COMPREHENSIVE REPORT

BEAST

HYBRID APPLICATION ASSESSMENT 2017

DECEMBER 8, 2017

BISHOP FOX

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.

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

TABLE OF CONTENTS

SUMMARY

Project Overview

The author of Beast, Vinnie Falco, engaged Bishop Fox to assess the security of the Boost C++ Beast HTTP/S networking library. The following report details the findings identified during the course of the engagement, which started on September 11, 2017.

Goals

- Identify critical- and high-risk issues (especially memory corruption issues) in the Beast library that could be exploited to subvert expected application behavior
- Review Beast for security vulnerabilities with a focus on those documented by OWASP and specific to web application technologies
- Determine whether the design of the Beast library meets secure-by-design principles
- Thoroughly fuzz the example `advanced-server` Beast library application
- Perform a manual review of Beast library application source code to uncover subtle implementation issues that may impact library security

Finding Counts
1 High
1 Medium

2 Total findings

Scope
Beast Library v.124

SHA1 Commit Hash
`9dc9ca13b9c08c1597d0`
`5bcf6c19be357e426041`

Please see the Summary's Additional Resources for a link to v.124 commit and the author's public key

Dates
09/11/2017
Kickoff

09/11/2017 – 10/31/2017
Active testing

12/8/2017
Report delivery

Approach

The assessment team conducted a hybrid application assessment of the Beast library. Bishop Fox's hybrid application assessment methodology leverages real-world attack techniques through application penetration testing in combination with targeted source code review to identify application security vulnerabilities. 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 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.

The team considered it important to assess the application in a live environment using modern fuzzing frameworks, including both `honggfuzz` and `afl-fuzz`, as well as to identify security vulnerabilities through manual source code review.

All references to Beast in this report are to version 124.

Fuzzing Strategy

A primary goal of the engagement was to identify vulnerabilities in the HTTP and WebSocket protocol parsing implementations. As such, the `advanced-server` application was chosen as it implemented a basic HTTP and WebSocket server, exercised significant portions of the Beast library, and simultaneously reduced the amount of custom code needed to get up and running. The `advanced-server` is found at the following location:

```
https://github.com/boostorg/beast/tree/v124/example/advanced/server
```

The engagement team modified the `advanced-server` code using the following steps to prepare Beast for fuzzing:

1. Start the HTTP server on a randomized port.
2. Read the payload to be sent from a user-specified command-line filename argument.
3. Select the fuzzing mode based on a user-supplied value. (Three modes were implemented: HTTP server fuzzing, WebSocket protocol fuzzing, and WebSocket per-message deflate fuzzing.)
4. Connect to the server.
    a. If fuzzing the HTTP implementation, simply write the payload as is to the socket stream.
    b. If fuzzing the WebSocket protocol implementation, perform the WebSocket upgrade request, and then write the raw WebSocket protocol frame bytes to the socket stream.
    c. If fuzzing the WebSocket per-message deflate implementation, prepare the WebSocket frame header manually, apply the frame header mask value to the payload, and write the resulting frame to the socket stream.
5. Close the connection.

The engagement team used `wireshark` to observe the correctness of the fuzzing code, as shown below:

