Veridise. Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Arcane Finance

Arcane Router

Veridise Inc.
February 16, 2024

► **Prepared For:**

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

► **Prepared By:**

Bryan Tan
Nicholas Brown

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

► **Version History:**

Feb. 16, 2024	V1

Contents

Contents iii

1 Executive Summary 1

2 Project Dashboard 3

3 Audit Goals and Scope 5
- 3.1 Audit Goals 5
- 3.2 Audit Methodology & Scope 5
- 3.3 Classification of Vulnerabilities 5

4 Vulnerability Report 7
- 4.1 Detailed Description of Issues 8
  - 4.1.1 V-AFR-VUL-001: Router does not guarantee quote is executable 8
  - 4.1.2 V-AFR-VUL-002: Signature for price response is not validated 9
  - 4.1.3 V-AFR-VUL-003: Price request data not cleared for unsuccessful responses 10
  - 4.1.4 V-AFR-VUL-004: Missing check to prevent multiple quotes from the same client 11
  - 4.1.5 V-AFR-VUL-005: Unchecked assumptions in checkForDoPriceResponse 12
  - 4.1.6 V-AFR-VUL-006: Consider rate limiting POST /api/price 13
  - 4.1.7 V-AFR-VUL-007: Memory leak when response timeout is reached 14
  - 4.1.8 V-AFR-VUL-008: Potential race condition in completePriceRequest . . . 16
  - 4.1.9 V-AFR-VUL-009: Timeout race condition in doPriceResponse 18
  - 4.1.10 V-AFR-VUL-010: Potentially missing return statement 20
  - 4.1.11 V-AFR-VUL-011: Typo in price_pesponse table name 21

Glossary 23

Executive Summary 1

From Jan. 31, 2024 to Feb. 8, 2024, Arcane Finance engaged Veridise to review the security of the Arcane Router component of their Arcane Finance protocol. Veridise conducted the assessment over 2 person-weeks, with 2 engineers reviewing code over 1 week on commit `859396f8e4`. 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 the Arcane Router, an HTTP service that responds to user price requests for token swaps on the Arcane Finance protocol. Specifically, users may provide the Arcane Router with an amount of some token they would like to trade for another token. The Arcane Router will then forward the request to third-party *market maker clients* that are connected to the router, each of which may respond with a cryptographically-signed price quote. The quote with the best price will be sent to the user, who can then execute the swap on the Aleo blockchain by providing the quote to the Arcane RFQ smart contract [*].

Code assessment. The Arcane Router developers provided the source code of the Arcane Router for review. The source code appeared to be original code written by the Arcane Router developers. To facilitate the Veridise auditors' understanding of the code, the Arcane Router developers shared a document describing the high-level idea of the project. However, no documentation was contained in or accompanied the source code itself. The source code did not contain a test suite.

Summary of issues detected. The audit uncovered 11 issues, 1 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, quotes returned by the router are not guaranteed to be executable on-chain (V-AFR-VUL-001), allowing a malicious market maker client to perform a denial-of-service attack against all users on the protocol. The Veridise auditors also identified 3 medium-severity issues, including some missing validations (V-AFR-VUL-002, V-AFR-VUL-004) and race conditions (V-AFR-VUL-008), as well as 1 warning and 1 informational finding. The Arcane Router developers have not resolved any of the issues as of the time of writing.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Arcane Router. First, we recommend that the Arcane Router developers not deploy this application until several of the denial-of-service problems, such as V-AFR-VUL-001, are fixed. In addition, we recommend that the Arcane Router developers clarify their threat model for maker clients, as it appears from the code that malicious maker clients were not taken into consideration in the design of the application. Finally, we recommend that the Arcane Router

[*] Veridise audited this smart contract previously, but the audit report of the contract is not publicly available at the time of writing. Should it be made public in the future, it will be available on Veridise's website at `https://veridise.com/audits/`

developers add a unit test suite to ensure the application behaves correctly in both successful and unsuccessful scenarios.

