Source Code Audit on simplejson
for Open Source Technology Improvement Fund (OSTIF)

Final Report and Management Summary

2023-04-18

PUBLIC

X41 D-SEC GmbH
Krefelderstr. 123
D-52070 Aachen
Amtsgericht Aachen: HRB19989

https://x41-dsec.de/
info@x41-dsec.de

Organized by the Open Source Technology Improvement Fund

Revision	*Date*	*Change*	*Author(s)*
1	2023-03-29	Final Report and Management Summary	E. Sesterhenn, J. M., L. Gommans
2	2023-04-13	Public Release	L. Gommans
3	2023-04-18	Minor clarifications	L. Gommans

Contents

1 Executive Summary **4**

2 Introduction **6**
2.1 Methodology 6
2.2 Findings Overview 8
2.3 Scope 8
2.4 Coverage 8
2.5 Recommended Further Tests 10

3 Rating Methodology for Security Vulnerabilities **11**
3.1 Common Weakness Enumeration 12

4 Results **13**
4.1 Findings 13
4.2 Informational Notes 20

5 About X41 D-Sec GmbH **30**

A Fuzzing **32**
A.1 Differential Fuzzing - python-afl 32
A.2 Differential Fuzzing - pythonfuzz 34
A.3 Differential Fuzzing - atheris 35
A.4 Differential Fuzzing - pythonfuzz / orjson 36

Dashboard

Target	
Customer	Open Source Technology Improvement Fund (OSTIF)
Name	simplejson
Type	Python Library
Version	Version 3.18.4
Engagement	
Type	Source Code Audit
Consultants	3: Eric Sesterhenn, J. M., and Luc Gommans
Engagement Effort	12 person-days, 2023-03-19 to 2023-03-24
Total issues found	3

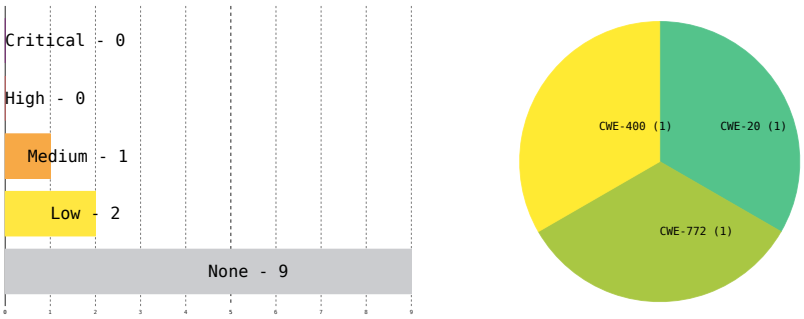

Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

1 Executive Summary

In March 2023, X41 D-Sec GmbH performed a source code audit against simplejson to identify vulnerabilities and weaknesses in the library.

A total of three vulnerabilities were discovered during the test by X41. None were rated as having a critical or high severity, one as medium, and two as low. Additionally, nine issues without a direct security impact were identified.

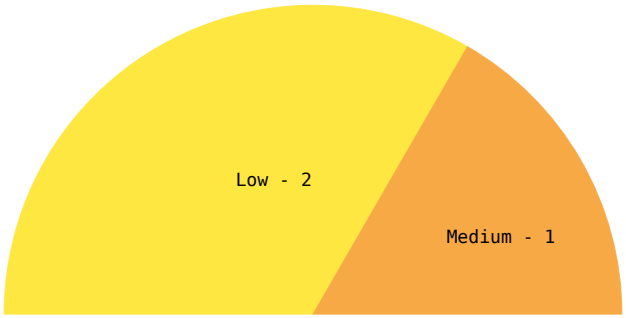

Figure 1.1: Issues and Severity

simplejson is a library to parse JSON data in Python. It has a similar interface to the built-in JSON parser but supports older versions of Python including 2.5. Alongside a pure Python implementation, it also implements parts of the code in C for performance reasons. Vulnerabilities in the library could allow an attacker to compromise software which uses JSON parsing for untrusted input, as is not uncommon for data interfaces.

In a source code audit, all information about the system is available. The test was performed by three experienced security experts between 2023-03-19 and 2023-03-24.

The most severe issue discovered relates to a security fix for newer Python versions not being implemented in simplejson. This allows an attacker to cause a denial of service by specifying an uncommonly large number in the JSON data. Besides taking down networked services, one scenario could be blocking a JSON-based updater to keep users on an old version of software for which a patch was recently released. Because the software hangs instead of producing a clear error message, the malicious nature of the situation can go unnoticed.