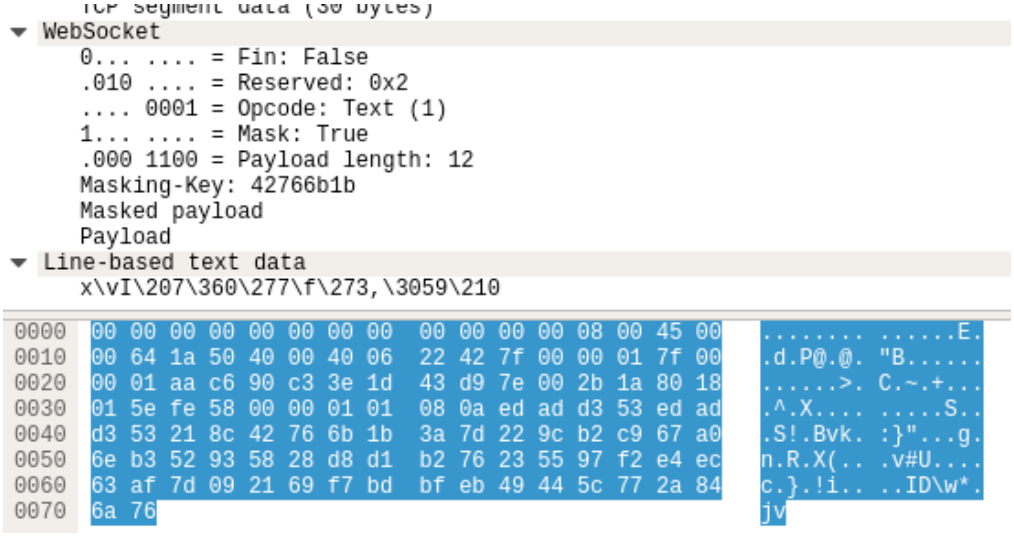

FIGURE 1 - WebSocket protocol fuzzing payload as viewed in `wireshark`

Fuzzing payloads were generated using both `afl-fuzz` and `honggfuzz` built-in functionality. Test cases and the fuzzing implementation will be provided after report publication and internal review.

Static Source Code Analysis

The engagement team executed the Checkmarx scanner against the Beast codebase to achieve additional coverage and supplement the manual source code review and fuzzing activities. Specifically, Checkmarx CX version 8.1.0 was used.

While Checkmarx did flag issues in third-party `zlib` library test code that were deemed unimportant, it flagged only three informational issues in the Beast `core` folder that the assessment team did not consider important enough to include in this report. Due to the time-boxed nature of the engagement and sparse static code analysis results, the assessment team focused on thoroughly fuzzing the server to uncover issues.

Manual Source Code Review

In addition to the fuzzing activities, the assessment team devoted a considerable amount of time to manual source code review with the purpose of both identifying vulnerabilities and understanding how the Beast library actually works. Due to the time-boxed nature of the engagement, an exhaustive line-by-line review was not performed. The manual source code review focused on the HTTP message parsing code, the WebSocket protocol implementation, and the WebSocket per-message deflate extension implementation via a ported `zlib` library. The Insecure Randomness finding detailed below was identified as a result of the manual review.

Summary of Findings

The assessment team identified the following issues as a result of the time-boxed assessment of Beast library version 124:

Denial of Service — The assessment team successfully demonstrated three denial-of-service attacks against Beast by sending malformed WebSocket frames containing a compressed payload. The issues were identified by fuzzing the WebSocket server code responsible for uncompressing client messages.

Insecure Randomness — Beast uses an insufficient source of entropy as a seed value to a linear congruential generator (LCG) in order to generate random values that serve as the masking value when WebSocket client frames are sent. In special circumstances, an attacker may be able to exploit this issue to poison HTTP caches served from improperly implemented intermediaries.

At the time of this writing, no other issues have been identified. Please refer to the Assessment Report below for a technical description of the issues, which also includes steps that can be taken to mitigate the risk.

OBSERVATIONS

- **Library Functionality —** The Beast library provides a set of classes, derived from and consistent with the popular Boost ASIO networking library, to assist developers in writing networking applications that serve HTTP/S clients or consume HTTP/S services. In addition to supporting HTTP 1.0 and 1.1 protocols, the library also implements WebSocket communication functionality. The Beast library relies heavily upon features in the C++11 standard.
- **Security Features —** The Beast library does not implement specific security features related to common web applications. It is the responsibility of library consumers to understand web application security vulnerabilities they may be exposed to, and to develop their applications to account for those issues. This includes vulnerabilities such as cross-site scripting (XSS), cross-site request forgery (CSRF), application-layer denial of service, insecure SSL/TLS configuration, path traversal and file disclosure, command injection, UI redressing, and HTTP header-related vulnerabilities (e.g., insecure cookie settings, HSTS, CORS). The Open Web Application Security Project (OWASP) is a great resource for developers to learn more about common web application security issues.

SECURE DEVELOPMENT RECOMMENDATIONS

Developers should consider the following recommendations when writing applications that use the Beast networking library:

- **Perform Strict Input Validation —** Treat all untrusted input as potentially malicious or malformed, and validate according to length, type, range, and format. Prefer whitelisting of inputs (rejecting based on a known valid set) to blacklisting (accepting based on a known invalid set), when possible. Please refer to Appendix B for additional information regarding the expectations developers should have when implementing secure applications using the Beast library.
- **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 address space layout randomization (ASLR), `-fstack-protector-all` for stack guard 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 includes a version string that is exposed in responses as a header field in the `advanced-server` example (as of Beast version 124). While exposing this information is not necessarily a security vulnerability, it can help attackers identify vulnerable

versions of the library and automate attacks against those versions. Developers should avoid disclosing this information in publicly facing environments.

