ASSESSMENT REPORT

RIPPLE LABS INC.

SIDE CHAINS SECURITY ASSESSMENT 2023

JUNE 15, 2023

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 Ripple Labs 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

Ripple Labs Inc. engaged Bishop Fox to assess the security of the XRP Ledger side chains feature. The following report details the findings identified during the course of the engagement, which started on May 3, 2023.

Goals

- Identify vulnerabilities in systems and services exposed on the internet-facing services related to the XRP side chains bridging infrastructure
- Assess the overall security of the rippled bridging protocol, witness server, and RPC interfaces
- Enumerate any weaknesses or potential vulnerabilities within the rippled XRP bridging feature that could be remediated to improve the feature's security posture

FINDING COUNTS
2 Medium
5 Low
1 Informational

8 Total findings

SCOPE
rippled XRP bridging component

xbridge witness server

DATES
05/03/2023
Kickoff

05/10/2023 – 05/26/2023
Active testing

06/15/2023
Report delivery

Summary of Findings

The assessment team performed a security assessment of the XRP bridging feature, including a fork of rippled that implemented the XRP bridging protocol and the corresponding xbridge_witness server. During this assessment, the team determined that the RPC networking interfaces were well-protected against memory manipulation attacks and identified no core issues with the bridging functionality.

In contrast, the assessment team did discover multiple issues related to the applications' build processes, such as the use of outdated and vulnerable software and a lack of binary hardening. While such issues were not directly exploited during the assessment, they reduce the security posture of the affected applications.

Additionally, the team enumerated multiple issues with lesser impact that did not present an immediate risk to the XRP bridging functionality, including minor issues related to authentication checks, legacy build artifacts, and insecure building processes.

As the XRP bridging feature was still under active development, the team could not perform certain tests, such as analysis of the finalized `.DEB` and `.RPM` packages. Additionally, several issues affecting rippled from a previous engagement had not been remediated. Information on these findings can be found in the `Ripple – XRP Ledger Security Assessment 2021 – Assessment Report – 20220629.pdf` document. Furthermore, the assessment team provided information regarding these unremediated findings via Ripple's Slack channels and GitHub issue tracker. These issues are not included in this report as they did not directly relate to the XRP bridging feature. However, the assessment team recommends mitigating these issues as they may indirectly affect the XRP bridge.

Overall, despite attempting multiple potential attack paths against the XRP bridge, the team did not identify mechanisms for a remote attacker to violate the memory integrity of the XRP bridge applications or forge bridging transactions.

RECOMMENDATIONS

Update Software Dependencies — Switch to current stable versions of upstream software and ensure dependencies are frequently updated to reduce known vulnerabilities.

Harden rippled Binary, Build Environment, and Linux Packaging — Implement best practices in the build process to mitigate attacks and remove opportunities for attackers to escalate access.

ASSESSMENT REPORT

Hybrid Application Assessment

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

- xbridge_witness application (`https://github.com/seelabs/xbridge_witness`)
- rippled application (`https://github.com/seelabs/rippled/tree/xbridge`)

Identified Issues

1 INSUFFICIENT BINARY HARDENING MEDIUM

Definition

To combat the risk of security vulnerabilities, applications can use security mitigation techniques to limit specific forms of attack. For C and C++ applications, memory corruption issues and other vulnerabilities can lead to catastrophic results including arbitrary code execution. Therefore, it is essential for high-risk applications to make use of all available defense mechanisms to ensure that the exploitation of code issues is as difficult as possible.

Details

The assessment team identified multiple types of missing binary hardening options for the rippled and xbridge_witness applications on Linux. When modern hardening features such as stack canaries and relocation read-only (RELRO) are not enabled when building an application, built applications may lack defense-in-depth mitigations that prevent attackers from successfully exploiting a vulnerability.