In addition, the Python implementation parses character formats which are not in compliance with specification. This is due to the use of a general-purpose integer conversion function without validating the input beforehand. An attacker might be able to exploit a situation where the validator uses simplejson and passes all checks, but then crashes a back-end system due to the invalid JSON data.

Among the informational notes, hardening could be applied by signing the code changes with a developer's private key. X41 had no insight into the GitHub account or organization settings and thus recommends to review these in a separate audit. X41 furthermore recommends to apply additional restrictions to untrusted input data.

Overall, the lack of high-severity issues being identified attests to the maturity of this project and code base, but further hardening can be applied on a technical and general organizational level.

2 Introduction

X41 reviewed the source code of simplejson which lets users serialize and deserialize JSON[1] objects.

Simplejson is considered sensitive because often JSON data is considered to be untrusted input. The library is used by various projects and any security issues will therefore affect a wide range of systems. According to pypistats.org[2], the daily number of daily downloads for the project is over half a million, of which about 51000 are by systems with an old Python version.

Attackers could try to attack simplejson by using specially crafted JSON input in order to create unexpected deserialized objects, which might then be used to attack software relying on simplejson. Likewise, attackers could try to use specially crafted Python object attributes in order to create unexpected serialized JSON, hoping to abuse software that relies on the serialization. Furthermore, attackers could also try to cause failures in simplejson, causing unexpected errors or rendering the software relying on simplejson unavailable. Additionally, attackers might try to trigger memory safety violations in the C parts that allow for information leaks or memory corruptions.

2.1 Methodology

The review was based on a source code review of the Python and C source code.

A manual approach for penetration testing and for code review is used by X41. This process is supported by tools such as static code analyzers and industry standard web application security tools[3].

X41 adheres to established standards for source code reviewing and penetration testing. These

[1] JavaScript Object Notation
[2] `https://pypistats.org/packages/simplejson`
[3] `https://portswigger.net/burp`

are in particular the *CERT Secure Coding*[4] standards and the *Study - A Penetration Testing Model*[5] of the German Federal Office for Information Security.

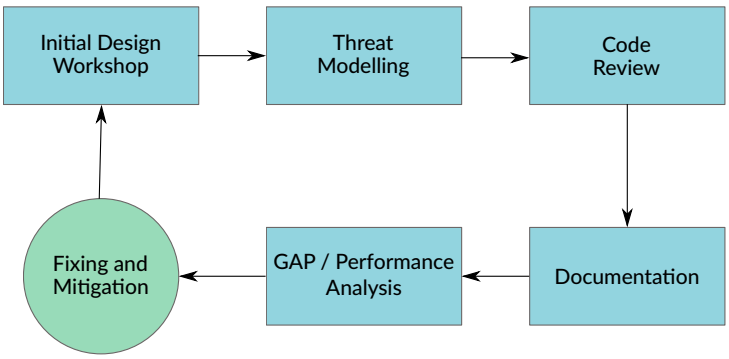

Figure 2.1: Code Review Methodology

[4] https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards
[5] https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1

2.2 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
Invalid Handling of Broken Unicode Escape Sequences	LOW	SJ-PT-23-01	4.1.1
Missing Reference Count Decrease	LOW	SJ-PT-23-02	4.1.2
Quadratic Number Parsing	MEDIUM	SJ-PT-23-03	4.1.3
Broken Error Display	NONE	SJ-PT-23-100	4.2.1
Unused Import	NONE	SJ-PT-23-101	4.2.2
Unused Tuple-related Function Arguments	NONE	SJ-PT-23-102	4.2.3
Unused Function Argument _one_shot	NONE	SJ-PT-23-103	4.2.4
Type Hints Not Used	NONE	SJ-PT-23-104	4.2.5
Deprecated Python Versions Supported	NONE	SJ-PT-23-105	4.2.6
Unsigned Git Commits	NONE	SJ-PT-23-106	4.2.7
Infinity and NaN	NONE	SJ-PT-23-107	4.2.8
Support of Duplicate Key Names	NONE	SJ-PT-23-108	4.2.9

Table 2.1: Security-Relevant Findings

2.3 Scope

The audit was performed against the most recent simplejson version, 3.18.4[6]. The code contained around 3000 lines of Python and 3000 lines of C code, including tests.

2.4 Coverage

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being found in the future.