- **Utilize Encrypted WebSocket Connections —** WebSocket developers should enforce encrypted (`wss://`) WebSocket connects for any application that deals with sensitive data, such as authentication credentials, business data, and personably identifiable information.
- **Stay Current with Latest Patches —** Ensure on an ongoing basis that Boost, Beast, and any other supporting libraries (e.g., OpenSSL) are up to date with the latest patches, and rebuild the application to include the patched versions.

ADDITIONAL RESOURCES

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

Beast v.124 Commit
https://github.com/boostorg/beast/tree/9dc9ca13b9c08c1597d05bcf6c19be357e426041

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

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

Open Web Application Security Project (OWASP)
https://www.owasp.org/index.php/Main_Page

Public Key for Beast Author Vinnie Falco
https://api.github.com/users/vinniefalco/gpg_keys

ASSESSMENT REPORT

The author of the Beast networking library, Vinnie Falco, engaged Bishop Fox to assess the security of Beast v.124 prior to inclusion in Boost as an official Boost networking library. The following report details the findings identified during the course of the engagement, which started on September 11, 2017.

Hybrid Application Assessment

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

- Beast v.124
    - `https://github.com/boostorg/beast/tree/v124`
    - SHA1 commit hash: `9dc9ca13b9c08c1597d05bcf6c19be357e426041`

Identified Issues

1 DENIAL OF SERVICE HIGH

Definition

Denial of service vulnerabilities occur when an attacker prevents authorized users from accessing a resource. This type of attack arises in three ways. First, it can occur when the transmission medium is disrupted between the user and the resource, leaving no path for communication. Second, the target system may be coaxed to reset, oftentimes repeatedly, which forces any established connections to also reset. Third, the target resource is fooled into consuming all available computing resources, thereby leaving no available resources to handle legitimate requests.

Details

The assessment team identified three denial-of-service vulnerabilities while fuzzing the Beast per-message deflate implementation. These vulnerabilities were triggered when the team used `honggfuzz` to fuzz the `advanced-server` example code using the methodology described in the Fuzzing Strategy section above. The assessment team used the `zlib-flate` command-line tool to generate the initial test case used by `honggfuzz`:

```
$ echo “AAAAAAA” | zlib-flate -compress > test-case-dir/websocket-zlib-test-case.req
```

The raw bytes for the compressed payload are given in hexadecimal below:

```
0x9c 0x78 0x74 0x73 0x03 0x04 0x00 0x2e 0xf5 0x08 0xd2 0x01
```

The assessment team launched honggfuzz against advanced-server using the following command line:

```
honggfuzz -u --max_file_size=120 --timeout 1 -e req -f ~/Desktop/fuzzing/test-case-
websocket-zlib/ -- ./bin/gcc-5.4.0/debug/threading-multi/advanced-server 2
___FILE___
```

Executing the above command launched honggfuzz against the team's customized advanced-server binary:

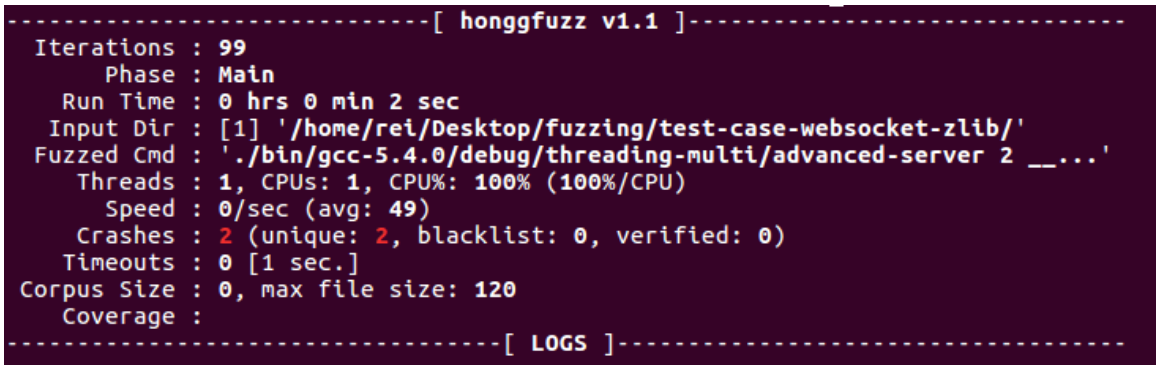

FIGURE 2 - Fuzzing the WebSocket per-message deflate implementation with honggfuzz