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.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Arcane Router	`859396f8e4`	Typescript	Node.js, Aleo

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	1	0	1
Medium-Severity Issues	3	0	3
Low-Severity Issues	5	0	5
Warning-Severity Issues	1	0	1
Informational-Severity Issues	1	0	1
TOTAL	11	0	11

Table 2.4: Category Breakdown.

Name	Number
Logic Error	4
Denial of Service	2
Data Validation	2
Race Condition	2
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Arcane Router's WebSocket service. In our audit, we sought to answer questions such as:

- Does the router correctly validate all data sent by third-parties?
- Can a malicious maker client perform a denial of service on the router?
- Can a malicious maker client get an invalid/suboptimal quote to be sent to the end user?
- Are there any race conditions that could cause the router to behave unexpectedly?
- Does the router properly manage WebSocket client connections?
- Are there any memory leaks in the router?

3.2 Audit Methodology & Scope

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

Scope. The scope of this audit is limited to the `src/` and `db/` folders of the source code provided by the Arcane Router developers, which contains the implementation of the Arcane Router.

Methodology. Veridise auditors first reviewed the provided documentation as well as previous relevant audit reports. Then, the Veridise auditors conducted a manual review of 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

Vulnerability Report 4

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-AFR-VUL-001	Router does not guarantee quote is executable	High	Open
V-AFR-VUL-002	Signature for price response is not validated	Medium	Open
V-AFR-VUL-003	Price request data not cleared for unsuccessful. . .	Medium	Open
V-AFR-VUL-004	Missing check to prevent multiple quotes from t. . .	Medium	Open
V-AFR-VUL-005	Unchecked assumptions in checkForDoPriceRespo	Low	Open
V-AFR-VUL-006	Consider rate limiting POST /api/price	Low	Open
V-AFR-VUL-007	Memory leak when response timeout is reached	Low	Open
V-AFR-VUL-008	Potential race condition in completePriceRequest	Low	Open
V-AFR-VUL-009	Timeout race condition in doPriceResponse	Low	Open
V-AFR-VUL-010	Potentially missing return statement	Warning	Open
V-AFR-VUL-011	Typo in price_pesponse table name	Info	Open

4.1 Detailed Description of Issues

4.1.1 V-AFR-VUL-001: Router does not guarantee quote is executable

Severity	High	**Commit**	859396f
Type	Denial of Service	**Status**	Open
File(s)	`ws.handler.ts`		
Location(s)	N/A		
Confirmed Fix At	N/A		

A malicious market maker client could perform a denial-of-service attack on the protocol by always submitting an unrealistically favorable `priceResponse` that will never be accepted on-chain. Such a `priceResponse` will likely be sent to the user because it will have a better price than all other makers' responses. The maker could construct such a price response by taking advantage of V-AFR-VUL-002, or the maker could remove any funds it has deposited in the on-chain contract prior to the user executing the swap.

Impact A malicious maker client will be requested for a quote on *every* price request and can therefore disable the entire protocol for *all* users.

Recommendation Implement mitigations against quotes returned by malicious maker clients. Some ideas include:

- Fix V-AFR-VUL-002
- Force maker funds to be put into escrow on-chain when they return a quote, with the funds only being released if the user executes the swap or if the quote expires. This would likely also require a mechanism to ensure that users cannot arbitrarily lock up funds belonging to makers.
- Return all price responses instead of just the best ones, so that users can try the swaps in order from best price to worst price.
- Implement some sort of anomaly detection for the price responses, so that prices that are significantly lower than historical prices are discarded.
- Allow the user to specify a minimum and maximum price so that malicious makers cannot specify very favorable prices.

4.1.2 V-AFR-VUL-002: Signature for price response is not validated

Severity	Medium	**Commit**	859396f
Type	Data Validation	**Status**	Open
File(s)	`websocket.service.ts`		
Location(s)	receivePriceData()		
Confirmed Fix At	N/A		

When the `WebSocketService.receivePriceData()` function is called to handle a price response from a market maker, it does not validate the signature of the quote attached to the price response.