The time allocated to X41 for this assessment was sufficient to yield a good coverage of the given scope.

Besides a manual audit for out-of-bound memory accesses and logic issues the C code was inspected using Cppcheck[7], Semgrep[8] and analyzed by GCC[9], LLVM[10] and cpychecker[11] analyzers.

[6] https://github.com/simplejson/simplejson/releases/tag/v3.18.4
[7] https://cppcheck.sourceforge.io/
[8] https://semgrep.dev/
[9] https://gcc.gnu.org/
[10] https://llvm.org/
[11] https://gcc-python-plugin.readthedocs.io/en/latest/cpychecker.html

Several differential fuzz harnesses were implemented to be able to identify parsing discrepancies between the C and Python implementations (simplejson and orjson[12]) as well as memory corruption issues in the C parts. The fuzz testing process is explained in detail in Appendix A.

Besides manual auditing for issues in the Python code, it has also been analyzed using the following tools:

- Bandit[13]
- Pyre[14]
- Flake8[15], using the plugins
  - dlint
  - flake8-bugbear
  - flake8-string-format
  - flake8-unused-arguments
  - flake8_encodings
  - flake8_secure_coding_standard
  - flake8_warnings
  - hacking.core
  - mccabe
  - pycodestyle
  - pyflakes
  - warn-symbols

In addition, JSONTestSuite[16] was used to check for possible JSON parsing issues.

Suggestions for next steps in securing this scope can be found in section 2.5.

[12] https://github.com/ijl/orjson
[13] https://bandit.readthedocs.io/en/latest/
[14] https://pyre-check.org
[15] https://github.com/pycqa/flake8
[16] https://github.com/nst/JSONTestSuite

2.5 Recommended Further Tests

X41 recommends to mitigate the issues described in this report. For issues with a direct security impact, CVE[17] IDs[18] should be requested and customers be informed (e.g. via a changelog or a special note for issues with higher severity) to ensure that they can make an informed decision about upgrading or other possible mitigations.

Further tests could cover the security of the GitHub repository.

[17] Common Vulnerabilities and Exposures
[18] Identifiers

3 Rating Methodology for Security Vulnerabilities

Security vulnerabilities are given a purely technical rating by the testers as they are discovered during the test. Business factors and financial risks for Open Source Technology Improvement Fund (OSTIF) are beyond the scope of a penetration test which focuses entirely on technical factors. Yet technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

In total, five different ratings exist, which are as follows:

Severity Rating
None
Low
Medium
High
Critical

A low rating indicates that the vulnerability is either very hard for an attacker to exploit due to special circumstances, or that the impact of exploitation is limited, whereas findings with a medium rating are more likely to be exploited or have a higher impact. High and critical ratings are assigned when the testers deem the prerequisites realistic or trivial and the impact significant or very significant.

Findings with the rating 'none' are called informational findings and are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

3.1 Common Weakness Enumeration

The CWE[1] is a set of software weaknesses that allows the categorization of vulnerabilities and weaknesses in software. If applicable, X41 provides the CWE-ID for each vulnerability that is discovered during a test.

CWE is a very powerful method to categorize a vulnerability and to give general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*[2]. More information can be found on the CWE website at `https://cwe.mitre.org/`.

[1] Common Weakness Enumeration
[2] `https://www.mitre.org`

4 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 4.1. Additionally, findings without a direct security impact are documented in Section 4.2.

4.1 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

4.1.1 SJ-PT-23-01: Invalid Handling of Broken Unicode Escape Sequences

Severity:	LOW
CWE:	20 – Improper Input Validation
Affected Component:	simplejson/decoder.py:py_scanstring()

4.1.1.1 Description

The ***py_scanstring()*** function is used by the Python code to parse escaped strings. Among these are `\u` strings, that are followed by four ASCII[1] characters that describe a hexadecimal value. When one of the characters is not a valid hexadecimal character (0-9, A-F) errors occur.

```
1 00000000  22 5c 75 20 45 44 44 22                        |"\u EDD"|
```