The assessment team replayed individual crashes captured by honggfuzz against both the customized advanced-server and an unmodified version from the Beast codebase. Doing so resulted in the discovery of three different code paths that triggered assert statements, resulting in the Beast server aborting execution. For example, the following payload will trigger an assert:

```
0x8c 0xc1 0x76 0x42 0x1c 0x6b 0xea 0x90 0x2e 0x18 0x33 0x8a 0xc9 0x08 0x46 0x61 0xff
0xff 0x46 0xff 0xff 0xff 0x46 0xff 0xff 0xff 0x00 0x45
```

When replayed against advanced-server, the server process aborted with the following error message:

```
$ ./bin/gcc-5.4.0/debug/threading-multi/advanced-server 1
'./SIGABRT.PC.7ffff6e22428.STACK.d99b8115e.CODE.-
6.ADDR.(nil).INSTR.cmp____$0xfffffffffffff000,%rax.2017-12-04.21:43:46.4380.req'
random port: 51010
read 27 bytes from ./SIGABRT.PC.7ffff6e22428.STACK.d99b8115e.CODE.-
6.ADDR.(nil).INSTR.cmp____$0xfffffffffffff000,%rax.2017-12-04.21:43:46.4380.req
advanced-server: /beast/boost/boost_1_65_1/boost/beast/websocket/impl/read.ipp:524:
void boost::beast::websocket::stream<NextLayer>::read_some_op< <template-parameter-
2-1>, <template-parameter-2-2> >::operator()(boost::beast::error_code, std::size_t,
bool) [with MutableBufferSequence =
```

```
boost::beast::basic_multi_buffer<std::allocator<char> >::mutable_buffers_type;
Handler =
boost::beast::websocket::stream<boost::asio::basic_stream_socket<boost::asio::ip::tc
p> >::read_op<boost::beast::basic_multi_buffer<std::allocator<char> >,
boost::asio::detail::wrapped_handler<boost::asio::io_service::strand,
std::_Bind<std::_Mem_fn<void (websocket_session::*)(boost::system::error_code, long
unsigned int)>(std::shared_ptr<websocket_session>, std::_Placeholder<1>,
std::_Placeholder<2>)>, boost::asio::detail::is_continuation_if_running> >;
NextLayer = boost::asio::basic_stream_socket<boost::asio::ip::tcp>;
boost::beast::error_code = boost::system::error_code; std::size_t = long unsigned
int]: Assertion `zs.total_out == 0' failed.
Aborted (core dumped)
```

FIGURE 3 - Confirming application crash via command-line

Examining the Beast source code revealed the line responsible for raising the `assert` (line number 524), which is highlighted below:

```
519                     zs.avail_in = sizeof(empty_block);
520                     ws_.pmd_->zi.write(zs, zlib::Flush::sync, ec);
521                     BOOST_ASSERT(! ec);
522                     // VFALCO See:
523                     // https://github.com/madler/zlib/issues/280
524                     BOOST_ASSERT(zs.total_out == 0);
525                     cb_.consume(zs.total_out);
526                     ws_.rd_size_ += zs.total_out;
527                     bytes_written_ += zs.total_out;
528                     if(
529                         (ws_.role_ == role_type::client &&
```

FIGURE 4 - Beast code responsible for `assert`

The individual payloads known to trigger the `assert` statements are included in the Affected Locations section below.

