ASSESSMENT REPORT

THE C PLUS PLUS ALLIANCE, INC.

BOOST.JSON SECURITY ASSESSMENT 2021

MARCH 17, 2021

This engagement was performed in accordance with the Statement of Work, and the procedures were limited to those described in that agreement. The findings and recommendations resulting from the assessment are provided in the attached report. Given the time-boxed scope of this assessment and its reliance on client-provided information, the findings in this report should not be taken as a comprehensive listing of all security issues.

This report is intended solely for the information and use of The C Plus Plus Alliance, Inc.

Bishop Fox Contact Information:
+1 (480) 621-8967
contact@bishopfox.com
8240 S. Kyrene Road
Suite A-113
Tempe, AZ 85284

TABLE OF CONTENTS

EXECUTIVE REPORT

Project Overview

The C Plus Plus Alliance, Inc. engaged Bishop Fox to assess the security of the Boost.JSON C++ library. The following report details the findings identified during the course of the engagement, which started on February 8, 2021.

Goals

- Ensure the Boost.JSON parser is secure against untrusted inputs and that undefined behavior and crashing are not possible
- Review the container, as in theory it could be the target of algorithmic complexity attacks
- Look for buffer overflows and other vulnerabilities in the Boost.JSON parser
- Thoroughly fuzz the Boost.JSON library
- Provide best practices for users of the Boost.JSON library

FINDING COUNTS

1 Low

1 Total finding

SCOPE

Boost.JSON library v.1.75.0

Commit:
edbf86641b7217676eceb6582ee486e31bb42745

DATES

02/08/2021
Kickoff

02/08/2021 – 02/23/2021
Active testing

03/17/2021
Report delivery

Approach

The assessment team conducted a hybrid application assessment of the Boost.JSON library. Bishop Fox's hybrid application assessment methodology leverages real-world attack techniques of application penetration testing in combination with targeted source code review. These full-knowledge assessments begin with automated scans of the deployed application and source code. Next, analyses of the scan results are combined with manual review to thoroughly identify potential application security vulnerabilities. In addition, the team conducts a review of the application architecture and business logic to locate any design-level issues. Finally, the team performs manual exploitation and review of these issues to validate the findings.

For this engagement, the team made use of the current fuzzing implementation, which the repository used with LLVM's AddressSanitizer (ASAN) and UndefinedBehaviorSanitizer (UBSAN) by running the fuzzing tests for extended periods of time. In conjunction with LLVM, the team made use of alternate fuzzers such as AFL++ with guidance from a symbolic execution framework, angr, to help extend coverage that was not provided by LLVM. The

use of these fuzzers in conjunction with manual review helped the team to explore the core functionality of the library and to identify memory management or corruption issues.

Strategic Approach

Fuzzing Strategy

One of the assessment goals was to identify vulnerabilities that can occur when Boost.JSON parses JSON input and serializes to JSON output. The assessment team found that the majority of Boost.JSON fuzzing programs and examples were set up in a similar fashion to tackle these ideas. This allowed the team to deploy the already-existing fuzzing harnesses into other fuzzing platforms for extended periods of time. The team modified the examples by adding a thread for executing AFL++ in persistent mode. Further modifications allowed harnesses to run in LibFuzz, AFL++, and AFL_IJON. All were guided with angr for further code coverage.

To fuzz the JSON parser, the team created a fuzzing harness based on the harnesses that already existed in the fuzzing folder within the Boost.JSON repository. This harness, based on the `fuzzer_parse`, `fuzzer_basic_parser`, and `fuzzer_parser` files in the fuzzing folder, was instrumented with the custom clang compiler `afl-clang-fast++` to assist with fuzzing the library with AFL. Parameters such as `AFL_PERSISTENT` were also introduced into the program to speed up fuzzing cycles.

The harnesses read in content from files generated by the fuzzer, then parse, stream parse, and validate the given JSON. These programs run until the number of identified paths stops growing and the number of completed cycles matches the paths discovered. The use of angr gave AFL the ability to find more "interesting" paths and avoided a "stuck" state, thereby allowing further code coverage within the library. The team determined that the fuzzing conducted was sufficient to produce reliable results in the time provided.

Static Source Code Analysis

The assessment team used Checkmarx against the Boost.JSON codebase for additional coverage and to supplement the fuzzing and manual code review process. Checkmarx flagged 219 issues. However, after review, the team determined that these were negligible and irrelevant to the goals of the engagement.

The team moved forward to check for known bad code practices, such as the use of unconstrained buffers and the use of unsafe execution of calls to the system. The library used innovative practices for dynamic memory management and properly deallocated memory created in the JSON objects. Since this library needed only basic functionality, it did not make any calls risky to the OS such as `execve` or system function.

