ASSESSMENT REPORT

THE C PLUS PLUS ALLIANCE, INC.

HYBRID APPLICATION ASSESSMENT 2020

SEPTEMBER 24, 2020

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

Table of Contents **3**
Executive Report **4**
Project Overview 4
Summary of Findings 5
Appendix A — Measurement Scales **14**
Finding Severity 14

EXECUTIVE REPORT

Project Overview

The C Plus Plus Alliance, Inc. engaged Bishop Fox to assess the security of the Beast library. The following report details the findings identified during the course of the engagement, which started on July 20, 2020.

Goals

- Identify critical- and high-risk issues in the Beast library that could be exploited to subvert expected application behavior
- Review Beast for security vulnerabilities with a focus on vulnerabilities documented by OWASP and specific to web application technologies
- Determine whether the design of the Beast library meets secure-by-design principles
- Review Beast example projects and provide security recommendations for developers looking to use the library

Approach

The assessment team conducted a hybrid application assessment with the following targets in scope:

- Beast 1.74.0

Bishop Fox's hybrid application assessment methodology leverages the 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 performs 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.

In this instance, the team also made use of fuzzing with the LLVM sanitizers AddressSanitizer (ASAN) and UndefinedBehaviorSanitizer (UBSAN) for extra runtime analysis to

FINDING COUNTS

0 Critical
0 High
0 Medium
0 Low
0 Informational

0 Total findings

DATES

07/20/2020
Kickoff

07/20/2020 – 09/18/2020
Active testing

09/24/2020
Report delivery

exercise the core functionality of the library and to identify memory management or corruption issues.

Summary of Findings

The assessment team identified no issues during the time-boxed assessment of Beast library version 1.74.0. The team found the Beast library to be resilient to the memory attacks that can plague native libraries. Static analysis returned negligible informational results that did not warrant reporting.

Bishop Fox had previously assessed Beast in November of 2017, identifying an insecure randomness vulnerability that could lead to cryptographically weak masking values. This issue has been reviewed as part of the 2020 engagement and was found to be remediated in the master branch on July 10, 2018. More information about the remediation can be found in the Masking section of the report below.

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 Beast library.

Strategic Approach

Fuzzing Strategy

One of the assessment goals was to attempt to identify vulnerabilities in the way Beast parsed HTTP and WebSocket data. The assessment team found that the majority of Beast examples were set up in a similar fashion, so testing each example individually was not necessary. The `advanced-server` and `websocket-async` examples were chosen for the majority of the fuzzing efforts. The `advanced-server` example was chosen because of its use of both the HTTP and WebSocket functionality. The Bishop Fox team modified the examples by adding a thread for executing AFL++ in persistent mode.

To fuzz the HTTP parser, the team used a standard WebSocket upgrade request along with many more common HTTP headers and the built-in AFL HTTP dictionary. This instance was allowed to run until the number of identified paths stopped growing and the number of cycles completed by the fuzzer matched the paths discovered. The team determined that the fuzzing conducted was sufficient to produce reliable results in the time provided.

To fuzz the WebSocket protocol, the team followed the approach of the 2017 engagement. The `advanced-server` example was selected for the majority of the fuzzing and was found at the following URL:

```
https://github.com/boostorg/beast/tree/boost-1.74.0/example/advanced/server
```

The engagement team modified the `advanced-server` code using the following steps:

- The team used `boostrap.sh`, included with Boost, to configure the library to use the clang compiler and created a `user-config.jam` file to point to the optimized `afl-clang-fast++`:

```
# Used on CI

import os ;

local OPENSSL_ROOT = [ os.environ OPENSSL_ROOT ] ;

project
  : requirements
    <include>$(OPENSSL_ROOT)/include
    <variant>debug:<library-path>$(OPENSSL_ROOT)/lib
    <target-os>windows<variant>debug:<library-path>$(OPENSSL_ROOT)/debug/lib
    <variant>release:<library-path>$(OPENSSL_ROOT)/lib
  ;
using clang : : afl-clang-fast++ ;
```