Listing 4.1: Invalid Unicode Escape Sequence

[1] American Standard Code for Information Interchange

The space character (*0x20*) is invalid in this context, and the C code raises an exception when parsing this.

```
Traceback (most recent call last):
  File "/home/eric/code/fuzzing/xx.py", line 57, in <module>
    fuzz()
  File "/home/eric/code/fuzzing/xx.py", line 28, in fuzz
    r2 = json2.loads(s)
         ^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.11/dist-packages/simplejson2-3.18.4-py3.11-linux-x86_64.egg/simple
  ↪ json2/__init__.py", line 525, in
  ↪ loads
    return _default_decoder.decode(s)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.11/dist-packages/simplejson2-3.18.4-py3.11-linux-x86_64.egg/simple
  ↪ json2/decoder.py", line 372, in
  ↪ decode
    obj, end = self.raw_decode(s)
               ^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.11/dist-packages/simplejson2-3.18.4-py3.11-linux-x86_64.egg/simple
  ↪ json2/decoder.py", line 402, in
  ↪ raw_decode
    return self.scan_once(s, idx=_w(s, idx).end())
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
simplejson2.errors.JSONDecodeError: Invalid \uXXXX escape sequence: line 1 column 3 (char 2)
```

Listing 4.2: Exception in C Code

The Python code serializes this into the value *0edd*. This happens because the Python code uses ***int()*** to parse this value.

```
# Unicode escape sequence
msg = "Invalid \\uXXXX escape sequence"
esc = s[end + 1:end + 5]
escX = esc[1:2]
if len(esc) != 4 or escX == 'x' or escX == 'X':
    raise JSONDecodeError(msg, s, end - 1)
try:
    uni = int(esc, 16)
except ValueError:
    raise JSONDecodeError(msg, s, end - 1)
```

Listing 4.3: Python Parsing for Unicode Escape Sequences

The function ***int()*** ignores surrounding whitespace and underscores between digits[2].

> If *x* is not a number or if *base* is given, then *x* must be a string, bytes, or bytearray instance representing an integer in radix *base*. Optionally, the string can be preceded by `+` or `-` (with no space in between), have leading zeros, be surrounded by whitespace, and have single underscores interspersed between digits.

This can cause erroneous JSON to be parsed as valid and other implementations might either reject these inputs or even serialize it to other values. This might lead to confusions about the actual contents.

4.1.1.2 Solution Advice

X41 recommends to verify the digits in Unicode escape sequences in the Python code as well and raise an exception if those are not valid.

[2] `https://docs.python.org/3/library/functions.html#int`

4.1.2 SJ-PT-23-02: Missing Reference Count Decrease

Severity:	LOW
CWE:	772 – Missing Release of Resource after Effective Lifetime
Affected Component:	/simplejson/_speedups.c:_match_number_str()

4.1.2.1 Description

In `_speedups.c`, the function ***_match_number_str()*** creates a variable `numstr` using ***PyString_FromStringAndSize()***[3] and might be a shared object. This requires ***Py_DECREF()***[4] to be called on the object once it is no longer in use. This is done except when ***PyOS_string_to_double()*** fails.

```
1  /* copy the section we determined to be a number */
2  numstr = PyString_FromStringAndSize(&str[start], idx - start);
3  if (numstr == NULL)
4      return NULL;
5  if (is_float) {
6      /* parse as a float using a fast path if available, otherwise call user defined method */
7      if (s->parse_float != (PyObject *)&PyFloat_Type) {
8          rval = PyObject_CallOneArg(s->parse_float, numstr);
9      }
10     else {
11         /* rval = PyFloat_FromDouble(PyOS_ascii_atof(PyString_AS_STRING(numstr))); */
12         double d = PyOS_string_to_double(PyString_AS_STRING(numstr),
13                                          NULL, NULL);
14         if (d == -1.0 && PyErr_Occurred())
15             return NULL;
16         rval = PyFloat_FromDouble(d);
17     }
18 }
19 else {
20     /* parse as an int using a fast path if available, otherwise call user defined method */
21     if (s->parse_int != (PyObject *)&PyInt_Type) {
22         rval = PyObject_CallOneArg(s->parse_int, numstr);
23     }
24     else {
25         rval = PyInt_FromString(PyString_AS_STRING(numstr), NULL, 10);
26     }
27 }
28 Py_DECREF(numstr);
```

Listing 4.4: Missing Py_DECREF()

[3] https://docs.python.org/2/c-api/string.html#c.PyString_FromStringAndSize
[4] https://docs.python.org/3/c-api/refcounting.html#c.Py_DECREF

This causes a reference being held to the *numvar* string which is therefore not freed by the garbage collector. This can cause a memory leak.

The code affected is only active for Python versions lower than 3.

4.1.2.2 Solution Advice

X41 recommends to add the missing call to ***Py_DECREF()*** into the error handling.