Impact If the router chooses to complete a price request with a price response that has an invalid signature, then the user will be unable to execute the swap on-chain. Similar to V-AFR-VUL-001, this would allow a malicious market maker to perform a denial-of-service attack against the protocol by always returning invalid signatures (assuming their quoted price is the lowest).

Recommendation Validate the signature of the price response before considering it as a candidate to forward to the user.

4.1.3 V-AFR-VUL-003: Price request data not cleared for unsuccessful responses

Severity	Medium	**Commit**	859396f
Type	Logic Error	**Status**	Open
File(s)	`websocket.service.ts`		
Location(s)	doPriceResponse()		
Confirmed Fix At	N/A		

If the router sends a price request to all the maker clients but does not receive any responses for the request, then the `doPriceResponse()` method will be called after some timeout period. In this case, the originating HTTP request to `POST /api/price` will be fulfilled with an error message. However, it does not clear the entries in `priceRequests` and `priceResponses`, like is done for the case where there is at least one price response.

```
private async doPriceResponse(priceRequestId: number): Promise<void> {
  const priceRequestWithResObj = this.priceRequests.get(priceRequestId)
  if (!priceRequestWithResObj) return

  const { priceRequest, res: resObj } = priceRequestWithResObj

  const priceResponses = this.priceResponses[priceRequestId]

  if (!priceResponses) {
    resObj.json({
      error: 'No price responses',
    })
    return
  }

  /* ... code that handles least one price response ... */

  //clear data for price request
  this.priceRequests.delete(priceRequestId)
  delete this.priceResponses[priceRequestId]
}
```

Snippet 4.1: Relevant code in `doPriceResponse()`

Impact This can lead to entries for expired price requests remaining in these objects permanently, causing a memory leak. Lingering entries in `priceResponses` may increase the severity of other business logic errors, including V-AFR-VUL-010 and V-AFR-VUL-008.

Recommendation Clear the entries in `priceRequests` and `priceResponses` immediately after checking that they exist.

4.1.4 V-AFR-VUL-004: Missing check to prevent multiple quotes from the same client

Severity	Medium	**Commit**	859396f
Type	Data Validation	**Status**	Open
File(s)	`websocket.service.ts, priceResponse.repository.ts`		
Location(s)	createPriceResponse()		
Confirmed Fix At	N/A		

The code for creating a price response doesn't check if the same client has already responded to the request. This can allow a client to submit multiple responses for the same request.

```
try {
  const priceResponse = priceResponseRepository.save(createData)

  return priceResponse
} catch (err: any) {
  return new Error(‘Error creating price response: ${err.message}‘)
}
```

Snippet 4.2: The code in `createPriceResponse()` that saves the price response to the database.

Impact Due to V-AFR-VUL-005, if a maker client responds quickly enough to the same price request with multiple responses, they could greatly increase the chance that their quote is the best quote. Consequently, slower clients would be unable to submit a better quote, which means that the user wouldn't get the best quote. This would allow malicious clients to perform a denial-of-service attack against other maker clients.

Recommendation Add a uniqueness constraint to the pair of fields `priceRequestId` and `marketMakerId` in the database, so that two rows with the same value in both fields cannot be created. This will cause the call to `createPriceResponse()` to throw an error in the event that a maker responds to a price request that it has already responded to.

4.1.5 V-AFR-VUL-005: Unchecked assumptions in checkForDoPriceResponse

Severity	Low	**Commit**	859396f
Type	Logic Error	**Status**	Open
File(s)	`websocket.service.ts`		
Location(s)	checkForDoPriceResponse()		
Confirmed Fix At	N/A		

The `checkForDoPriceResponse()` function is called when receiving a price response from a maker client, and it appears to be used to determine if the price request can be completed early before the timeout period. Concretely, the code will check if the number of currently connected clients is less than the number of price responses received by the maker client. This implicitly assumes that (1) maker clients will not disconnect or connect while a price request is being handled; and (2) that each maker client will only submit one response. However, none of these assumptions are validated, so a malicious client could submit many price responses to trigger the calculation of the best price before other clients have responded.

```
private checkForDoPriceResponse(priceRequestId: number): void {
  if (this.wsClients.size <= this.priceResponses[priceRequestId].length) {
```

Snippet 4.3: The check at the beginning of `checkForDoPriceResponse()`.