To identify the issue, the team compiled the programs on a Debian system and leveraged `checksec.sh`, a tool to analyze an executable's properties, to determine which features had been enabled:

```
$ checksec --file=xbridge_witnessd
RELRO           STACK CANARY      NX  P          IE           …omitted for brevity…
Partial RELRO   No canary found   NX enabled     PIE enabled  …omitted for brevity…
```

FIGURE 1 - Analyzing `xbridge_witnessd` built on Debian via `checksec.sh`

When hardening options are not explicitly set, the fallback hardening configuration heavily depends on the distribution-specific behavior of the available compiler. As a result, release builds on different systems may have varying degrees of protection

applied to the generated binary. As shown above, neither stack canaries nor complete RELRO protection were explicitly enabled in the default build for the xbridge_witness server. With such features disabled, attackers may more easily craft exploits that target memory safety issues.

Affected Locations

Affected Applications

- rippled
- xbridge_witness

Total Instances **2**

Recommendations

To mitigate the risk of insufficient mitigations, the assessment team recommends the following actions:

- Ensure that all possible hardening steps are applied on release builds by default regardless of the build environment.
- Use the strongest available hardening level where possible.
- Ensure that newly introduced build-hardening flags are correctly passed to build dependencies to avoid unprotected subprojects.
- Consider warning or aborting the build process if essential requirements for hardening mechanisms are not present.

Additional Resources

CWE-693: Protection Mechanism Failure
https://cwe.mitre.org/data/definitions/693.html

2 VULNERABLE SOFTWARE MEDIUM

Definition

Vulnerable software exists when an application has not been updated with the latest security patches. These insecure versions of software can contain issues (e.g., arbitrary remote code execution or SQL injection) that could allow a malicious user to gain elevated access to the application itself or its supporting infrastructure.

Details

When analyzing the rippled and xbridge_witness repositories' use and integration of third-party dependencies, the team identified several dependencies or libraries that either contained known potential security issues or had not been updated to recent versions. When an application uses unpatched third-party libraries, vulnerabilities or security weaknesses may be introduced into the application.

The assessment team reviewed the `conanfile.py` and `CMake` files used by the rippled and xbridge_witness build processes and found multiple dependencies with known security concerns. For example, both repositories depended on a vulnerable version of `libarchive` version 3.6.0 that did not contain security mitigations. The repositories also relied on security updates applied in version 3.6.2 to fix a missing `NULL` check vulnerability (CVE-2022-36227):

```
$ cat xbridge_witness/conanfile.py
    …omitted for brevity…
    requires = [
        'boost/1.77.0',
        'date/3.0.1',
        'libarchive/3.6.0',
        'lz4/1.9.3',
        'grpc/1.50.1',
        'nudb/2.0.8',
        'openssl/1.1.1m',
        'protobuf/3.21.4',
        'snappy/1.1.9',
        'soci/4.0.3',
        'sqlite3/3.38.0',
        'zlib/1.2.12',
        'fmt/8.1.1'
    ]
    …omitted for brevity…
    def requirements(self):
        if self.options.jemalloc:
            self.requires('jemalloc/5.2.1')
        if self.options.reporting:
            self.requires('cassandra-cpp-driver/2.15.3')
            self.requires('libpq/13.6')
        if self.options.rocksdb:
```

```
            self.requires('rocksdb/6.27.3')
            if platform.startswith('linux'):
                self.requires('liburing/2.2')

 exports_sources = (
       'CMakeLists.txt', 'Builds/*', 'bin/getRippledInfo', 'src/*', 'cfg/*'
   )
```

FIGURE 2 - Dependencies referenced in `xbridge_wtiness/conanfile.py`

Other dependencies lacked recent patches with potential security updates but no known CVEs. For example, the included version of `snappy`, 1.1.9, did not contain a fix for undefined behavior that was applied in version 1.1.10.

The team also identified dependencies that had available updates but no known security issues. For example, both repositories utilized version 5.2.1 of `jemalloc`, which lacked three years of fixes that could be rectified by updating to version 5.3.0.

Finally, the team also enumerated outdated and potentially vulnerable software within multiple referenced Docker container builds in the rippled repository, as shown below:

```
$ cat rippled/Builds/containers/ubuntu-builder/Dockerfile
ARG DIST_TAG=18.04
FROM ubuntu:$DIST_TAG
```

FIGURE 3 - Example reference to Ubuntu 18.04 within rippled repository

The team identified multiple uses of Ubuntu version 18.04 within the container build systems present in the rippled repository. The assessment team confirmed that Ubuntu 18.04 had reached end of life in May 2023. Builds leveraging this version of Ubuntu may contain unmitigated vulnerabilities and rely on older versions of `libc`, `clang`, and `gcc` that do not contain newer security features present in more up-to-date versions.