A benefit of the Boost.JSON library, as of 1.75.0, is its limited list of dependencies to the Boost Library Collection. The only dependencies found were to `boost/system`, `boost/assert`, `boost/exception`, `boost/container`, and `boost/align`. This helped to

strengthen the integrity of Boost.JSON, since any potential future vulnerabilities found in addition to the Boost Library Collection would not affect the Boost.JSON library. If vulnerabilities are found in the general Boost Library Collection, it is recommended to ensure that `boost/system`, `boost/assert`, `boost/exception`, `boost/container`, and `boost/align` have not been affected.

Investigations of the current bugs and issues posted to the repository indicated that they were negligible to the integrity of the Boost.JSON library. The team built out small programs based on the bugs described and noted the success or failures of each. No failures were found, and the bugs were negligible to the operation and did not cause any undefined behavior. This suggests that the bugs either were patched or had occurred but could not be reproduced by the team. Any bugs that are already patched should be documented as such in the issues posted to the repository.

Dynamic Source Code Analysis

Aside from static code analysis, the team also took a dynamic approach to finding potential issues in the library. The team thoroughly reviewed the library to identify issues relating to type confusion, deserialization attacks, and improper deserialization/serialization of objects.

The assessment team found a common issue with JSON parsing that happens with most JSON parsers. Deserializing the following string resulted in a `type confusion` where the JSON parser within Boost.JSON interpreted `1E400` as an INF value instead of a semi-precise integer value. The following proof-of-concept code demonstrates an edge-case that users of the Boost.JSON library may run into:

```
#include <boost/json.hpp>
#include <iostream>

using namespace boost::json;

int main() {
        // Testing some type confusion issues
        std::string experiment = "{\"description\": \"Float (exp)\", \"test\": 1E400}";
        value test = parse(experiment);
        std::string real = serialize(test);
        std::cout << "[!] EXPECTED: " << experiment << std::endl;
        std::cout << "[-] Result:   " << real << std::endl;
}
```

FIGURE 1 - Demo code to show potential type representation issues

There is a potential for this use-case to cause issues in developer implementations. RFC 8259 section 6 specifies that the use of values such as INF, pi, or 1E400 may cause "interoperability problems," and the receiver may not be able to properly interpret the data sent by the sender. This would need to be verified by the user of the Boost.JSON library.

Security Considerations

The assessment team found that Boost.JSON was a mature and stable library. The security issues that are most likely to emerge with the library relate to several key areas that are outside of the scope that Boost.JSON can provide the developer.

The following sections address some of the more pertinent issues and provide examples of what developers should be aware of.

Data/Input Validation

The Boost.JSON library has two unique ways of handling user input. The first way is using the API to catch error codes, which should be done when the user input is coming from a trusted or untrusted source. The Boost.JSON error codes are defined in `error.hpp`, which inherits from `std::error_code`. These error codes are returned when a problem occurs in functions such as write or serialize, as shown in the pretty and validate example files. Error codes are lighter on resources than using exceptions, which is beneficial for programs that need to be as efficient as possible. To properly make use of the error codes, the user's program should read the error code value, which will be a 0 or a non-zero value. The user's program should then pass this error code to the `error::make_error_code()` function and print the results, which will inform the user what the error is.

The second way of handling user input is by utilizing exceptions. Exceptions should not be used when the user input is coming from an untrusted source as the exception will terminate the program. Users should throw an exception when the program can identity an external problem that prevents execution. The exception mechanism has a minimal performance cost if no exception is thrown. If an exception is thrown, the cost of the stack traversal and unwinding is roughly comparable to the cost of a function call. It is ultimately up to the user to handle data input validation.

The assessment team identified that it was possible for deserialization attacks to occur if an object was mishandled after being parsed whenever the input was trusted by the user. This implementation was specific to the developer using the library and by default, was not a security risk within the library itself. Developers should validate objects that are being parsed with the Boost.JSON library.

General Guidance

The validation and parsing of JSON can be implemented with the Boost.JSON library in a safe and secure fashion. As previously mentioned, developers should be aware of the limitations on the range of representable values for integer and floating-point types. This could cause implementation issues and undefined behavior for programs that use the JSON parser within their project.

Although the team did not find any crashes, the CI system for the library should fuzz for longer durations, potentially allowing for additional vulnerabilities to be identified during the fuzzing sessions. The current configuration in the Boost.JSON repository runs the built-

in Clang LibFuzz for 30 seconds. It is recommended to run for longer periods of time (greater than an hour if possible) to get the most benefit from fuzzing.

Summary of Findings

The assessment team identified one low-risk issue during the time-boxed assessment of Boost.JSON library version 1.75.0. The team identified a type representation vulnerability that may cause issues in implementations by users of the library. The team found that the Boost.JSON library was resilient to the memory attacks that can plague native libraries. Static analysis returned negligible informational results that did not warrant reporting.

The sections below detail the strategy used by Bishop Fox during the assessment, as well as other security considerations and recommendations for the production use of the Boost.JSON library.

RECOMMENDATIONS

- Users must decide how to handle errors returned by the parser upon receiving invalid JSON. The user must check for the errors, then handle them or allow the library to throw an exception and terminate the program.
- Users should be aware of the limitations with numeric representation for integers and floating-point values used by the library.