Impact

- If a maker client disconnects after submitting a response, this condition will be met before all of the other clients have responded. Thus, some (future) responses may be ignored, even if they are submitted within the timeout window.
- If a new client connects, it won't have received the most recent price requests, but the new client will count towards this check. Thus, the check won't ever be true for these requests (if all clients are honest). This is fine, because the timeout will cause the responses to be processed eventually, but this is likely not intended behavior.
- If some makers don't respond to every request, this check may rarely succeed.
- Malicious clients can also abuse this behavior, such as to perform a denial-of-service attack. See V-AFR-VUL-004.

Recommendation Remove this check and only process requests from the timeout call to `doPriceResponse()`. This will avoid the potential race conditions discussed in V-AFR-VUL-009, and will make the code more clear.

4.1.6 V-AFR-VUL-006: Consider rate limiting POST /api/price

Severity	Low	**Commit**	859396f
Type	Denial of Service	**Status**	Open
File(s)	See description		
Location(s)	N/A		
Confirmed Fix At	N/A		

The `POST /api/price` HTTP endpoint can be very expensive to service because the following actions will occur:

- Router will create a row in the database for a new price request
- Router will send the price request to all maker clients
- Each maker client will:
  - Perform an unspecified number of I/O and network operations
  - Perform potentially expensive cryptographic signing operations
  - Send a response back to the router

A malicious actor could spam the `POST /api/price` endpoint to trigger a significant amount of network and/or other I/O operations. The router code does not appear to have any functionality, such as rate limits, to mitigate such spam.

Impact A malicious actor could perform a denial-of-service attack against the Arcane Finance router, which would allow them to degrade the performance of the protocol. In particular, some maker clients may be unable to respond to price requests in time due to higher load. This will cause users to receive worse prices.

Furthermore, if a maker client is cooperating with the malicious agent, an amplification attack could also be performed by having the maker client send `POST /api/price` requests to the router whenever it receives a price request. Such an attack would likely saturate the resources of the router, assuming the maker clients collectively have more resources than the router.

Recommendation Impose rate limits on the `POST /api/price` endpoint.

4.1.7 V-AFR-VUL-007: Memory leak when response timeout is reached

Severity	Low	**Commit**	859396f
Type	Logic Error	**Status**	Open
File(s)	`websocket.service.ts`		
Location(s)	getPrice()		
Confirmed Fix At	N/A		

When a `POST /api/price` HTTP request is sent to the router to request a swap price, the `WebSocketService.getPrice()` function will be called to service the request. Specifically, the price request will be forwarded to all connected maker clients. The `getPrice()` function will also launch a `timeout` task that will ensure that an HTTP response will be sent back to the user after some time, either with the best price returned or with a rejection. The `timeout` value will be stored in the `responseTimeouts` field so that it can be cancelled later if all makers respond early.

```
const timeout = setTimeout(() => {
  this.doPriceResponse(priceRequest.id)
}, WEBSOCKET_RESPONSE_TIMEOUT_MS)

this.responseTimeouts.set(priceRequest.id, timeout)

this.sendGetPriceToWsClients({
  amountIn,
  tokenIn,
  tokenOut,
  priceRequestId: priceRequest.id,
})
```

Snippet 4.4: Lines in `getPrice()` that launch the timeout task and forward the price request to the connected maker clients.

However, the `checkForDoPriceResponse()` function—which is only called when a response from the maker client is received—is the only location where the entry in `responseTimeouts` is removed. If not all maker clients respond, then the callback in `timeout` will only execute `doPriceResponse`, so that `timeout` is never removed from `responseTimeouts`.

