Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Arcane Finance

RFQ Market Maker

Veridise Inc.

February 19, 2024

► **Prepared For:**

Arcane Finance
https://www.arcane.finance/

► **Prepared By:**

Bryan Tan
Nicholas Brown

► **Contact Us:** contact@veridise.com

► **Version History:**

Feb. 19, 2024 V1

Contents

1 Executive Summary

From Jan. 31, 2024 to Feb. 8, 2024, Arcane Finance engaged Veridise to review the security of their RFQ Market Maker. Veridise conducted the assessment over 2 person-weeks, with 2 engineers reviewing code over 1 week on commit `ca9e744915`. The auditing strategy involved an extensive manual code review of the source code performed by Veridise engineers.

Project summary. The security assessment covered the source code of an RFQ Market Maker, a reference implementation of a market maker client used with the Arcane Finance protocol. The RFQ Market Maker is a backend web application consisting of three microservices: the `price-client` that fetches token exchange rates from the CoinGecko API*, the `ws-client` that communicates with the Arcane Finance Router†, and the `signature` service that produces cryptographically signed token swap quotes. The `ws-client` service will receive price requests from the Arcane Finance Router via a WebSocket connection, each of which the `ws-client` will then forward to the `price-client` and `signature` services to generate a cryptographically-signed price quote. To complete the price request, the corresponding quote will then be sent to the router.

Code assessment. The RFQ Market Maker developers provided the source code of the RFQ Market Maker‡ for review. The source code consists of TypeScript and Rust source files, and it appears to be mostly original code written by the RFQ Market Maker developers. It contains some documentation in the form of READMEs. To facilitate the Veridise auditors' understanding of the code, the RFQ Market Maker developers also shared a document describing a high-level overview of their system. The source code did not contain a test suite.

Summary of issues detected. The audit uncovered 2 issues, consisting of 1 low-severity rounding issue (V-AFM-VUL-001) and 1 informational finding (V-AFM-VUL-002) as assessed by the Veridise auditors. The RFQ Market Maker developers have not resolved any of the issues as of the time of writing.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

* An API served by `https://coingecko.com` that provides information about token prices
† Another backend web application implemented by Arcane Finance, which the Veridise engineers also performed a security review of in parallel. The details of the Arcane Finance Router audit can be found in a separate audit report.
‡ The source code is publicly available at `https://github.com/arcane-finance-defi/rfq-market-maker`

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
RFQ Market Maker	`ca9e744915`	Typescript, Rust	Application Server

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jan. 31 - Feb. 8, 2024	Manual & Tools	2	2 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Fixed	Acknowledged
Critical-Severity Issues	0	0	0
High-Severity Issues	0	0	0
Medium-Severity Issues	0	0	0
Low-Severity Issues	1	0	1
Warning-Severity Issues	0	0	0
Informational-Severity Issues	1	0	1
TOTAL	2	0	2

Table 2.4: Category Breakdown.

Name	Number
Logic Error	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of RFQ Market Maker's Typescript and Rust code. In our audit, we sought to answer questions such as:

- Do all microservices in the RFQ Market Maker properly handle errors?
- Does the `price-client` service interact with the CoinGecko API correctly?
- Are there any numerical issues with how token-related values are computed?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a manual analysis by human experts.

Scope. The scope of this audit is limited to the `priceclient/src`, `ws-client/src`, and `signature/src` folders of the source code provided by the RFQ Market Maker developers, which contains the Typecript and Rust implementation of the RFQ Market Maker.