Affected Locations

Source Files

- `https://github.com/boostorg/beast/blob/v124/include/boost/beast/websocket/impl/read.ipp#L521`
- `https://github.com/boostorg/beast/blob/v124/include/boost/beast/websocket/impl/read.ipp#L524`
- `https://github.com/boostorg/beast/blob/v124/include/boost/beast/websocket/impl/read.ipp#L542`

Hexadecimal-encoded Payloads

- Payload 1 (triggers `read.ipp:521`)

        - `0x8c 0xc1 0x76 0x42 0x1b 0x6b 0x3e 0x10 0xac 0xe2 0x66 0x38 0x45 0x75 0x75 0xe4 0xe4 0x45 0x38 0xac 0x75 0x66 0xe4 0x45 0x45 0x75 0x8a 0xe4`
    - Payload 2 (triggers `read.ipp:524`)
        - `0x8c 0xc1 0x76 0x42 0x1c 0x6b 0xea 0x90 0x2e 0x18 0x33 0x8a 0xc9 0x08 0x46 0x61 0xff 0xff 0x46 0xff 0xff 0xff 0x46 0xff 0xff 0xff 0x00 0x45`
    - Payload 3 (triggers `read.ipp:542`)
        - `0x9c 0x53 0x74 0x73 0x03 0x04 0x00 0x2e 0xf5 0x08 0x37 0x99`

Total Instances **3**

Business Impact

The potential impact of a denial-of-service attack greatly depends on how and by whom the application is used. Applications that deal heavily in financial transactions may suffer both financial and brand damage as the result of a denial-of-service attack. An unavailable application that provides critical business data may result in users not being able to do their jobs.

In the case of the Beast HTTP networking library, three denial-of-service issues were identified in the WebSocket per-message deflate implementation that resulted in application crashes. Beast implementations that enabled per-message deflate on the server to take advantage of the performance benefits of per-message deflate could be the target of denial-of-service attacks by malicious actors aware of this vulnerability.

Recommendations

Denial-of-service attacks can occur through a number of vectors, and as a result, require unique remediation activities. To prevent or mitigate the impact of denial-of-service attacks, the assessment team recommends the following steps:

- For unpatched versions of the Beast library, disable per-message deflate as a temporary mitigation.
- Update Boost to version 1.66.0, which is the first official public version of Boost to include the Beast library. This release includes the fix for the vulnerability described herein.

Additional Resources

Beast v.124 Commit
https://github.com/boostorg/beast/tree/9dc9ca13b9c08c1597d05bcf6c19be357e426041

Compression Extension for WebSocket
https://tools.ietf.org/html/rfc7692

Application Denial of Service
http://www.owasp.org/index.php/Application_Denial_of_Service

RFC 7692: Compression Extensions for WebSocket
https://tools.ietf.org/html/rfc7692

2 INSECURE RANDOMNESS

MEDIUM

Definition

Insecure randomness errors occur when a function that produces predictable values is used as a source of randomness in a security-sensitive context. Oftentimes, standard pseudo-random number generators (PRNGs), which cannot withstand cryptographic attacks, are used to generate random numbers for security purposes.

Details

The engagement team identified two vulnerabilities related to how Beast creates WebSocket client frame header mask values, which are included in every client-initiated WebSocket request. The WebSocket RFC requires that WebSocket client implementations create frame mask values using a secure, unpredictable method in order to protect against HTTP cache poisoning attacks. The following paragraph is taken from section `10.3 – Attacks on Infrastructure (Masking)` of the WebSocket protocol as described in RFC 6455:

```
Clients MUST choose a new masking key for each frame, using an algorithm that cannot
be predicted by end applications that provide data.  For example, each masking could
be drawn from a cryptographically strong random number generator. If the same key is
used or a decipherable pattern exists for how the next key is chosen, the attacker
can send a message that, when masked, could appear to be an HTTP request (by taking
the message the attacker wishes to see on the wire and masking it with the next
masking key to be used, the masking key will effectively unmask the data when the
client applies it).
```

First, the team noted that Beast relied upon `std::random_device` to produce a random value at the following location:

```
63  maskgen_t<_>::rekey()
64  {
65      std::random_device rng;
66  #if 0
67      std::array<std::uint32_t, 32> e;
68      for(auto& i : e)
69          i = rng();
70      // VFALCO This constructor causes
71      //        address sanitizer to fail, no idea why.
72      std::seed_seq ss(e.begin(), e.end());
73      g_.seed(ss);
74  #else
75      g_.seed(rng());
76  #endif
77  }
```

FIGURE 5 - `std::random_device` used to seed `std::minstd_rand`

The result of the above operation was then used to seed the MINSTD linear congruential generator via the C++11 standard library `std::minstd_rand` class:

```
79  // VFALCO NOTE This generator has 5KB of state!
80  //using maskgen = maskgen_t<std::mt19937>;
81  using maskgen = maskgen_t<std::minstd_rand>;
82
83  //------------------------------------------------
```

FIGURE 6 - Frame mask values generated via `std::minstd_rand`

There are two problems with this approach. First, the random number produced by `std::random_device` is not guaranteed to be a source of entropy, as the implementation is platform specific. Second, even if the values produced by `std::random_device` were secure, `std::minstd_rand` cannot be relied upon to produce unpredictable values, as it is possible to brute-force the initial seed used by observing a sequence generated by repeated calls to `std::minstd_rand`.