Impact If the price request is not completed before `timeout` is triggered, then there will be a memory leak because `responseTimeouts[priceRequest.id]` will never be deleted.

Furthermore, this will (partially) enable a race condition to occur when a maker client responds and the timeout triggers at the same time, which can contribute to bugs such as V-AFR-VUL-008.

Recommendation Move the timeout-clearing logic in `checkForDoPriceResponse()` to the top of `doPriceResponse()` .

```
private checkForDoPriceResponse(priceRequestId: number): void {
  if (this.wsClients.size <= this.priceResponses[priceRequestId].length) {
    console.log(‘Clear timeout for id ${priceRequestId}‘)
    const responseTimeout = this.responseTimeouts.get(priceRequestId)

    if (!responseTimeout) return

    clearTimeout(responseTimeout)

    this.responseTimeouts.delete(priceRequestId)

    this.doPriceResponse(priceRequestId)
  }
}
```

Snippet 4.5: Definition of `checkForDoPriceResponse()`

4.1.8 V-AFR-VUL-008: Potential race condition in completePriceRequest

Severity	Low	Commit	859396f
Type	Race Condition	Status	Open
File(s)	`priceRequest.repository.ts`		
Location(s)	completePriceRequest()		
Confirmed Fix At	N/A		

The `completePriceRequest()` function is used to save the final response to a price request. It may suffer from race conditions that result from several bugs in the code:

1. The `complete` field is not checked when retrieving the price request
2. The `findOneBy` and `save` actions are together meant to constitute an atomic read-modify-write operation. However, they are not encapsulated in a transaction.

```
export const completePriceRequest = async (id: number, finalPriceResponseId: number):
     Promise<PriceRequestModel | Error> => {
  try {
    const priceRequest = await priceRequestRepository.findOneBy({ id })

    if (!priceRequest) {
      return new Error('Price request not found')
    }

    priceRequest.finalPriceResponseId = finalPriceResponseId
    priceRequest.complete = true

    await priceRequestRepository.save(priceRequest)

    return priceRequest
  } catch (err: any) {
    return new Error(`Error completing price request: ${err.message}`)
  }
}
```

Snippet 4.6: Definition of `completePriceRequest()`

Impact Because the `findOneBy()` and `save()` are not performed in the same transaction, it is possible for V-AFR-VUL-009 to cause two concurrent calls to `completePriceRequest()` with the same request ID but different response IDs to complete successfully, leaving the database in an inconsistent state.

For example, suppose there are two concurrent calls `completePriceRequest(x, a)` and `completePriceRequest (x, b)` (which we will name "Call 1" and "Call 2" respectively). The following evaluation order is possible:

1. Call 1 looks up the price request: `priceRequestRepository.findOneBy({ id: x })`
2. Call 2 looks up the price request: `priceRequestRepository.findOneBy({ id: x })`
3. Call 2 saves response ID `b` to the database
4. Call 1 saves response ID `a` to the database
5. Call 1 returns the price request with response ID `a`
6. Call 2 returns the price request with response ID `b`

The above example exhibits the following undesirable behavior:

- Response `a` is saved as the "final" price response, even if response `b` has a better price.
- At step (3), the price request has already been marked as complete. However, step (4) will attempt to "complete" the response again.
- In `WebSocketService.doPriceResponse()`, The same HTTP response will be "completed" twice, which can result in errors or other undesirable behavior.

Since these requests are rarely modified in the current implementation (i.e. only when the request is completed), the likelihood and impact of this race condition is low. However if there are any additional `priceRequest` modifications added in the WebSocket server in the future, this could become a more severe issue.

Recommendation

- Perform the lookup and update in the same transaction as shown in the `typeorm` documentation: `https://orkhan.gitbook.io/typeorm/docs/transactions`
- Specify `complete: false` in the call to `findOneBy`, so that only incomplete price requests are found.