Affected Locations

For a full list of affected dependencies, please see the attached spreadsheet.

Total Instances 17

Recommendations

To mitigate the risk of vulnerable software, the assessment team recommends the following action:

- Update the affected libraries to their latest versions.

Strategic Considerations

- Introduce a mechanism into the CI/CD pipeline of the affected applications to automatically identify and escalate issues related to outdated or vulnerable dependencies.
- Regularly review security vulnerability lists and vendor advisory pages related to applications used within the environment. Apply all security patches released for these systems in a timely manner.

Additional Resources

OWASP Top Ten 2021 - Vulnerable and Outdated Components
https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/

CWE-937: Using Components with Known Vulnerabilities
https://cwe.mitre.org/data/definitions/937.html

3 AUTHENTICATION CHECK WITHOUT CONSTANT TIME

LOW

Definition

When software performs a comparison, such as checking if a submitted password matches the correct password, in non-constant time, it may be possible to perform a time-based brute force attack against the logic where an attacker could determine each character of a password based on how quickly they get a response from the server.

Details

When analyzing the authentication enforcement capabilities embedded in the xbridge_witness application, the team determined that the RPC server performed an authentication check in non-constant time. Without the use of a constant-time algorithm, an attacker could potentially perform a time-based brute-force attack against the service and use the response time to guess the correct password one character at t time.

To identify the issue, the team reviewed the logic within the `RPCHandler.cpp` file that compared the submitted password to the valid password, as shown below:

```
$ cat xbridge_witness/src/xbwd/rpc/RPCHandler.cpp
…omitted for brevity…
    if (ac.pass)
   {
       bool userMatch = params.isMember("Username") &&
           params["Username"].isString() &&
           params["Username"].asString() == (*ac.pass).user;
       bool passwordMatch = passwordOp && *passwordOp == (*ac.pass).password;
       if (!userMatch || !passwordMatch)
           return false;
   }
…omitted for brevity…
```

FIGURE 4 - Password comparison without constant time

As shown above, the authentication check did not leverage a secure constant-time comparison function or hash the valid password and password candidate prior to comparing them. While such a time-based attack would be difficult to perform in practice, implementing such best practices ensures that the application is not exposing unnecessary attack surfaces to malicious actors.

Affected Locations

Affected File

`xbridge_witness/src/xbwd/rpc/RPCHandler.cpp:637-640`

Total Instances 1

Recommendations

The assessment team recommends the following actions to address the insufficient authorization controls:

- Leverage a secure constant-time function to compare the submitted password with the valid password.
- Alternatively, prior to comparing the passwords, hash both the submitted and valid passwords and compare the hashes.

Additional Resources

CWE-208: Observable Timing Discrepancy
https://cwe.mitre.org/data/definitions/208.html

OWASP - Compare Password Hashes Using Safe Functions
https://cheatsheetseries.owasp.org/cheatsheets/Authentication_Cheat_Sheet.html#compare-password-hashes-using-safe-functions

4 INSECURE NETWORK TRANSMISSION LOW

Definition

Insecure network transmission occurs when sensitive information is sent over a network without adequate protection. When data is sent across insecure communication channels, it may be susceptible to interception and modification by third parties, resulting in unauthorized information disclosure.

Details

The assessment team reviewed the build pipelines for the rippled repository and determined that many of the containers in the GitLab CI pipeline depended on the Docker image `thejohnfreeman/rippled-build-ubuntu:4b73694e07f0`, as shown below:

```
$ cat rippled/.gitlab-ci.yml
…omitted for brevity…
 image: thejohnfreeman/rippled-build-ubuntu:4b73694e07f0
```

FIGURE 5 - Docker image referenced in GitLab CI pipeline

The team investigated the specific source code revision used to build this image on GitHub and identified instances where the build process downloaded and installed software over an insecure HTTP channel without the use of TLS, as shown below:

```
doxygen_slug="doxygen-${doxygen_version}"
curl --location --remote-name \
 "http://doxygen.nl/files/${doxygen_slug}.src.tar.gz"
tar xzf ${doxygen_slug}.src.tar.gz
rm ${doxygen_slug}.src.tar.gz
```

FIGURE 6 - Docker install script insecurely downloading `doxygen`

The assessment team found a similar pattern in multiple variants of the Docker build including the builds for the images based on Ubuntu 22.04, 21.10, and 20.04. During the build process, if the traffic were intercepted, an attacker could perform a Man-in-the-Middle (MitM) attack and replace the download with malicious code.