4.1.3 SJ-PT-23-03: Quadratic Number Parsing

Severity:	MEDIUM
CWE:	400 – Uncontrolled Resource Consumption ('Resource Exhaustion')
Affected Component:	simplejson/decoder.py:356

4.1.3.1 Description

Parsing numbers in JSON strings takes a quadratic amount of time, that is, the time taken quadruples when the input size doubles[5]. According[6] to the documentation, "There exists no algorithm that can convert a string to a binary integer or a binary integer to a string in linear time, *unless* the base is a power of 2." JSON does not support notations for bases other than ten, aside from characters in strings where no more than four hexadecimals can be specified.

Number Size	Time Taken
1 MiB	3.6 seconds
2 MiB	14.3 seconds
3 MiB	33.4 seconds
4 MiB	59.4 seconds

Table 4.1: Number Size vs. Parsing Time on Debian Stable

Because JSON data is very often untrusted input, this could allow an attacker to hang a process with a few megabytes of data. When done with enough parallelism, this often makes the service deny further (legitimate) clients.

This bug is classified as CVE-2020-10735[7]. The latest CPython versions limit the input to ***int()*** to 4300 digits by default, but also have faster number parsing: 3.4 seconds for a 4 MiB number in CPython 3.12.0a6+ (commit `87be8d9522`) on the same setup as used in table 4.1.

4.1.3.2 Solution Advice

For security fixes which are not backported to all Python versions which simplejson supports, X41 recommends to implement the security fixes in simplejson itself. In this case, simplejson could check for the existence of the new method[8] and implement its own limit if this is not available. An example is shown in listing 4.5.

[5] X41 previously parallel-discovered and reported this issue as TUF-CR-22-03 in a different scope: https://www.x41-dsec.de/static/reports/X41-TUF-Audit-2022-Final-Report-PUBLIC.pdf
[6] https://docs.python.org/3/library/stdtypes.html#integer-string-conversion-length-limitation
[7] https://nvd.nist.gov/vuln/detail/CVE-2020-10735
[8] https://docs.python.org/3/library/stdtypes.html#recommended-configuration

```
import sys

# Module-global constant for brevity; configurability may be desired
INT_MAX_STR_DIGITS = 4300

def bounded_int(x, base=10):
    if (not hasattr(sys, "set_int_max_str_digits") and base & (base-1) == 0
            and hasattr(x, '__len__') and len(x) > INT_MAX_STR_DIGITS):
        raise ValueError("[...]" % (INT_MAX_STR_DIGITS, len(x)))

    return int(x, base)
```

Listing 4.5: Limiting int() Input

4.2 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

4.2.1 SJ-PT-23-100: Broken Error Display

Affected Component: _speedups.c

4.2.1.1 Description

When an exception is thrown by the C parsing code, the C code in some cases reports the errors with an offset of one greater than the Python counterpart.

```
00000000  2d 2d                                             |--|
```

Listing 4.6: Example File

This results in two different errors, where the Python code reports the error at char *0* and C at char *1*.

```
# Different errors:
e: Expecting value: line 1 column 1 (char 0)
e2: Expecting value: line 1 column 2 (char 1)
```

Listing 4.7: Off-By-One in C Error Display

The correct behavior seems to be the one of the Python implementation, which reports the start of the non-matched string.

```
def _scan_once(string, idx):
    errmsg = 'Expecting value'
...
```

```
    m = match_number(string, idx)
    if m is not None:
...
    else:
        raise JSONDecodeError(errmsg, string, idx)
```

Listing 4.8: Expecting Value Error

In some cases, this might even lead to different errors being reported when a file is parsed.

```
00000000  7b                                                |{|

# Different errors:
e: Expecting property name enclosed in double quotes: line 1 column 2 (char 1)
e2: Expecting property name enclosed in double quotes or '}': line 1 column 2 (char 1)
```

Listing 4.9: Different Reported Errors

In other cases, the Python implementation is one character ahead of where the error is actually happening.

```
00000000  22 18 64                                          |".d|

# Different errors:
e: Invalid control character 'd' at: line 1 column 3 (char 2)
e2: Invalid control character '\x18' at: line 1 column 2 (char 1)
```

Listing 4.10: Incorrect Character Reported in Error

4.2.1.2 Solution Advice

X41 recommends to unify the error reporting to achieve the same behavior from the C and Python implementation.