4.1.9 V-AFR-VUL-009: Timeout race condition in doPriceResponse

Severity	Low	**Commit**	859396f
Type	Race Condition	**Status**	Open
File(s)	websocket.service.ts		
Location(s)	doPriceResponse()		
Confirmed Fix At	N/A		

It may be possible for two concurrent calls to be made to `doPriceResponse()` in the following situation:

1. (Call 1) The router receives a price response from a maker client in `receivePriceData()` and then invokes `checkForDoPriceResponse()`, which will then call `doPriceResponse()`.
2. (Call 2) Simultaneously, the price request times out and calls `doPriceResponse()`.

This is possible due to a combination of bugs in the code:

- The beginning of `doPriceResponse()` will check that the `priceRequestId` is contained in `priceRequests`. However, in `doPriceResponse()`, the entry is only deleted at the end of the function (see also: V-AFR-VUL-003). This means that if both Call 1 and Call 2 are currently at the beginning of `doPriceResponse()`, then they will both pass the containment check as long as either one of the calls yields to the other on the `await completePriceRequest(...)` line.

```
private async doPriceResponse(priceRequestId: number): Promise<void> {
  const priceRequestWithResObj = this.priceRequests.get(priceRequestId)
  if (!priceRequestWithResObj) return
  const { priceRequest, res: resObj } = priceRequestWithResObj
  const priceResponses = this.priceResponses[priceRequestId]
  if (!priceResponses) { /* ... */ return }
  const validPriceResponses = /* ... */
  if (validPriceResponses.length === 0) { /* ... */ return }
  const { priceResponse: maxAmountOutPriceReponse, quote: maxAmountOutQuote } = /*
    ... */
  await completePriceRequest(priceRequest.id, maxAmountOutPriceReponse.id)
  // ... code to send back price response ...

  this.priceRequests.delete(priceRequestId)
  delete this.priceResponses[priceRequestId]
}
```

Snippet 4.7: Definition of `doPriceResponse()`, with irrelevant portions commented out.

- To reach `doPriceResponse()`, Call 1 will have to pass through several checks in `receivePriceData()` involving `priceRequests` and `priceResponses`. However, these checks will be bypassed due to the above problem.
- There is no logic preventing Call 1 and Call 2 from concurrently calling `doPriceResponse()`, due to the missing check for already-completed price requests as described in V-AFR-VUL-008.

Impact A race condition in `doPriceResponse()` can lead to subtle bugs, including:

```
const priceRequestWithRes = this.priceRequests.get(priceRequestId)
if (!priceRequestWithRes) {
  console.log(‘No price request with id ${priceRequestId}‘)
  return
}
if (!data) {
  console.log(‘No data in message with id ${priceRequestId}‘)
  this.priceResponses[priceRequestId].push(/*..*/)
  this.checkForDoPriceResponse(priceRequestId)
  return
}
const { priceRequest } = priceRequestWithRes
if (priceRequest.amountIn !== data.amountIn || priceRequest.tokenIn !== data.tokenIn
    || priceRequest.tokenOut !== data.tokenOut) {
  console.log(‘Price request with id ${priceRequestId} is invalid‘)
  return
}
if (priceRequest.createdAt.getTime() + WEBSOCKET_RESPONSE_TIMEOUT_MS < new Date().
    getTime()) {
  console.log(‘Price request with id ${priceRequestId} is expired‘)
}
console.log(‘Add price to id ${priceRequestId}‘)
if (!this.priceResponses[priceRequestId]) {
  return
}
```

Snippet 4.8: Checks that can be bypassed in `receivePriceData()`

- Allowing V-AFR-VUL-008 to occur.
- If Call 2 (timeout) enters `doPriceResponse()` first, then the price computed by Call 2 may be different than the price computed by Call 1 (maker client response), because Call 1 will write to `priceResponses` in `receiveCallData()` .
- Suppose that Call 2 (timeout) occurs, that there are currently no valid maker responses saved in `priceResponses()`, and that Call 1 is currently returning from `createPriceResponse()` in `receivePriceData()`. Then due to V-AFR-VUL-003, the `doPriceResponse()` method will still be invoked in Call 1 even though the request has timed out already. However, Call 2 may have already completed the HTTP request with data, which can result in errors.