FIGURE 1 – `user-config.jam file pointing to afl-clang-fast++`

- The `advanced-server` example was then modified to run an `AFL_PERSISTENT` loop to speed up fuzzing cycles. Inside of the thread, the team read from `STDIN` and sent the WebSocket frames or HTTP request using raw sockets. The pseudo-code for the loop function is shown below:

```
void fuzzSocket(unsigned short port)
{
    //setup buffer to hold our payload

    while ( __AFL_LOOP(1000)){

        // Zero buffer before each loop
        memset(buf, 0, BUFSIZE);

        // Read buffer from STDIN
        size_t read_bytes = read(0, buf, BUFSIZE);

        // setup raw socket connection to the designated port

        // Send the payload over the socket

        if (send(sockfd, buf, read_bytes, MSG_NOSIGNAL) != read_bytes) {
            perror("send() failed 1");
            exit(1);
        }
        if (read(sockfd, b, sizeof(b))< 0){
            perror("failed read");
```

```
        }

        // Close our socket for the next loop
        close(sockfd);
    }
    exit(0);
}
```

FIGURE 2 – Pseudocode for fuzzing harness

- Once the harness was built, the team created sample data for AFL++ to consume for payload generation. For the HTTP parser fuzzing, a WebSocket upgrade request was used with additional headers and the AFL HTTP dictionary. For the fuzzing of WebSockets, the team referenced the WebSocket RFC and generated payloads for all message frame options according to the RFC standard.
- The team then started parallel fuzzing with multiple instances of AFL on the server. The port number was also increased for each instance to run on its own thread.

Using the WebSocket sample set described above and persistent mode parallel fuzzing with AFL++, the team produced the following overall results on the `advanced-server` WebSocket test:

```
cycles done : 323
total paths : 323
uniq crashes : 0
uniq hangs : 15
```

FIGURE 3 – Results after parallel fuzzing for a week on `advanced_server`

From the results above, 323 total paths were discovered, and the number of cycles completed by AFL++ matched the paths discovered. There were no hard crashes to the application during the fuzzing exercise and only 15 hangs. The hangs were reviewed by the team and were determined to be inconsequential when testing on a default build of the example.

Static Source Code Analysis

The engagement team used Checkmarx against the Beast codebase for additional coverage and to supplement the fuzzing and manual code review process. Checkmarx flagged two low-risk issues. However, after review, the team determined that these were negligible and irrelevant to the goals of the engagement.

Security Considerations

The Bishop Fox assessment team found Beast to be a mature and stable library. The security issues that are most likely to evolve in the library will emerge from several key areas that are outside of the scope of what Beast can provide the developer. The following sections address some of the more pertinent issues and provide examples of what developers should be aware of.

In addition to the areas highlighted below, the 2017 public report from Bishop Fox contains an appendix with recommendations that are still relevant. The public report can be found in the Security Review section of the Beast documentation online at the following URL:

```
https://www.boost.org/doc/libs/develop/libs/beast/doc/html/beast/quick_start/security_review_bishop_fox.html
```

Data/Input Validation

Beast does not provide any security mechanisms to prevent web or back-end vulnerabilities that may arise due to improper input validation or contextual output encoding. The examples provided by the Beast library provide solid direction on establishing a session and passing data back and forth between a client and the server. The job of a WebSocket is to pass data, in any format, between client and server. The developer must be aware of the context that they are sending that data to and encode or escape data properly. It would be impossible for the team to identify every scenario that a developer will encounter, but the examples below highlight some common issues and how they can be prevented.

GENERAL GUIDANCE

Encoding and escaping should be handled within libraries or functionality built into the chosen web framework or language. Maintained security libraries and functions built into these frameworks will provide more complete and less error-prone functions for properly encoding output. Applications depending on Beast should follow the guidelines below that contribute to a defense-in-depth strategy:

- Data type validation — Verify that the buffer received from the client conforms to the type expected (e.g., ASCII, numeric).
- Value limits — If there is a minimum and maximum allowable value, ensure that the data received by the client is within this range before processing.
- Limited character sets — If the data should contain only a limited character set, this should be validated. For example, an allow list of characters could be implemented using the `Boost.Regex` library.

- If a data format such as JSON is chosen, syntax, or schema validation should occur before processing.

These controls should be implemented server-side in the WebSocket stream `on_read()` implementation. For additional security, the same controls should be placed on the client-side, prior to sending the data, but this does not guarantee that data will arrive at the server in the sanitized condition.

