Index
URLs: Monorail issues: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=49805	

Description
An issue was found by the `DataInputFuzzer` whereby a well-crafted string could cause Jackson-core to crash with an uncaught exception when the string was used to create a parser, and the user would subsequently loop through all the field names.

This was the snippet that found the issue:

```
JsonFactory jf = new JsonFactory();
try {
    JsonParser jp = jf.createParser(new MockFuzzDataInput(MALICIOUS_STRING));
    while (jp.nextFieldName() != null) {
        ;
    }
} catch (IOException ignored) {
}
```

Which produced the following stack trace:

```
== Java Exception: java.lang.ArrayIndexOutOfBoundsException: Index 200 out of
bounds for length 200
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser._parseFloat(UTF8DataInpu
tJsonParser.java:1190)
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser._parseUnsignedNumber(UTF
8DataInputJsonParser.java:1083)
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser._nextTokenNotInObject(UT
F8DataInputJsonParser.java:752)
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser.nextFieldName(UTF8DataIn
putJsonParser.java:828)
  at DataInputFuzzer.fuzzerTestOneInput(DataInputFuzzer.java:171)
```

ADA-Jackson-4: Missing bounds check when returning parser value and integer

Severity: Medium	Difficulty: Medium
Fixed: Yes	Vector: CWE-248: Uncaught Exception, CWE-129: Improper Validation of Array Index
URLs: Monorail issues: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=49805	

Description
An issue was found by where by `DataInputFuzzer` a well-crafted string could cause Jackson-core to crash with an uncaught exception when the string was used to create a parser, and the user would subsequently return a value as an integer.

This was the snippet that found the issue:

```
JsonFactory jf = new JsonFactory();
try {
    JsonParser jp = jf.createParser(new MockFuzzDataInput(MALICIOUS_STRING));
    int outInt = jp.getValueAsInt();
} catch (IOException ignored) {
}
```

Which produced the following stack trace:

```
== Java Exception: java.lang.StringIndexOutOfBoundsException: String index out
of range: 3
  at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:48)
  at java.base/java.lang.String.charAt(String.java:712)
  at
com.fasterxml.jackson.core.io.doubleparser.AbstractFloatingPointBitsFromCharSequ
ence.parseHexFloatLiteral(AbstractFloatingPointBitsFromCharSequence.java:331)
  at
com.fasterxml.jackson.core.io.doubleparser.AbstractFloatingPointBitsFromCharSequ
ence.parseFloatingPointLiteral(AbstractFloatingPointBitsFromCharSequence.java:21
5)
  at
com.fasterxml.jackson.core.io.doubleparser.FastDoubleParser.parseDouble(FastDoub
leParser.java:50)
  at
com.fasterxml.jackson.core.io.doubleparser.FastDoubleParser.parseDouble(FastDoub
leParser.java:34)
  at com.fasterxml.jackson.core.io.NumberInput.parseDouble(NumberInput.java:353)
  at com.fasterxml.jackson.core.io.NumberInput.parseAsInt(NumberInput.java:252)
```

```
  at
com.fasterxml.jackson.core.base.ParserMinimalBase.getValueAsInt(ParserMinimalBas
e.java:391)
  at
com.fasterxml.jackson.core.json.UTF8DataInputJsonParser.getValueAsInt(UTF8DataIn
putJsonParser.java:284)
  at DataInputFuzzer.fuzzerTestOneInput(DataInputFuzzer.java:181)
```

ADA-Jackson-5: File reads in Jackson projects follow symlinks per default

Severity: Informational	Difficulty: Informational
Fixed: No	Vector: CWE-125: Out-of-bounds Read

Description
File reads in both Jackson-core and Jackson-databind follow symbolic links per default. This can lead to insecure use cases, where Jackson users unknowingly allow unprivileged users to read any file on the file system through symbolic links. This could happen in case an untrusted or trusted user with limited privileges were able to control the file path that would be used to read files on the machine via Jackson. If an attacker was able to create a symbolic link pointing to any location on the machine, arbitrary file reads would be possible.

Issue: Chance of misusing Jackson which can lead to reading files potentially not in scope of the attacker.

Recommendations:
1. By default disallow following symlinks when reading files and require users to enable it if desired.

ADA-Jackson-6: ObjectMapper.writeValue() allows arbitrary file writes

Severity: Informational	Difficulty: Informational
Fixed: No	Vector: CWE-787: Out-of-bounds Write

Description
An untrusted user who controls the `File` argument passed to `ObjectMapper.writeValue()` can arbitrarily write files to the file system. This has a number of exploitation vectors:

1. Overwriting password files and escalating privileges.
2. Overwriting arbitrary files to make services on the machine unavailable.
3. Writing lots of files to the filesystem and exhausting disk space.

Recommendations
- Allow Jackson users to manage the destination to which files are written. This could be done through whitelisting certain paths on the filesystem.
- Sanitize filepaths by default and allow users to disable this by choice. Sanitization should:
    - Disallow absolute paths
    - Disallow “../” sequences.
- Document the security issues of allowing arbitrary untrusted input in the path to `ObjectMapper.writeValue()`.