Affected Locations

Source Files

- `https://github.com/boostorg/beast/blob/v124/include/boost/beast/websocket/detail/mask.hpp#L65`
- `https://github.com/boostorg/beast/blob/v124/include/boost/beast/websocket/detail/mask.hpp#L81`

Total Instances 2

Business Impact

The amount of unpredictability in a random number generator is directly correlated to the level of security provided when it is used in a security-sensitive situation. If a malicious user can even slightly predict the behavior of a random number generator, they may be able to leverage this information to help breach the security of a system.

The unpredictability of the frame mask values helps protect WebSocket users against HTTP cache poisoning attacks when unencrypted WebSocket messages are processed by intermediary servers. The team found that the implementation of frame masking by Beast could allow an attacker to replace the cache entry for a legitimate resource with malicious data, thereby affecting the integrity of application data. Note: This issue affects only WebSocket clients, as servers do not mask messages when communicating with clients.

Recommendations

The root cause of this problem is two-fold. First, `std::random_device` may or may not be deterministic, depending on the implementation, which is platform specific. Second, `std::minstd_rand` is an LCG and not suitable for producing unpredictable random numbers. The following recommendation is intended for the Beast developer in order to remediate this issue:

- Replace the reliance on `std::random_device` and `std::minstd_rand` with a cryptographically secure pseudo random number generator (CSPRNG).

To mitigate the risk of this issue, the assessment team recommends that Beast users take the following steps:

- While not a perfect mitigation, enforce the use of WebSocket over SSL (`wss://`), as the frame masking issue affects only intermediaries that process cleartext WebSocket traffic. This may not resolve the issue for certain deployments that terminate SSL connections and then forward the SSL traffic through additional intermediaries.
- Upgrade to the latest version of Beast once this issue has been addressed.

Note: This issue affects only WebSocket clients implemented with Beast. Server implementations are not affected and do not need to mitigate this problem.

Additional Resources

Insecure Randomness on OWASP
http://www.owasp.org/index.php/Insecure_Randomness

CWE-330: Use of Insufficiently Random Values
http://cwe.mitre.org/data/definitions/330.html

Cryptographically Secure Pseudo-Random Number Generators
https://en.wikipedia.org/wiki/Cryptographically_secure_pseudorandom_number_generator

NIST: Random Bit Generation
https://csrc.nist.gov/projects/random-bit-generation

The WebSocket Protocol: Attacks on Infrastructure (Masking)
https://tools.ietf.org/html/rfc6455#section-10.3

Untwisting the Mersenne Twister: How I Killed the PRNG
https://www.bishopfox.com/blog/2014/08/untwisting-mersenne-twister-killed-prng/

Talking to Yourself for Fun and Profit
http://w2spconf.com/2011/papers/websocket.pdf

APPENDIX A — MEASUREMENT SCALES

The assessment team used the following criteria to rate the findings in this report. Bishop Fox derived these risk ratings from the industry and organizations such as OWASP.

Finding Severity

The severity of each finding in this report is independent. Finding severity ratings combine direct technical and business impact with the worst-case scenario in an attack chain. The more significant the impact and the fewer vulnerabilities that must be exploited to achieve that impact, the higher the severity.