ASSESSMENT REPORT

Hybrid Application Assessment

The assessment team performed an application penetration test with the following target in scope:

- Boost.JSON v.1.75.0: `https://github.com/boostorg/json/commit/edbf86641b7217676eceb6582ee486e31bb42745`

Identified Issues

1 TYPE REPRESENTATION LOW

Definition

The representation of numbers is like that used in most programming languages. A number is represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed.

Details

The assessment team identified a common issue with JSON parsing that happens with most JSON parsers. When deserializing a user given input, a type-confusion can happen when the JSON parser within Boost.JSON interprets `1E400` as an INF value instead of a semi-precise integer value. The following proof-of-concept code demonstrates an edge-case that users of the Boost.JSON library may run into:

```
#include <boost/json.hpp>
#include <iostream>

using namespace boost::json;

int main() {
        std::string experiment =
            "{\"description\": \"Float (exp)\", \"test\": 1E400}";
        value test = parse(experiment);
        std::string real = serialize(test);
        std::cout << "[!] EXPECTED: " << experiment << std::endl;
        std::cout << "[-] Result:   " << real << std::endl;
}
```

FIGURE 2 – Demo code to show potential type representation issues

The output of running the above code is shown below:

```
[!] EXPECTED: {"description": "Float (exp)", "test": 1E400}
[-] Result:   {"description": "Float (exp)","test": Infinity}
```

FIGURE 3 – Demo code output

There is a potential for this use-case to cause issues in developer implementations. RFC 8259 section 6 specifies that the use of values such as INF, pi, or 1E400 may cause "interoperability problems," and the receiver may not be able to properly interpret the data sent by the sender. This would need to be verified by the implementor of the Boost.JSON library.

Affected Locations

Application
Boost.JSON v. 1.75.0

Total Instances 1

Recommendations

To mitigate the risk of type representation, the assessment team recommends the following actions:

- Integrate a type check between values fed into the JSON parser to verify there no value was interpreted differently.
- Make use of exceptions when an INF value is found within the JSON. Per RFC 8259 mentioned below, this should not exist within the JSON spec.

Additional Resources

The JavaScript Object Notation (JSON) Data Interchange Format
https://tools.ietf.org/html/rfc8259#section-6

RFC 8259
https://tools.ietf.org/html/rfc8259

APPENDIX A — MEASUREMENT SCALES

Finding Severity

Bishop Fox determines severity ratings using in-house expertise and industry-standard rating methodologies such as the Open Web Application Security Project (OWASP) and the Common Vulnerability Scoring System (CVSS).

The severity of each finding in this report was determined independently of the severity of other findings. Vulnerabilities assigned a higher severity have more significant technical and business impact and achieve that impact through fewer dependencies on other flaws.

Critical	Vulnerability is an otherwise high-severity issue with additional security implications that could lead to exceptional business impact. Findings are marked as critical severity to communicate an exigent need for immediate remediation. Examples include threats to human safety, permanent loss or compromise of business-critical data, and evidence of prior compromise.
High	Vulnerability introduces significant technical risk to the system that is not contingent on other issues being present to exploit. Examples include creating a breach in the confidentiality or integrity of sensitive business data, customer information, or administrative and user accounts.
Medium	Vulnerability does not in isolation lead directly to the exposure of sensitive business data. However, it can be leveraged in conjunction with another issue to expose business risk. Examples include insecurely storing user credentials, transmitting sensitive data unencrypted, and improper network segmentation.
Low	Vulnerability may result in limited risk or require the presence of multiple additional vulnerabilities to become exploitable. Examples include overly verbose error messages, insecure TLS configurations, and detailed banner information disclosure.
Informational	Finding does not have a direct security impact but represents an opportunity to add an additional layer of security, is a deviation from best practices, or is a security-relevant observation that may lead to exploitable vulnerabilities in the future. Examples include vulnerable yet unused source code and missing HTTP security headers.

APPENDIX B — TEST PLAN

The following section contains the test cases completed for each methodology in scope.

HAA METHODOLOGY

COMPLETED TEST CASES

TEST CASE	DESCRIPTION
Perform dynamic testing for memory management vulnerabilities	Attempt to identify user-supplied inputs that are unsafely loaded into memory or that unsafely reference existing memory.
Perform dynamic testing for object deserialization issues	Identify whether user-supplied inputs are used as serialized objects and sent to an unsafe deserialization routine.
Conduct dynamic testing for known vulnerabilities	Attempt to find known vulnerabilities in components of the system.
Manually review source code	Examine the application source code to find errors overlooked in the initial development phase.
Run static analyzers and review the results	Run Checkmarx and cppcheck against the Boost.JSON 1.75.0. library.
Review the bugs attached to the Boost.JSON GitHub and identify if any of them are security related.	Identify bugs that are listed on the Boost.JSON GitHub and determine if they are valid.