CROSS-SITE SCRIPTING (XSS)

Cross-site scripting is a common web vulnerability and not always considered when using a WebSocket connection. Developers must be aware that if the data consumed by the server is used on a webpage, JavaScript, or CSS, encoding or escaping must take place for the context where the data is being used. The OWASP: Cross Site Scripting Prevention Cheat Sheet referenced in the Additional Resources section below provides examples of how to properly sanitize input from users before it is displayed or used in a web application, and identifies libraries that can be used across many different common web languages.

SQL INJECTION

If data passed on the WebSocket is used in database queries, these queries can be susceptible to SQL injection just like a traditional application. Prepared statements with parameterized queries explicitly distinguish the query from data and are the most reliable defensive measure. Input escaping can also be used to help prevent the likelihood of an injection attack. Escaping rules can be contextual depending on the configuration and type of database server in use. One of the best resources for this issue is the OWASP SQL Injection Prevention Cheat Sheet, found in the Additional Resources section below.

Authentication/Authorization

Authentication and authorization are another critical piece of developing applications, including WebSocket components. When a developer decides to add a WebSocket implementation into their application, these sessions will not be bound by the same rules as the authentication or authorization frameworks that may be in use within the application. The developer must make conscious decisions about how to ensure that WebSocket communications are authenticated. One option is to make use of pre-existing authentication headers or cookies that are transmitted in the upgrade request. Another option that is widely adopted is generating a token such as a cross-site request forgery (CSRF) token to be sent inside of a header or URL parameter on the upgrade request that can be checked by the server before establishing a persistent connection. It is also up to the developer to manage what should be done with sessions after the point of authenticating (i.e., if the socket lifetime should be synchronized with a parallel web-

based session, if the number of sessions allowed to a single user should be limited, or if the user is authorized to access the resource being requested).

WebSockets are not bound by the Same Origin Policy. This can create a potential scenario allowing attackers to establish socket connections from malicious third parties on a user's behalf. It is therefore important that a WebSocket server also check the `origin` header of a WebSocket upgrade request and ensure that it matches the source that is expected.

User authentication information will ideally be parsed and verified prior to the upgrade request transferring ownership of the socket. An example of where this logic would take place is shown below inside the HTTP session `on_read()` implementation of the `advanced_server` example:

```
https://github.com/boostorg/beast/blob/develop/example/advanced/server/advanced_serv
er.cpp
```

```
480.           void
481.               on_read(beast::error_code ec, std::size_t bytes_transferred)
482.               {
483.                   boost::ignore_unused(bytes_transferred);
484.
485.                   // This means they closed the connection
486.                   if(ec == http::error::end_of_stream)
487.                       return do_close();
488.
489.                   if(ec)
490.                       return fail(ec, "read");
491.
492.                   // See if it is a WebSocket Upgrade
493.                   if(websocket::is_upgrade(parser_->get()))
494.                   {
495.                       // Check for authenticated/valid user here, before
    transferring ownership of the socket.
496.                       // Create a websocket session, transferring ownership
497.                       // of both the socket and the HTTP request.
498.                       std::make_shared<websocket_session>(
499.                           stream_.release_socket())->do_accept(parser_-
    >release());
500.                       return;
501.                   }
```

FIGURE 4 – Example location for verification of user identity or authorization

Checking for a valid user session and origin, prior to transferring ownership to the WebSocket, will help prevent unauthorized users from establishing socket streams as well as unnecessarily consuming resources on the server.

Encryption

WebSockets used over a cleartext communication channel should be considered insecure. There are no mechanisms by default to protect information from being modified in transit between a client and server. The Beast library has provided examples of deploying SSL to force the use of the WebSocket Secure (WSS) protocol. WSS should always be used for a production-ready application. If additional security is warranted, then any additional symmetric or asymmetric encryption could be implemented to encapsulate the data between the client and server.

Masking

The WebSocket Protocol RFC 6455 section 10.3 states that masking WebSocket data is put in place from the client-side to prevent intermediary cache poisoning. A key point is that for the masking value to be effective, it must not be guessable by an attacker and must change with each frame sent. If an attacker could guess the next mask or if a non-rotating mask is used, then the cache poisoning attack becomes plausible.