Recommendation

- Fix the following issues:
  - V-AFR-VUL-007
  - V-AFR-VUL-003
  - V-AFR-VUL-004.
- Use a mutex or another appropriate synchronization primitive to ensure that `doPriceResponse()` is executed atomically.
- Or instead of implementing synchronization manually, use the database to perform synchronization. Specifically, replace the maps such as `priceResponses` and `priceRequests` with appropriate calls to the database.

4.1.10 V-AFR-VUL-010: Potentially missing return statement

Severity	Warning	**Commit**	859396f
Type	Logic Error	**Status**	Open
File(s)	`websocket.service.ts`		
Location(s)	receivePriceData()		
Confirmed Fix At	N/A		

When `receivePriceData()` is called to handle a maker client's response to a price request, the function will first validate that the response matches the data in the price request. One of these validations checks that the price request is no longer expired; however, no action is taken if the price request is expired. Due to V-AFR-VUL-003, if the request's entry into `priceResponses` hasn't been deleted, then the next check will succeed and the rest of the function will execute.

```
if (priceRequest.amountIn !== data.amountIn || priceRequest.tokenIn !== data.tokenIn
    || priceRequest.tokenOut !== data.tokenOut) {
  console.log('Price request with id ${priceRequestId} is invalid')
  return
}
if (priceRequest.createdAt.getTime() + WEBSOCKET_RESPONSE_TIMEOUT_MS < new Date().
    getTime()) {
  console.log('Price request with id ${priceRequestId} is expired')
}
console.log('Add price to id ${priceRequestId}')
if (!this.priceResponses[priceRequestId]) {
  return
}
const priceResponse = await createPriceResponse(/* ... */)
if (priceResponse instanceof Error) {
  console.log('Error on create price response: ${priceResponse.message}')
  return
}
this.priceResponses[priceRequestId].push({
  priceResponse,
  quote: generateQuote(data),
})
this.checkForDoPriceResponse(priceRequestId)
```

Snippet 4.9: Relevant lines from `receivePriceData()`

Impact If a price request times out with no responses from maker clients, but then a maker sends a response for that price request, then it is possible to bypass the `if (!this.priceResponses [priceRequestId])` check. This would result in (1) the price response being inserted into the database and (2) a call to `checkForDoPriceResponse()`, even though these should not happen.

Recommendation Add a `return` statement after the statement that logs that the price request is expired.

4.1.11 V-AFR-VUL-011: Typo in price_pesponse table name

Severity	Info	**Commit**	859396f
Type	Maintainability	**Status**	Open
File(s)	`start.migration.ts`, `priceResponse.model.ts`		
Location(s)	StartMigration, PriceResponseModel		
Confirmed Fix At	N/A		

The price response objects sent from the maker clients to the router are stored in a database table named `price_pesponse`. It is likely that the table is actually meant to be titled `price_response`.

```
public async up(queryRunner: QueryRunner): Promise<void> {
  await queryRunner.query(/* ... */)
  await queryRunner.query(
    `CREATE TABLE "price_pesponse" /* ... */`,
  )
  await queryRunner.query(/* ... */)
  await queryRunner.query(/* ... */)
  await queryRunner.query(
    `ALTER TABLE "price_pesponse" /* ... */`,
  )
  await queryRunner.query(
    `ALTER TABLE "price_pesponse" /* ... */`,
  )
}
```

Snippet 4.10: The `StartMigration.up()` function that creates the database table. The table is also misnamed in the `StartMigration.down()` function.

```
@Entity({ name: 'price_pesponse' })
export class PriceResponseModel {
  // ...
}
```

Snippet 4.11: The `PriceResponseModel` ORM class representing a row in the `price_pesponse` table.

Impact The typo may lead to future developer confusion.

Recommendation Fix the table name in `StartMigration` and `PriceResponseModel` .

Glossary

Aleo A network for creating applications that use zero knowledge proofs. See `https://aleo.org/` for more . 1