4.2.2 SJ-PT-23-101: Unused Import

Affected Component:	encoder.py

4.2.2.1 Description

unichr is imported from `compat.py`, but not used.

4.2.2.2 Solution Advice

X41 recommends to remove unused imports.

4.2.3 SJ-PT-23-102: Unused Tuple-related Function Arguments

Affected Component: __init__.py

4.2.3.1 Description

The ***simplejson.load*** function signature defines the `namedtuple_as_object=True` and `tuple_as_array=True` arguments, but they are not used.

4.2.3.2 Solution Advice

X41 recommends to either use or remove the unused arguments.

4.2.4 SJ-PT-23-103: Unused Function Argument _one_shot

Affected Component: decoder.py

4.2.4.1 Description

The ***simplejson.encoder._make_iterencode*** function signature defines the `_one_shot` argument, but it is not used.

4.2.4.2 Solution Advice

X41 recommends to either use or remove the unused argument.

4.2.5 SJ-PT-23-104: Type Hints Not Used

Affected Component:	Python code

4.2.5.1 Description

Python Type Hints (PEP 484[9]) are not used in the Python code. They could aid in detecting possible mistakes through the use of static analyzers, among others.

4.2.5.2 Solution Advice

X41 recommends to make use of Type Hints.

[9] `https://peps.python.org/pep-0484`

4.2.6 SJ-PT-23-105: Deprecated Python Versions Supported

Affected Component: Python code

4.2.6.1 Description

simplejson supports Python version 2.5 and above. Because there are many differences between Python 2 and Python 3, this requires many version checks and code branches, adding avoidable complexity.

Python 2.7 has been deprecated for many years and reached EOL[10] in 2020[11]. The also-supported 2.5 version was last updated[12] in 2006 and does not appear to ever have had official security support[13].

4.2.6.2 Solution Advice

X41 recommends to plan and announce the removal of Python 2 support in simplejson.

[10] End Of Life
[11] `https://www.python.org/doc/sunset-python-2/`
[12] `https://peps.python.org/pep-0356/#release-schedule`
[13] `https://devguide.python.org/versions/`

4.2.7 SJ-PT-23-106: Unsigned Git Commits

Affected Component:	https://github.com/simplejson/simplejson

4.2.7.1 Description

Commits and tags in the git repository are currently not signed, providing no certainty about the authenticity of a commit.

In addition, any tag pushed to the repository is automatically published to the Python Package Index (PyPI).

A breach of a maintainer account or of GitHub, or a vulnerability in the git protocol, might allow an attacker to create unauthorized commits or tags in the repository.

4.2.7.2 Solution Advice

X41 recommends to make use of commit and tag signatures, to document the used signing keys, and to add the signing keys to the relevant GitHub account[14]. In addition, X41 recommends to require signed commits on the repository[15], and to verify the commit and tag signatures in GitHub Actions before publishing on PyPI.

[14] https://docs.github.com/en/authentication/managing-commit-signature-verification/adding-a-gpg-key-to-your-github-account

[15] https://docs.github.com/en/repositories/configuring-branches-and-merges-in-your-repository/defining-the-mergeability-of-pull-requests/about-protected-branches#require-signed-commits

4.2.8 SJ-PT-23-107: Infinity and NaN

Affected Component:	Encoder and Decoder

4.2.8.1 Description

simplejson supports the serialization and deserialization of `Infinity`, `-Infinity`, and `NaN`. The JSON specifications ECMA-404 (2nd Edition / December 2017)[16] and RFC[17] 9529[18] both state that numeric values that cannot be represented as sequences of digits "(such as `Infinity` and `NaN`) are not permitted". While this can be disabled in simplejson using the `allow_nan` and `ignore_nan` options, the default is not standards-compliant, which may be unexpected by users of the software.

4.2.8.2 Solution Advice

X41 recommends to make the default standards-compliant.

[16] https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf#page=12
[17] Request for Comments
[18] https://www.rfc-editor.org/rfc/rfc8259#section-6

4.2.9 SJ-PT-23-108: Support of Duplicate Key Names

Affected Component: Decoder

4.2.9.1 Description

RFC 9529[19] states that "names within an object SHOULD be unique" and further clarifies it means that "all software implementations receiving that object will agree on the name-value mappings". Given that Python dictionaries require unique keys and that duplicate names in JSON are almost always the result of a mistake or an attempt of leveraging implementation differences for attacks, duplicate names should not happen under normal circumstances. The data is likely to be either erroneous or malicious.

4.2.9.2 Solution Advice

X41 recommends to make it the default behavior to raise an exception when duplicate names are encountered.

[19] https://www.rfc-editor.org/rfc/rfc8259#section-4

5 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of the Git source code version control system[1]
- Review of the Mozilla Firefox updater[2]
- X41 Browser Security White Paper[3]
- Review of Cryptographic Protocols (Wire)[4]
- Identification of flaws in Fax Machines[5,6]
- Smartcard Stack Fuzzing[7]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

[1] `https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[2] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[3] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[4] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[5] `https://www.x41-dsec.de/lab/blog/fax/`
[6] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[7] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

ASCII American Standard Code for Information Interchange 13
CVE Common Vulnerabilities and Exposures 10
CWE Common Weakness Enumeration 12
EOL End Of Life 26
ID Identifier 10
JSON JavaScript Object Notation 6
RFC Request for Comments 28

A Fuzzing

This appendix describes the various fuzz tests performed by X41. The fuzz tests were performed against commit 463416cc9f9f7d177f963c31ac33ec8acf199e6e[1].

Fuzz testing[2] is a method for automated software testing. It is used to test for security vulnerabilities and other implementation errors. Classically a *fuzzer* or *fuzzing harness* is a program that generates input for other software with the hopes of triggering bugs. While classically used as a black box analysis method, fuzzing is nowadays often used with instrumentation and coverage analysis techniques, either by compiling instrumented binaries or by using dynamic binary instrumentation techniques.

With all fuzzing approaches the goals for a successful fuzzing operation are:

- Speed: A high number of iterations per second
- Accuracy: a high number of relevant inputs that provide a high coverage of the target code
- High Signal-to-Noise Ratio: Fuzzing results should be valid bugs

The fuzz corpus was seeded by the files from the go-fuzz[3] project and the JSONTestSuite[4].

The code is already being fuzz-tested[5] with a straight-forward fuzzer, but no differential tests seem to be performed.

A.1 Differential Fuzzing - python-afl

A fuzzer was developed to be able to identify discrepancies between the C and Python implementation as well as memory violations in the C implementation.

Differential fuzzing can be described using the following steps:

1. Generate input
2. Run implementation A and B individually on the input generated in step 1.
3. Compare the results
4. Crash/Abort if the results differ

[1] https://github.com/simplejson/simplejson/tree/463416cc9f9f7d177f963c31ac33ec8acf199e6e
[2] https://owasp.org/www-community/Fuzzing
[3] https://github.com/dvyukov/go-fuzz-corpus
[4] https://github.com/nst/JSONTestSuite
[5] https://github.com/simplejson/simplejson/pull/274

The fuzzer relies on simplejson being installed twice (once as simplejson, once as simplejson2), with on version using plain Python and the other using the C speedups. The fuzz testing is performed using python-afl[6].

```
# this expects simplejson to be installed twice, once with c
# and once without c speedups. We fuzz with the aim to see parsing
# discrepancies between the two implementations

import simplejson as json
import simplejson2 as json2
import afl
import sys
import os

def fuzz():
    # setup and read data
    e = ""
    e2 = ""
    r1 = None
    r2 = None
    sys.stdin.seek(0)
    try:
        s = sys.stdin.read()
    except UnicodeDecodeError:
        sys.exit(1)

    if not s:
        sys.exit(1)

    # parse with c and non-c version
    try:
        r1 = json.loads(s)
    except Exception as err:
        e = str(err)
        pass

    try:
        r2 = json2.loads(s)
    except Exception as err:
        e2 = str(err)
        pass

    # Check whether both implementations raised different errors
    if e != e2:
        print("# Different errors:")
        print("e: " + e)
        print("e2: " + e2)
        print("r: " + json.dumps(r1))
        print("r2: " + json.dumps(r2))
        raise Exception("Different errors")
```

[6] https://github.com/jwilk/python-afl

```
47      else:
48          # check if input got serialized into different objects
49          if json.dumps(r1) != json.dumps(r2):
50              print("# Different objects:")
51              print("r: " + json.dumps(r1))
52              print("r2: " + json.dumps(r2))
53              raise Exception("Different objects")
54
55  def main():
56
57      while afl.loop(1000):
58          fuzz()
59
60  if __name__ == '__main__':
61      json.loads("{}")
62      json2.loads("{}")
63      while afl.loop():
64          main()
65
66      fuzz()
```