The Bishop Fox assessment team identified an insecure randomness finding in Beast in November of 2017, due to a potentially unsafe function used to generate masking keys. The Beast library has since remediated the issue by implementing a ChaCha stream cipher for generating masking values. The commit addressing this issue can be found at the URL below:

```
https://github.com/boostorg/beast/pull/1181/commits/749e54f31be97e47434e5bcdc2f6cf5b43c94d50
```

Since the above release, the Beast library sets `secure_prng` to `true` by default to use the new algorithm. Additionally, a user can implement their own seed for the algorithm. This improves the implementation provided by Beast for when users cannot or do not want to rely on `std::random_device`. The example snippet below shows how a developer may implement a custom seed source:

```
        std::seed_seq seedSource{
                    genSeed(), genSeed(), genSeed(), genSeed(),
                    genSeed(), genSeed(), genSeed(), genSeed()};

        boost::beast::websocket::seed_prng(seedSource);
        // Create websocket::stream and connect to server below
```

FIGURE 5 – Setting a custom seed source

It is important to note that `seed_prng` must be called before constructing a WebSocket connection from the client side. In the above example, `genSeed()` would be a function that returns a random 32-bit integer. Beast recommends at least 256 bits of entropy be fed into the `seed_seq`, hence the usage of 8 * 32-bit integers in this example. Using the

above format, developers can implement other externally trusted sources of RNG, such as hardware modules, which may be important for embedded systems or critical high-volume applications. Another source may be to use CSPRNG functions included with the OpenSSL library, such as `RAND_bytes()`, or one of the many generators included inside OpenSSL can be used to change the source for `RAND_bytes()` appropriately.

ADDITIONAL BEST PRACTICES

The 2017 Bishop Fox report outlined additional best practice recommendations that are still relevant and can be found below:

- **Harden Builds with Compile-time Security Flags —** Depending on the target architecture and toolset, use available compile-time flags to harden application builds. Common flags include GCC's `-pie` or `-fPIE` to enable ASLR, `-fstack-protector-all` for guard stack protection, and `-WI,nxcompat`, used on Windows to enable Data Execution Prevention (DEP). Additional informational flags such as `-Wall`, `-Wextra`, and `-Wformatsecurity` should be enabled to alert developers to problematic code. It is recommended that `-Werror` be included to turn warnings into compile-time errors. These flags may or may not be included by default as part of the build environment, and it is the responsibility of the developer to understand whether the flags are enabled and how they may impact the compiled binary and application performance. Consult relevant compiler documentation and security resources for information about supported security flags.
- **Conform to SEI CERT C++ Standards —** The SEI CERT C++ Coding Standard enables developers to write secure and reliable systems by documenting a set of common security pitfalls in the C++ language. The collected recommendations are provided as a publicly available PDF document, a link to which has been included in the Additional Resources section below. As there are a large number of rules included in the standard, developers are encouraged to check their applications for standard violations using TS 17961-compliant static source code analysis tools.
- **Avoid Exposing Implementation Details —** The Beast library HTTP examples include a version string that is exposed in the responses `Server` HTTP header. This is not directly a security concern itself but if an issue is identified in the Beast library in the future, it could aid attackers in finding vulnerable versions. It is recommended that developers remove the version number string, usually found in the example projects as `res.set(http::field::server, BOOST_BEAST_VERSION_STRING).`

- **Stay Current with Latest Patches —** Ensure on an ongoing basis that Boost, Beast, and any other supporting libraries are up to date and that a clean build is deployed.

ADDITIONAL RESOURCES

Beast
https://github.com/boostorg/beast

Boost C++ Libraries
http://www.boost.org/

American Fuzzy Lop plus plus
https://github.com/AFLplusplus/AFLplusplus

The WebSocket Protocol
https://tools.ietf.org/html/rfc6455

OWASP: Cross Site Scripting Prevention Cheat Sheet
https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html

OWASP: SQL Injection Prevention Cheat Sheet
https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html

Token-based Header Authentication for WebSockets behind Node.js
https://yeti.co/blog/token-based-header-authentication-for-websockets-behind-nodejs/

WebSockets not Bound by SOP and CORS? Does this mean...
https://blog.securityevaluators.com/websockets-not-bound-by-cors-does-this-mean-2e7819374acc

Cross-Site WebSocket Hijacking (CSWSH)
https://christian-schneider.net/CrossSiteWebSocketHijacking.html

OpenSSL Random Numbers
https://wiki.openssl.org/index.php/Random_Numbers#Generation

SEI CERT C++ Coding Standard
https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637

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.