Critical	Vulnerability is an otherwise high-severity issue with additional security implications that could lead to exceptional business impact. Examples include trivial exploit difficulty, business-critical data compromised, bypass of multiple security controls, direct violation of communicated security objectives, and large-scale vulnerability exposure.
High	Vulnerability may result in direct exposure including, but not limited to: the loss of application control, execution of malicious code, or compromise of underlying host systems. The issue may also create a breach in the confidentiality or integrity of sensitive business data, customer information, and administrative and user accounts. In some instances, this exposure may extend farther into the infrastructure beyond the data and systems associated with the application. Examples include parameter injection, denial of service, and cross-site scripting.
Medium	Vulnerability does not lead directly to the exposure of critical application functionality, sensitive business and customer data, or application credentials. However, it can be executed multiple times or leveraged in conjunction with another issue to cause direct exposure. Examples include brute-forcing and client-side input validation.
Low	Vulnerability may result in limited exposure of application control, sensitive business and customer data, or system information. This type of issue provides value only when combined with one or more issues of a higher risk classification. Examples include overly detailed error messages, the disclosure of system versioning information, and minor reliability issues.
Informational	Finding does not have a direct security impact but represents an opportunity to add an additional layer of security, is considered a best practice, or has the possibility of turning into an issue over time. Finding is a security-relevant observation that has no direct business impact or exploitability, but may lead to exploitable vulnerabilities. Examples include poor communication between engineering organizations, documentation that encourages poor security practices, and lack of security training for developers.

APPENDIX B — SECURE DEVELOPMENT WITH BEAST

Developers who intend to implement the Beast HTTP library in their own products should understand what is required in order to develop secure code. The Beast library is a low-level HTTP and WebSocket networking library, and as such, does not implement security controls found in modern, full-featured web application development frameworks.

Beast Security Features

The Beast HTTP library aims to provide the following security features:

- **Secure HTTP and WebSocket Protocol Parsing —** HTTP and WebSocket messages are parsed in a safe, consistent, and efficient manner free of memory corruption issues including stack and heap buffer overruns, and insecure use of freed or uninitialized memory addresses.
- **Fail Closed Operation —** When unrecoverable errors are encountered, Beast prefers to abort server execution rather than risk an unknown application state affecting user and data security.
- **Limited HTTP Field Validation —** Beast performs limited validation and handling for the `Connection`, `Proxy-Connection`, `Content-Length`, `Transfer-Encoding`, and `Upgrade` fields. More information regarding these fields is available here:

  Boost: Beast Library – Protocol Primer
  http://www.boost.org/doc/libs/master/libs/beast/doc/html/beast/using_http/protocol_primer.html#beast.using_http.protocol_primer.special_fields

Beast Developer Responsibilities

While not intended to be comprehensive, developers should be aware that it is their responsibility to address the following security concerns:

- **HTTP Header Security —** Developers are responsible for implementing HTTP header security as appropriate for their application. For example, applications that convey an authenticated state via session cookies will be responsible for ensuring that session IDs are generated securely, that the cookies are invalidated and deleted when the current session exits, that the `HttpOnly` and `secure` flags are set, and that the `domain` and `path` values are appropriate and not overly broad. Cleartext sensitive data should never be stored in web application cookie values.

Modern web application servers also utilize a number of HTTP headers to harden their applications from attack and protect their users and data. This includes the `X-Frame-Options` header, which is used to help prevent UI redressing attacks, and the `Strict-Transport-Security` header, which is used to instruct user-agents that HTTPS is required for all client-server communications.

Developers who use a header-based redirect via the `Location` field should be aware that it is often a target of abuse by malicious users, and should take the appropriate steps to mitigate that risk. For example, a developer may choose to perform strict input validation against the `Location` value to forbid offsite redirects, or to limit the target of those redirects to a set of whitelisted domains.

The `Content-Security-Policy` (CSP) header may be implemented to achieve additional hardening of web applications against client-side malicious script injection attacks (commonly referred to as cross-site scripting, or XSS). A CSP can be used to limit the domains from which JavaScript can be sourced, prohibit the inlining of JavaScript, and forbid JavaScript functions like `eval`, among other actions.

Developers that wish to expose their application to third-party domains may need to implement cross-origin resource sharing via the `Access-Control-Allow-Origin` and other `Access-Control` headers. CORS implementers must be careful when parsing the HTTP `Origin` header value received from clients to ensure that security controls cannot be bypassed.

Developers must exercise caution when outputting user-supplied values into HTTP header fields. Failure to account for unexpected input, like the carriage return and line feed, may allow a malicious user to construct a completely new HTTP response with malicious input, potentially poisoning intermediary HTTP caches.