ADA-Jackson-7: Arbitrary file lookup when parsing map key of type URI with StdKeyDeserializer

Severity: Informational	Difficulty: Informational
Fixed: No	Vector: CWE-125: Out-of-bounds Read

CWEs

- CWE-23: Relative Path Traversal
- CWE-36: Absolute Path Traversal
- CWE-73: External Control of Filename or Path

Description

StdKeyDeserializer._parse(String key, DeserializationContext ctxt) creates a URI for a file pathprovided by untrusted input:
https://github.com/FasterXML/jackson-databind/blob/2.14/src/main/java/com/fasterxml/jackson/databind/deser/std/StdKeyDeserializer.java
A malicious user could make Jackson-databind perform the following action:

```
case TYPE_URI:
        try {
                return URI.create("file://sensitive-file-contents.txt");
        } catch (Exception e) {
                return _weirdKey(ctxt, key, e);
        }
```

The lookup happens when a MappingIterator is created with URI as the key type:

```
ObjectMapper mapper = new ObjectMapper();
ObjectReader r = mapper.readerFor(String.class);
try {
        MappingIterator<Map<URI, String>> iterator = mapper
                .reader()
                .forType(new TypeReference<Map<URI, String>>(){})
                .readValues("{\"/etc/pwd2\":\"value\"}");
while (iterator.hasNext()) {
        iterator.next(); // <- The URI creation happens here
}
```

Mitigation

Users should mitigate this by not performing file reads from the returned URI.

ADA-Jackson-8: Regex Denial of Service (ReDos) in FromStringDeserializer

Severity: Informational	Difficulty: Informational
Fixed: No	Vector: CWE-1333: Inefficient Regular Expression Complexity
URLs Monorail issue: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=51219	

Description

The `FromStringDeserializer` passes untrusted input to java.util.regex.Pattern.compile() which could allow a malicious untrusted user to launch a Denial of Service (DoS) attack against Jackson users by providing a regex that is particularly expensive to evaluate.
This is not a Jackson issue as such but rather a consequence of regular expressions and their implementations. We list this issue in this report because one of the fuzzers reported it and it may be easy to fall into a code pattern that exposes the DoS vulnerability.

The vulnerable line:
https://github.com/FasterXML/jackson-databind/blob/2.14/src/main/java/com/fasterxml/jackson/databind/deser/std/FromStringDeserializer.java#L335

```
        @Override
        protected Object _deserialize(String value, DeserializationContext ctxt)
throws IOException
        {
            switch (_kind) {
            ...
            case STD_PATTERN:
                // will throw IAE (or its subclass) if malformed
                return Pattern.compile(value);
```

Example code pattern

```
ObjectMapper mapper = new ObjectMapper();
ObjectReader r = mapper.readerFor(Pattern.class);
Reader stringR = new StringReader(new String(UNTRUSTED INPUT));
Pattern result = r.readValue(stringR);
```

If result was to be matched against in the users application, a well-formed `Pattern` could cause Denial of Service of Jackson-databind:

```
Matcher m = result.matcher(newString);
```

Mitigation

It is a non-trivial task to identify Regexes that can cause this sort of behaviour, however, there are tools available to detect such regexes. We recommend Jackson users avoid a pattern above unless sure what Regex is used.

ADA-Jackson-9: DoS from stack exhaustion in Jackson-databind

Severity: High	Difficulty: Medium
Fixed: **Yes**	Vector: CWE-674: Uncontrolled Recursion
URLs Monorail issue: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=50490	
CVE: CVE-2022-42003 (7.5 HIGH)	

Description
A stack overflow vulnerability was found in Jackson-databinds `BeanDeserializer` which could cause Jackson-databind and the running application to crash from stack exhaustion.

Only users who had enabled the `DeserializationFeature.UNWRAP_SINGLE_VALUE_ARRAYS` feature would be exposed to the vulnerability.

The issue has been fixed in version 2.13.4 which was released on 3rd September 2022.

Monorail issue:
https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=50490&q=jackson&can=1

Fix URL:
https://github.com/FasterXML/jackson-databind/commit/063183589218fec19a9293ed2f17ec53ea80ba88

ADA-Jackson-10: DoS from stack exhaustion in Jackson-databind

Severity: High	Difficulty: Medium
Fixed: **Yes**	Vector: CWE-674: Uncontrolled Recursion
URLs Monorail issue: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=51219	
CVE: 2022-42004 (7.5 HIGH)	

Description
A stack overflow vulnerability was found in Jackson-databinds primitive value deserializers which could cause Jackson-databind and the running application to crash from stack exhaustion.

Only users who had enabled the `DeserializationFeature.UNWRAP_SINGLE_VALUE_ARRAYS` feature would be exposed to the vulnerability.

The issue has been fixed in version 2.14.0-rc1 which was released on 25th September 2022.

Fix URL:
https://github.com/FasterXML/jackson-databind/commit/d78d00ee7b5245b93103fef3187f70543d67ca33

ADA-Jackson-11: DoS from stack exhaustion in Jackson-databind

Severity: High	Difficulty: High
Fixed: **Yes**	Vector: CWE-674: Uncontrolled Recursion
URLs Monorail issue: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=50087	
CVE: 2022-42004 (7.5 HIGH)	

Description
A stack overflow was found in Jackson-databind by a fuzzer and was reported by OSS-Fuzz. The issue would occur when calling `JsonPointer.compile(...)` on a very deeply nested expression.

The issue is potentially exploitable for rare use cases where JsonPointers are deserialized. Because of the rarity of such use cases, a CVE has not been assigned.

ADA-Jackson-12: DoS from stack exhaustion in Jackson-databind

Severity: High	Difficulty: High
Fixed: **No**	Vector: CWE-674: Uncontrolled Recursion
URLs Monorail issue: https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=49828	
CVE: 2022-42004 (7.5 HIGH)	

Description
A stack overflow was found in Jackson-databind by a fuzzer and reported by OSS-Fuzz.

The issue would happen when Jackson-databind would read deeply nested content, and that content would be written out without changes in the following way:

```
ObjectMapper mapper = new ObjectMapper();
try {
    JsonNode root = mapper.readTree("MALICIOUS_DEEPLY_NESTED_STRING");
    String json = mapper.writeValueAsString(root); <------- Crash would
happen here
}
```