Affected Locations

Affected Files

- `rippled-docker/ubuntu-22.04/install.sh:106-111`
- `rippled-docker/ubuntu-21.10/install.sh:119-124`
- `rippled-docker/ubuntu-20.04/install.sh:109-114`

Total Instances **3**

Recommendations

To mitigate the threat of data interception and modification, the assessment team recommends the following step:

- Enforce the use of TLS when downloading and installing packages.

Additional Resources

OWASP Cheat Sheet Series - Transport Layer Protection
https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html

CWE-319: Cleartext Transmission of Sensitive Information
https://cwe.mitre.org/data/definitions/319.html

Wikipedia - HTTP Strict Transport Security
http://en.wikipedia.org/wiki/HTTP_Strict_Transport_Security

5 INSECURE SOFTWARE CONFIGURATION LOW

Definition

Insecure software configuration occurs when applications and infrastructure are configured in a manner that is inconsistent with industry best practices. These misconfigurations could allow unauthorized access to affected systems, the disclosure of sensitive information, and the exposure of critical application logs.

Details

The assessment team reviewed the configurations behind the Docker container builds used by the rippled XRP bridging applications and identified multiple misconfigurations that weakened the security posture of the runtime environment. While these issues could not be actively exploited by the assessment team, mitigating such issues would provide additional protections if alternative attack paths were identified.

The team identified software downloaded in the container build process without the use of integrity checks. For example, in the `install.sh` script for the Ubuntu 22.04 Docker image, the script pulled CMake over an HTTPS connection but did not perform an integrity check on the downloaded artifact, as shown below:

```
cmake_slug="cmake-${cmake_version}"
curl --location --remote-name \
"https://github.com/Kitware/CMake/releases/download/v${cmake_version}/${cmake_slug}.
tar.gz"
tar xzf ${cmake_slug}.tar.gz
rm ${cmake_slug}.tar.gz
```

FIGURE 7 - CMake downloaded without integrity check

While downloading the artifact over HTTPS ensures that the traffic cannot be manipulated when downloaded, it does not ensure that the upstream package manager or GitHub account has not maliciously modified the artifact or been compromised.

Additionally, the team also determined that the Docker installation scripts contained multiple issues related to the LLVM APT repository. For example, the `install.sh` scripts used HTTP for the repository URL instead of HTTPS, as shown below:

```
# Add sources for Clang.
curl --location https://apt.llvm.org/llvm-snapshot.gpg.key | apt-key add -
cat <<EOF >/etc/apt/sources.list.d/llvm.list
deb http://apt.llvm.org/${ubuntu_codename}/ llvm-toolchain-${ubuntu_codename}-
${clang_version} main
deb-src http://apt.llvm.org/${ubuntu_codename}/ llvm-toolchain-${ubuntu_codename}-
${clang_version} main
EOF
```

FIGURE 8 - LLVM PGP key lacking fingerprint check and APT repository without HTTPS

In addition to not using HTTPS for the repository location, no fingerprint was checked against the downloaded PGP key. Therefore, although the traffic to download applications from the repository cannot be manipulated, there appeared to be no protection against the compromise of the server itself because the APT source location and location of the downloaded PGP key resided on the same web server endpoint.

The assessment team also determined that the rippled Docker container's Dockerfiles did not downgrade privileges via the `USER` directive, as shown below:

```
FROM ubuntu:22.04
WORKDIR /root
COPY install.sh .
RUN ./install.sh && rm install.sh
```

FIGURE 9 - Dockerfile for Ubuntu 22.04 lacking `USER` privilege downgrade

Without the `USER` directive, the container ran any application as the root user inside the container itself. While escaping the container, if compromised, would require an additional vulnerability or attack path not identified during the assessment, ensuring the application does not run as `root` by default could mitigate certain attack paths.

Affected Locations

Affected Files

- `rippled-docker/ubuntu-22.04/install.sh`
- `rippled-docker/ubuntu-22.04/Dockerfile`
- `rippled-docker/ubuntu-21.10/install.sh`
- `rippled-docker/ubuntu-21.10/Dockerfile`
- `rippled-docker/ubuntu-20.04/install.sh`
- `rippled-docker/ubuntu-20.04/Dockerfile`
- `rippled-docker/ubuntu-18.04/install.sh`
- `rippled-docker/ubuntu-18.04/Dockerfile`

Total Instances 8

Recommendations

To mitigate the risk of software misconfigurations, the assessment team recommends the following action:

- Leverage user-based security separation by switching the runtime context to an unprivileged user where possible.

- Avoid installing GPG keys over a network without checking the keys' fingerprints.
- Where possible, use multi-stage container configurations to combine special images with different strengths.
- Where possible, reduce the attack surface within containers by using a minimal set of installed applications.

Additional Resources

CWE-653: Improper Isolation or Compartmentalization
https://cwe.mitre.org/data/definitions/653.html

CWE-1125: Excessive Attack Surface
https://cwe.mitre.org/data/definitions/1125.html

CWE-250: Execution with Unnecessary Privileges
https://cwe.mitre.org/data/definitions/250.html

Dockerfile reference, section USER
https://docs.docker.com/engine/reference/builder/#user

Docker documentation on multi-stage builds
https://docs.docker.com/develop/develop-images/multistage-build/

6 UNDEFINED BEHAVIOR LOW

Definition

When an application contains code or design patterns that are not prescribed by its programming language as being predictable, the application may contain undefined behavior. The impact from undefined behavior varies depending on its form and runtime characteristics. Regardless, undefined behavior should be avoided when programming an application to ensure that the application performs as intended.

Details

The assessment team determined that the use of the `boost` library prior to version 1.81 triggered an undefined behavior warning in the rippled application when compiled with `clang-16`. Undefined behavior can introduce application reliability issues that could lead to potential vulnerabilities in the application.

To identify the issue, the assessment team built the application and observed the following warning:

```
…omitted for brevity…
In file included from ./boost/mpl/integral_c.hpp:32:
 ./boost/mpl/aux_/integral_wrapper.hpp:73:31: error: integer value -1 is outside the
valid range of values [0, 3] for this enumeration type [-Wenum-constexpr-conversion]
 typedef AUX_WRAPPER_INST( BOOST_MPL_AUX_STATIC_CAST(AUX_WRAPPER_VALUE_TYPE, (value
- 1)) ) prior;
 ./boost/mpl/aux_/static_cast.hpp:24:47: note: expanded from macro
'BOOST_MPL_AUX_STATIC_CAST'
#   define BOOST_MPL_AUX_STATIC_CAST(T, expr) static_cast<T>(expr)
                                               ^
…omitted for brevity…
```

FIGURE 10 - Undefined behavior warning observable when building rippled

The team recommends mitigating undefined behavior to prevent unexpected impact on some compilers or with some optimization levels such as on custom builds for different system architectures.

Affected Locations

Affected Library

`boost` < 1.81

Total Instances 1

Recommendations

To mitigate the risk of undefined behavior, the assessment team recommends the following actions:

- Refactor the affected code to remove undefined behavior.
- Update any affected libraries containing undefined behavior.

Additional Resources

CWE-190: Integer Overflow or Wraparound
https://cwe.mitre.org/data/definitions/190.html

CWE-191: Integer Underflow (Wrap or Wraparound)
https://cwe.mitre.org/data/definitions/191.html

CWE-758: Reliance on Undefined, Unspecified, or Implementation-Defined Behavior
https://cwe.mitre.org/data/definitions/758.html

7 UNNECESSARY CODE

LOW

Definition

Program code that exists within an application but is not accessed or used at any point during the program's execution is referred to as unnecessary code. Unnecessary code is found within applications that have not been properly sanitized before deployment or during ongoing maintenance. The existence of unnecessary code often indicates either improper maintenance of the codebase or the incorrect implementation of application logic. In both cases, this can create both an increased attack surface and additional attack vectors within an application. This situation commonly occurs when ongoing updates replace older code but the older code is not properly removed.

Details

The assessment team identified potentially unused build configurations within the xbridge_witness code repository. When legacy build configurations are not removed from source code, application developers or end users may inadvertently use or reference the outdated configurations to build the application.

The team discovered these unused build configurations by reviewing the CI/CD configuration files present in the repository, as shown below:

```
$ cat xbridge_witness/circle.yml
…omitted for brevity…
   - sudo apt-get purge -qq libboost1.48-dev
   - sudo apt-get install -qq libboost1.60-all-dev
   - sudo apt-get install -qq clang-3.6 gcc-5 g++-5 libobjc-5-dev libgcc-5-dev
libstdc++-5-dev libclang1-3.6 libgcc1 libgomp1 libstdc++6 scons protobuf-compiler
libprotobuf-dev libssl-dev exuberant-ctags
   - lsb_release -a
   - sudo update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-5 99
   - sudo update-alternatives --install /usr/bin/g++ g++ /usr/bin/g++-5 99
   - sudo update-alternatives --force --install /usr/bin/clang clang /usr/bin/clang-
3.6 99 --slave /usr/bin/clang++ clang++ /usr/bin/clang++-3.6
   …omitted for brevity…
```

FIGURE 11 - Legacy `circle.yml` configuration containing outdated software

In this `circle.yml` build configuration, the team identified outdated versions of `gcc`, `libboost`, `cmake`, and `clang`. The assessment team confirmed with the application team that the affected CI/CD builds were no longer in use and should be removed.

Affected Locations

Source Code

- `xbridge_witness/.travis.yml`

- `xbridge_witness/circle.yml`
- `xbridge_witness/appveyor.yml`
- `xbridge_witness/attn_srv.json`

Total Instances 4

Recommendations

To prevent the risks involved with unnecessary code, the assessment team recommends the following step:

- Remove all unnecessary code from the application.

Additional Resources

CWE-489: Active Debug Code
https://cwe.mitre.org/data/definitions/489.html

Wikipedia - Dead code
http://en.wikipedia.org/wiki/Dead_code

8 LACK OF TEST COVERAGE

INFORMATIONAL

Definition

When an application's code base does not contain an adequate test suite, unknown bugs may be introduced into the application without any mechanisms to identify them without manual analysis.

Details

The assessment team determined that the xbridge_witness repository did not contain a full suite of unit and integration tests. While this concern does not represent a security vulnerability, missing test cases increase the likelihood of regressions or new bugs being introduced into the program.

To identify the issue, the team first attempted to run the unit test cases embedded in the built binary but determined that the command line flag had not been properly implemented to include the tests, as shown below:

```
$ ./xbridge_witnessd --unittest
0ms, 0 suites, 0 cases, 0 tests total, 0 failures
```

FIGURE 12 - Indication of missing unit tests in application

Next, the team analyzed the test cases available within the repository and determined that only certain features contained associated tests:

```
$ ls xbridge_witness/src/test | cat
KeyFileGuard.h
ValidatorKeys_test.cpp
ValidatorKeysTool_test.cpp
```

FIGURE 13 - Identifying available test cases

The assessment team reached out to the application team and confirmed that the application team was already aware of the issue and working on improving the test coverage.

Affected Locations

Affected Repository
xbridge_witness

Total Instances 1

Recommendations

The assessment team recommends the following steps to mitigate the risk of a lack of test coverage:

- Ensure that the code base contains proper unit tests for major components.
- Introduce integration tests into the application to ensure components correctly operate with one another.
- Include abuse case tests in the test suite to ensure that the code correctly handles unexpected or malicious input.

Additional Resources

CWE CATEGORY: 7PK - Code Quality
https://cwe.mitre.org/data/definitions/398.html

OWASP — Misuse/Abuse Testing
https://owaspsamm.org/model/verification/requirements-driven-testing/stream-b/

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 signature and authentication code validation issues	Attempt to alter or forge signed data that the application trusts as genuine.
Perform dynamic testing for function-level authorization controls	Attempt to perform actions or functions with a user or role that should be restricted from those actions and functions.
Conduct dynamic testing for mass assignment	Attempt to find mass assignment vulnerabilities.
Perform dynamic testing for directory traversal	Attempt to manipulate file paths so that they refer to files that are not intended to be accessed.
Conduct dynamic testing for resource-based authorization controls	Attempt to access resources that a user should be restricted from accessing.
Review the general build toolchain security	Analyze the build toolchain scripts and configuration for security weaknesses such as insecure downloads and other problematic actions.
Perform dynamic testing for code injection	Attempt to provide malicious input to code being constructed with user-provided content in order to cause the interpreter to execute the provided code.
Perform dynamic testing for command injection	Attempt to edit the contents of command-line calls using special characters inside user-provided parameters within the construction of the command.
Perform dynamic testing for the insecure	Locate places where the system sends sensitive data without proper safeguards.

TEST CASE	DESCRIPTION
transmission of sensitive data	
Review the handling of cryptographic secrets in the application	Analyze the application use of cryptographic libraries and custom functions for handling of cryptographic secrets to determine weaknesses such as information disclosure.
Conduct dynamic testing for the insecure handling of sensitive data	Attempt to find places where the system mishandles sensitive data, either by sending it to users that should not have that data or by storing it insecurely.
Perform dynamic testing for weak symmetric encryption	Attempt to find issues with symmetric encryption implementation that may cause sensitive data to be disclosed.
Test the application for susceptibility Denial-of-Service (DOS) scenarios	Perform application and network based Denial-of-Service (DOS) based attacks against the application.
Review dependency confusion vulnerabilities	Review application dependencies for potential exploitation through typosquatting or private vs. public repository sourcing.
Identify instances of the application throwing uncaught exceptions	Perform both a manual code review and dynamic testing to determine if the application throws unhandled exceptions when put in an unexpected state or presented with unexpected inputs.
Perform dynamic testing for file-handling vulnerabilities	Locate any files being mishandled in a manner that allows interaction with the server's filesystem or command execution on the server.
Perform dynamic testing for uncommon injection flaws	Attempt to find XML injection, LDAP injection, NoSQL injection, or expression language injection.
Perform NoSQL injections	Provide malicious input to NoSQL-based queries to perform unintended actions on a NoSQL database or to execute malicious code and unvalidated input within the application itself.

TEST CASE	DESCRIPTION
Review the potential misuse of logging statements by external actors	Analyze the use and configuration of log statements in the context of external interfaces to identify potential log pollution opportunities for attackers.
Conduct dynamic testing for information disclosure	Attempt to find responses that contain overly verbose information about the system under review.
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 authentication requirements	Evaluate the strength of the password requirements used for password-based logins and determine whether multi-factor authentication is used for sensitive logins.
Perform dynamic testing for session lifecycle issues	Evaluate sessions used by the system to ensure common issues are not present.
Perform forward tracing of security critical functionality in source code	Identify and analyze areas of critical functionality in code. Find and target system-specific goals and areas of high security relevance. Perform manual analysis of the security controls around these areas to ensure adherence to overall security goals.
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 used by the application.
Perform software composition analysis against system dependencies	Perform automated software composition analysis and checking of dependencies against public vulnerability lists. Investigate all findings to determine practical impact against the target system based on the usage and conditions necessary for exploitability.

TEST CASE	DESCRIPTION
Complete automated static code analysis of relevant codebases	Perform automated static code analysis for all codebases that are not open source. Manually investigate all findings for true positives and assess specific system risks.
Conduct dynamic testing for SQL injection	Attempt to edit the contents of a SQL query by inserting special characters inside user-provided parameters that are used to construct the query.
Perform dynamic testing for server-side request forgery (SSRF)	Attempt to manipulate some or all of the URLs in use by the system.
Perform dynamic testing for server-side template injection	Attempt to specify server-side template code that is evaluated by a template engine in order to execute arbitrary code on the affected system.
Analyze application source code for instances of Undefined Behavior (UB)	Manually review source code and leverage compiler tooling to identify instances of Undefined Behavior.
Conduct dynamic testing for XML external entities (XXE)	Attempt to find misconfigured XML parsers that allow the interpretation of XXE.