Listing A.1: Python-afl Harness

A.2 Differential Fuzzing - pythonfuzz

For additional coverage and different mutations, the fuzzer for python-afl was converted to pythonfuzz[7]. For this fuzz test, the code was modified to ignore different errors and only raise an exception when an input is serialized into different objects. This was done to identify strings that might be handled differently by boths implementations and do not generate errors.

```
1   # this expects simplejson to be installed twice, once with c
2   # and once without c speedups. We fuzz with the aim to see parsing
3   # discrepancies between the two implementations
4
5   import simplejson as json
6   import simplejson2 as json2
7   from pythonfuzz.main import PythonFuzz
8
9   @PythonFuzz
10  def fuzz(buf):
11      # setup and read data
12      r1 = None
13      r2 = None
14
15      try:
16          s = buf.decode("ascii")
```

[7] https://gitlab.com/gitlab-org/security-products/analyzers/fuzzers/pythonfuzz/

```
    except UnicodeDecodeError:
            return

    # parse with c and non-c version
    try:
        r1 = json.loads(s)
    except Exception as err:
        return

    try:
        r2 = json2.loads(s)
    except Exception as err:
        return

    # check if input got serialized into different objects
    if json.dumps(r1) != json.dumps(r2):
        print("# Different objects:")
        print("r: " + json.dumps(r1))
        print("r2: " + json.dumps(r2))
        raise Exception("Different objects")


if __name__ == '__main__':
    json.loads("{}")
    json2.loads("{}")
    fuzz()
```

Listing A.2: Pythonfuzz Harness

A.3 Differential Fuzzing - atheris

For additional coverage and different mutations, the fuzzer for python-afl was converted to atheris[8] as well which provided the most executions per second.

```
# this expects simplejson to be installed twice, once with c
# and once without c speedups. We fuzz with the aim to see parsing
# discrepancies between the two implementations

import atheris

with atheris.instrument_imports():
    import simplejson as json
    import simplejson2 as json2
    import sys
```

[8] https://pypi.org/project/atheris/

```
11
12 def TestOneInput(buf):
13     # setup and read data
14     r1 = None
15     r2 = None
16
17     try:
18         s = buf.decode("ascii")
19     except UnicodeDecodeError:
20             return
21
22     # parse with c and non-c version
23     try:
24         r1 = json.loads(s)
25     except (json.JSONDecodeError, RecursionError, ValueError) as err:
26         return
27
28     try:
29         r2 = json2.loads(s)
30     except (json2.JSONDecodeError, RecursionError, ValueError) as err:
31         return
32
33     # check if input got serialized into different objects
34     if json.dumps(r1) != json.dumps(r2):
35         print("# Different objects:")
36         print("r: " + json.dumps(r1))
37         print("r2: " + json.dumps(r2))
38         raise Exception("Different objects")
39
40 atheris.Setup(sys.argv, TestOneInput)
41 atheris.Fuzz()
```

Listing A.3: Atheris Harness

A.4 Differential Fuzzing - pythonfuzz / orjson

Using differential fuzzing to compare against other implementations such as orjson[9] were not straight forward, due to different indentation of the serialized JSON objects and the different integer precision. This was worked around by serializing and deserializing the data multiple times, which caused a loss of speed. Since a segfault was caused by the combination of atheris and orjson this fuzzer uses pythonfuzz as well.

```
1 import simplejson2 as json
2 import orjson as orjson
3 from pythonfuzz.main import PythonFuzz
4
```

[9] https://github.com/ijl/orjson

```
import sys
import os

@PythonFuzz
def fuzz(buf):
    # setup and read data
    r1 = None
    r3 = None

    try:
        s = buf.decode("ascii")
    except UnicodeDecodeError:
            return

    # parse with orjson and simplejson
    try:
        r1 = json.loads(s)
    except (json.JSONDecodeError, RecursionError, ValueError) as err:
        return

    try:
        r3 = orjson.loads(s)
    except (orjson.JSONDecodeError, RecursionError, ValueError) as err:
        return

    # check if input got serialized into different objects
    orj = orjson.dumps(r3)
    org = orjson.dumps(orjson.loads(json.dumps(r1)))
    if org != orj:
        print("# Different objects:")
        print("r : " + org)
        print("r3: " + orj)
        raise Exception("Different objects")



if __name__ == '__main__':
    json.loads("{}")
    orjson.loads("{}")
    fuzz()
```

Listing A.4: Pythonfuzz orjson Harness