Methodology. Veridise auditors reviewed the RFQ Market Maker documentation. They then began a manual audit of the code. During the audit, the Veridise auditors regularly met with the RFQ Market Maker developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-AFM-VUL-001	Rounding issues when calculating amountOut	Low	Open
V-AFM-VUL-002	Decimals field in tokens.json is unused	Info	Open

4.1 Detailed Description of Issues

4.1.1 V-AFM-VUL-001: Rounding issues when calculating amountOut

Severity	Low	Commit	ca9e744
Type	Logic Error	Status	Open
File(s)	ws.handler.ts		
Location(s)	handleWsMessage()		
Confirmed Fix At	N/A		

The `ws-client` service implements a market maker client that is meant to respond to price requests sent by the Arcane Finance router. Such requests are serviced by the `handleWsMessage()` function, which will take a price request, compute an `amountOut` for the swap, and send back a cryptographically signed quote that contains (among other fields) the `amountOut`.

The `amountOut` calculation is prone to rounding errors which may lead to values being lower than intended. Specifically:

1. The `amountIn` (a string containing the token amount as a fixed point integer) is first converted to a JavaScript number (a 64-bit IEEE 754 floating point number).
2. The `amountIn` is multiplied by the `exchangeRate` (a JavaScript number) cached by the `price-client` service, resulting in another JavaScript number. This can result in rounding errors if the result is sufficiently large.
3. The result is then converted into an arbitrary precision floating point number and then rounded to an arbitrary precision integer with no decimals.

```
1 const amountOut = BigInt(new BigNumberJs(Number(priceWsRequest.amountIn) *
      exchangeRate).toFixed(0))
```

Snippet 4.1: Line in `handleWsMessage()` that computes `amountOut`

Impact A large `amountIn`, `exchangeRate`, and/or multiplication result may result in rounding errors. For example, consider the following code snippet:

```
1 const inVal = '18446744073709551615'
2 const exchange = 2
3
4 const amountOut = BigInt(new BigNumberJs(Number(inVal) * exchange).toFixed(0))
5 console.log(amountOut) // 36893488147419103000n
6 // The correct answer is: 36893488147419103230n
```

Rounding errors may cause the maker client to return quotes with prices that are worse for the user than they are intended to be, which could result in financial losses.

Recommendation Convert both `amountIn` and `exchangeRate` to arbitrary precision numbers (e.g., with `BigNumberJs`) before multiplying. Note that the BigNumber constructor may need to be configured with a larger `DECIMAL_PLACES` value if supporting any token types with more than 20 decimals.

4.1.2 V-AFM-VUL-002: Decimals field in tokens.json is unused

Severity	Info	**Commit**	ca9e744
Type	Maintainability	**Status**	Open
File(s)	`price-client/tokens.json`, `price-client/src/main.ts`		
Location(s)	updateExchangeRates()		
Confirmed Fix At	N/A		

The `price-client` service in the RFQ market maker client is used to periodically fetch token prices from CoinGecko. The list of tokens to fetch is stored in a JSON file `tokens.json` at the root of the `price-client` project. Each token entry has a `decimals` field indicating the decimals of the corresponding token.

```
{
  "id": 1,
  "coinGeckoId": "tether",
  "decimals": 6
}
```

Snippet 4.2: Example of a token entry in `tokens.json`

However, when `updateExchangeRates()` is called to fetch the latest token prices, the call to the CoinGecko API does not use the `decimals` value for the token to specify the precision of the result.

```
await Promise.all(
  tokens.map(async ({ coinGeckoId, id }) => {
    const priceData = await coinGeckoClient.simplePrice({
      ids: coinGeckoId,
      vs_currencies: 'usd',
    })
    exchangeRates[id] = priceData[coinGeckoId].usd
  }),
)
```

Snippet 4.3: Call to CoinGeckoClient in `updateExchangeRates()`

Note that CoinGecko's API will return the currency values as floating point numbers.

Impact Ignoring the decimals value for the tokens may be intentional in the current implementation. If so, it can cause confusion for future developers if this isn't documented. It also may be confusing to future developers to define interfaces that are unused, because they will not know how the interface is intended to be used.

Recommendation

- If this behavior is intentional, document why the `decimals` field is unused.
- If `decimals` is not meant to be used, remove it from the file.