Please refer to the OWASP Secure Headers Project for additional information regarding how to use HTTP headers to harden web applications from attack and protect web application users and data:

OWASP Secure Headers Project
https://www.owasp.org/index.php/OWASP_Secure_Headers_Project

- **Cross-site Scripting (XSS) —** One of the most common web application vulnerabilities identified today is that of cross-site scripting, or XSS. This vulnerability typically occurs when user input is reflected into web application HTML, JavaScript, CSS, or other resources without appropriate validation and encoding. There are three types, broadly speaking: reflected, persistent, and DOM-based. Developers must validate input received from end users and choose the correct output encoding when using that input in application responses. Please refer to the following OWASP resource for additional information:

  OWASP: Cross-site Scripting (XSS)
  https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

- **WebSocket Security —** Beast developers should be aware that they are responsible for implementing authentication, authorization, and auditing controls when using the Beast WebSocket library. Implementations should perform strict validation of any user-supplied `Origin` values to ensure they match expected values, and `Origin` should not be relied upon to make security decisions as it is easily forged. Applications that send authentication credentials or sensitive user and/or business data should enforce the use of WebSocket over TLS (`wss://`). Please refer to the following OWASP resource for additional information:

  OWASP: Testing WebSockets
  https://www.owasp.org/index.php/Testing_WebSockets_(OTG-CLIENT-010)

- **TLS Security and Certificate Validation —** Beast developers who wrap HTTP and WebSocket streams in TLS are responsible for validating server and client certificates (when performing mutual authentication). Developers will be responsible for ensuring that their code uses an up-to-date version of OpenSSL, a secure version of the TLS protocol (TLS 1.2 is the latest, non-draft version at the time of this writing), and secure cipher suite algorithm (e.g., AES/GCM). While modern OpenSSL libraries typically support certificate validation routines (e.g., checking the hostname or expiration date), developers are responsible for understanding what is supported by the library versus what must be directly implemented. Please refer to the following resource for additional information:

OpenSSL Wiki: SSL/TLS Client
https://wiki.openssl.org/index.php/SSL/TLS_Client

- **Path Traversal —** Web application servers commonly serve files and other media based on the URI supplied by clients. Path traversal vulnerabilities can occur when a web application server does not validate the user-supplied URI path data to retrieve these files. Beast developers are responsible for validating URI paths; canonicalizing the supplied path to a single, unambiguous representation; and restricting resource requests to only an approved set of files and directories under the web application root directory. Please refer to the following OWASP resource for additional information:

  OWASP: Path Traversal
  https://www.owasp.org/index.php/Path_Traversal

- **Server-side Request Forgery (SSRF) —** Server-side request forgery vulnerabilities occur when an application accepts unvalidated user input and then uses that input to typically fetch an HTTP resource. Beast implementations that use the library to fetch HTTP resources on behalf of users must validate the request to ensure that users can only request resources from approved domains, URI paths, and ports. Failure to validate this data may allow a malicious user to fetch resources from internal networks, the server's file system, and probe the internal network for interesting services. Please refer to the following resource for additional information:

  OWASP: Server-side Request Forgery
  https://www.owasp.org/index.php/Server_Side_Request_Forgery

- **Cross-site Request Forgery (CSRF) —** Cross-site request forgery, or the confused deputy problem, exists due to the stateless nature of HTTP applications and how cross-domain requests are treated with respect to the automatic forwarding of session cookies. Essentially, attackers who lure users to a site under their control may be able to force those users to perform an unauthorized action on a web application they are authenticated with if that web application does not implement the proper safeguards. A common way to mitigate CSRF vulnerabilities is by double POSTing session cookie values, or including an additional anti-CSRF token in the

body of each POST operation, or as a custom header. Developers are responsible for implementing anti-CSRF controls in their web applications if the applications support authenticated user sessions. Please refer to the following OWASP resource for additional information:

OWASP: Cross-site Request Forgery (CSRF)
https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

- **Additional Vulnerabilities —** The above vulnerability classes and security considerations represent a handful of those that developers may encounter when using the Beast library. Depending on the application under development, and integrated technologies, developers need to be aware of additional vulnerability classes including but not limited to: SQL injection (SQLi), command injection, LDAP injection (LDAPi), XML external entities (XXE) injection, template injection, time-of-use-to-time-of-check attacks, insecure process privileges, insecure cryptography, and insecure randomness, among many more. As it is beyond the scope of this document to detail each of the aforementioned vulnerabilities, developers are encouraged to review public security resources to better understand their responsibilities and how these security risks can be mitigated in their own applications. Please refer to the following resources for additional information:

  OWASP
  https://www.owasp.org/index.php/Main_Page

  OWASP: Top 10 2013
  https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf