// Smart Contract Security Assessment 06.07.2024 - 07.26.2024

RFX Contracts

RFX Exchange

HALBORN

RFX Contracts - RFX Exchange

Prepared by: HALBORN

Last Updated 09/05/2024

Date of Engagement by: June 7th, 2024 - July 26th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
20	0	2	2	4	12

TABLE OF CONTENTS

1. INTRODUCTION

RFX Exchange engaged Halborn to conduct a security assessment on their smart contracts beginning on June 7th, 2024 and ending on July 26th, 2024. The security assessment was scoped to smart contracts in the GitHub repository provided to the Halborn team. Commit hashes and further details can be found in the Scope section of this report.

2. ASSESSMENT SUMMARY

Halborn assigned two full-time security engineers to review the security of the smart contracts in scope. The engineers are blockchain and smart contract security experts with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were mostly addressed by the RFX team. The main ones were the following:

- Include the current value of the execution fee when topping up subaccounts during order updates.
- Update the funding and borrowing state during deposits / withdrawals before modifying the pool amount.
- Validate that the oracle prices provided during the execution of the operation fall within an acceptable range.
- Remove the deposit cap validation if the price impact is greater than zero.

3. TEST APPROACH AND METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the contracts' solidity code and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walk-through.
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic-related vulnerability classes.
- Local testing with custom scripts (Foundry).
- Fork testing against main networks (Foundry).
- Static analysis of security for scoped contract, and imported functions.

4. CAVEATS

This assessment was specifically focused on the contracts defined within the scope, and therefore, the findings are limited to those contracts. However, there are numerous other contracts that were not included in this assessment but are part of the overall system.

Issues or vulnerabilities in these out-of-scope contracts could potentially affect the security of the protocol. Consequently, it is recommended to complement this assessment with a comprehensive review of the entire codebase to identify any additional vulnerabilities or issues that may not have been covered, ensuring a more robust security posture for the protocol.

5. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

5.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

5.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

5.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

6. SCOPE

FILES AND REPOSITORY

(a) Repository: rfx-contracts

(b) Assessed Commit ID: 7b547f2

(c) Items in scope:

- contracts/adl/AdlUtils.sol
- contracts/data/DataStore.sol
- contracts/data/Keys.sol
- contracts/deposit/DepositVault.sol
- contracts/deposit/ExecuteDepositUtils.sol
- contracts/error/Errors.sol
- contracts/event/EventEmitter.sol
- contracts/exchange/AdlHandler.sol
- contracts/exchange/BaseOrderHandler.sol
- contracts/exchange/DepositHandler.sol
- contracts/exchange/LiquidationHandler.sol
- contracts/exchange/OrderHandler.sol
- contracts/exchange/WithdrawalHandler.sol
- contracts/fee/FeeHandler.sol
- contracts/market/Market.sol
- contracts/market/MarketFactory.sol
- contracts/market/MarketToken.sol
- contracts/oracle/Oracle.sol
- contracts/oracle/OracleModule.sol
- contracts/oracle/OracleStore.sol
- contracts/oracle/OracleUtils.sol
- contracts/order/DecreaseOrderUtils.sol
- contracts/order/IncreaseOrderUtils.sol
- contracts/order/Order.sol
- contracts/order/OrderVault.sol
- contracts/order/SwapOrderUtils.sol
- contracts/position/Position.sol
- contracts/price/Price.sol
- contracts/referral/IReferralStorage.sol
- contracts/role/Role.sol
- contracts/role/RoleModule.sol
- contracts/role/RoleStore.sol
- contracts/router/Router.sol
- contracts/router/SubaccountRouter.sol
- contracts/swap/SwapHandler.sol
- contracts/swap/SwapUtils.sol

- contracts/utils/Precision.sol
- contracts/withdrawal/ExecuteWithdrawalUtils.sol
- contracts/withdrawal/Withdrawal.sol

Out-of-Scope: Files not included in the scope, third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- https://github.com/relative-finance/rfx-contracts/pull/50/commits/86f5e9398e3e2fbfb7ff21ed8dbd9fbdaa176170
- https://github.com/relative-finance/rfx-contracts/tree/develop-v2
- https://github.com/relative-finance/rfx-contracts/pull/50/commits/a4b457671fd048a706db420e790cc73c2e4ce85c
- https://github.com/relative-finance/rfx-contracts/pull/50/commits/351992fc4cf1e2eca6cfa96b07c3b0636e9654bb
- https://github.com/relative-finance/rfx-contracts/pull/50/commits/166bccf8beda744ea57dbae1c9ec86ec6511c10f
- https://github.com/relative-finance/rfx-contracts/pull/50/commits/2684b1511e0c85f621bfdb2d9d987b799ff9e52f
- https://github.com/relative-finance/rfx-contracts/pull/50/commits/ceda2c6def9ecc2fa5cb91d0b9ba67cf3b9ffb99
- https://github.com/relative-finance/rfx-contracts/pull/49/commits/e715715e565763496 01cf24fee317bffd7d8ca60

Out-of-Scope: New features/implementations after the remediation commit IDs.

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	2	4	12

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
EXECUTION FEE MIGHT NOT BE REFUNDED TO SUBACCOUNTS WHEN UPDATING ORDERS	HIGH	SOLVED - 09/01/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
NON-UPDATED BORROWING FACTOR LEADS TO FEE MISCALCULATIONS	HIGH	SOLVED - 08/21/2024
LACK OF PRICE VALIDATION ALLOWS OPERATIONS AT IMPROPER PRICES	MEDIUM	SOLVED - 08/21/2024
DEPOSIT CAP VALIDATION COULD CAUSE UNNECESSARY REVERTS	MEDIUM	SOLVED - 08/21/2024
FLAWED PAYMENT LOGIC	LOW	SOLVED - 08/21/2024
ORACLE FEED REPORTS MISSING FEED ID	LOW	SOLVED - 08/21/2024
ROLE SET IS NOT CORRECTLY UPDATED WHEN REVOKING ONE OF THE ROLES	LOW	SOLVED - 09/01/2024
UNTIMELY BALANCE CHECKS WHEN TOPPING UP SUBACCOUNTS	LOW	SOLVED - 09/01/2024
EMPTY EVENT LOG DATA IS RETURNED WHEN PROCESSING ORDER INCREASES	INFORMATIONAL	SOLVED - 09/01/2024
LACK OF ZERO ADDRESS CHECK	INFORMATIONAL	ACKNOWLEDGED - 09/01/2024
POTENTIAL VARIABLE DECLARATION MIGHT GO UNUSED	INFORMATIONAL	SOLVED - 09/01/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DUPLICATED FUNCTIONS	INFORMATIONAL	SOLVED - 09/01/2024
OPTIMIZATION IN CONDITIONAL LOGIC	INFORMATIONAL	SOLVED - 09/01/2024
OPTIMIZATIONS IN LOOPS	INFORMATIONAL	ACKNOWLEDGED - 09/01/2024
UNLOCKED PRAGMA	INFORMATIONAL	ACKNOWLEDGED - 09/01/2024
UNUSED COMPONENTS IN THE CODEBASE	INFORMATIONAL	PARTIALLY SOLVED - 08/21/2024
INCORRECT OR MISSING NATSPEC COMMENTS	INFORMATIONAL	PARTIALLY SOLVED - 08/21/2024
TYPO IN COMMENT	INFORMATIONAL	SOLVED - 08/21/2024
INCORRECT COMMENTS	INFORMATIONAL	SOLVED - 08/21/2024
UNUSED IMPORT STATEMENTS	INFORMATIONAL	PARTIALLY SOLVED - 08/30/2024

8. FINDINGS & TECH DETAILS

8.1 EXECUTION FEE MIGHT NOT BE REFUNDED TO SUBACCOUNTS WHEN UPDATING ORDERS

// HIGH

Description

When updating an order through the **SubaccountRouter** contract, a subaccount needs to spend some gas and, sometimes, an `execution fee` that must be deposited in advance. The gas spent is refunded to the subaccount, but the execution fee is not. To set the context, it is important to review the **top-up mechanism** of the protocol when a subaccount creates or updates an order through the **SubaccountRouter** contract.

When a subaccount creates an order, it needs to spend gas + an `execution fee`. This total value is topped up (i.e., refunded) up to a certain cap to the subaccount after the order is created.

On the other hand, when a subaccount updates an order, **only the spent gas is refunded, not the execution fee**. Sometimes, the subaccount doesn't need to add more execution fee, only to modify some parameters. However, in other scenarios is a must to add more execution fee to the orders, as seen in the following example:

1. An order is created.
2. A keeper executes the order, but it fails because it is unfulfillable. So, the `execution fee` for the order is now set to **0** and the order is frozen.
3. To unfreeze the order, the subaccount needs to update it. Since the current the current `execution fee` is **0** (see previous step), **the subaccount must have previously deposited an execution fee** to incentivize keepers to execute it.
4. Finally, the gas spent is topped up (i.e., refunded) to the subaccount, but **not the execution fee**.

The `execution fee` that subaccounts need to deposit in advance when updating orders is not considered when topping them up. As a result, the execution fee spent might not be correctly refunded to the subaccounts, which economically harms the subaccounts and causes them to lose tokens.

Code Location

The described issue happens because the `SubaccountRouter.updateOrder` function calls `_autoTopUpSubaccount` using **0** as execution fee, instead of relying on the amount of wrapped native tokens previously deposited by the subaccount:

```
152 _autoTopUpSubaccount(
153     order.account(), // account
154     msg.sender, // subaccount
155     startingGas, // startingGas
```

```
    0 // executionFee
);
```

It means that when `nativeTokensUsed` is calculated, it will consider `executionFee` as **0**, so the amount transferred to the `subaccount` could be missing the `executionFee`:

```
// cap the top up amount to the amount of native tokens used
uint256 nativeTokensUsed = (startingGas - gasleft()) * tx.gasprice + ex
if (nativeTokensUsed < amount) { amount = nativeTokensUsed; }

router.pluginTransfer(
  address(wnt), // token
  account, // account
  address(this), // receiver
  amount // amount
);

TokenUtils.withdrawAndSendNativeToken(
  dataStore,
  address(wnt),
  subaccount,
  amount
);
```

However, the `OrderHandler.updateOrder` function (which is called at some point during the order update operation) adds a new `executionFee` equivalent to the wrapped native tokens previously sent, which can be **greater than zero**, especially if the order was previously frozen (which set the `executionFee` to 0) and needs that someone adds a new `executionFee` **greater than zero** to make it work:

```
// allow topping up of executionFee as frozen orders
// will have their executionFee reduced
address wnt = TokenUtils.wnt(dataStore);
uint256 receivedWnt = orderVault.recordTransferIn(wnt);
order.setExecutionFee(order.executionFee() + receivedWnt);
```

Proof of Concept

The proof of concept was directly developed in the already existent `test/guardian/testFrozenOrder.ts` test file, which only needs a couple of minor updates to make the PoC work correctly.

Updates in the test file:

Add the following imports:

```
import { createAccount } from "../../utils/account";
import * as keys from "../../utils/keys";
```

Finally, include `subaccountRouter` and `orderVault` variables in the test file

Step-by-step explanation of the PoC:

1. The `subaccountRouter` is set. It is configured that the subaccount's top-up amount is **0.5 ETH**, which is the cap when refunding to subaccounts.
2. An order is created with **0.3 ETH** as `execution fee`.
3. The order is executed, but it fails because it is unfulfillable. So, the `execution fee` for the order is now set to **0** and the order is frozen.
4. The subaccount **updates the order for the first time** without depositing wrapped native tokens. As expected, the order is unfrozen but the `execute fee` is still **0**. Then, after refunding, the subaccount's balance has been reduced by only **0.000641938502995713 ETH** (partial gas expenses).
5. Finally, the subaccount **updates the order for the second time**, but now depositing **0.3 ETH** as `execution fee`. As expected, the `execute fee` is now **0.3 ETH**. However, after refunding, the subaccount's balance has now been reduced by **0.000667564503115301 ETH** (partial gas expenses), but also by the `execution fee` (**0.3 ETH**).

```
  it("Execution fee is not refunded to subaccount when updating order", asy

    const subaccount = createAccount();

    // Setting subaccountRouter
    await subaccountRouter
    .connect(user1)
    .multicall(
      [
        subaccountRouter.interface.encodeFunctionData("sendNativeToken", [su
        subaccountRouter.interface.encodeFunctionData("addSubaccount", [sub
        subaccountRouter.interface.encodeFunctionData("setMaxAllowedSubaccou
          subaccount.address,
          keys.SUBACCOUNT_ORDER_ACTION,
          20,
        ]),

        /**** STEP 1: SETTING THE SUBACCOUNT TOP-UP AMOUNT ****/
        subaccountRouter.interface.encodeFunctionData("setSubaccountAutoTopU
          subaccount.address,
          expandDecimals(5, 17), // Top up to 0.5 ETH
        ]),
      ],
```

```
  { value: expandDecimals(1, 18) }
);


/**** STEP 2: A LIMIT INCREASE ORDER IS CREATED ****/
await createOrder(fixture, {
  account: user1,
  market: ethUsdMarket,
  initialCollateralToken: wnt,
  initialCollateralDeltaAmount: expandDecimals(1, 18),
  sizeDeltaUsd: decimalToFloat(10_000),
  acceptablePrice: expandDecimals(5100, 12),
  triggerPrice: expandDecimals(5000, 12),
  orderType: OrderType.LimitIncrease,
  isLong: true,
  executionFee: expandDecimals(3, 17),
});

// Limit order go to future block
await mine(10);

const orderKeys = await getOrderKeys(dataStore, 0, 1);

const orderBeforeExecuted = await reader.getOrder(dataStore.address, or

// Check that the order's execution fee is 3e17, i.e.: 0.3 ETH
expect(orderBeforeExecuted.numbers.executionFee).eq(expandDecimals(3, 1


/**** STEP 3: THE ORDER IS EXECUTED AND BECOMES FROZEN ****/
await executeOrder(fixture, {
  tokens: [wnt.address, usdc.address],
  minPrices: [expandDecimals(5000, 4), expandDecimals(1, 6)],
  maxPrices: [expandDecimals(5000, 4), expandDecimals(1, 6)],
  tokenOracleTypes: [TOKEN_ORACLE_TYPES.DEFAULT, TOKEN_ORACLE_TYPES.DEF
  precisions: [8, 18],
  expectedFrozenReason: "InsufficientReserve",
});

const orderPreUpdate = await reader.getOrder(dataStore.address, orderKe

// Check that the order's execution fee is now 0
expect(orderPreUpdate.numbers.executionFee).eq(0);

// Check that the order has been frozen
expect(orderPreUpdate.flags.isFrozen).eq(true);
```

```
// ETH balance for subaccount before updating order
const ethBalancePreUpdate = await ethers.provider.getBalance(subaccount


/**** STEP 4: SUBACCOUNT UPDATES THE ORDER FOR THE FIRST TIME ****/
// subaccount updates the order to be filled at a different price point
await subaccountRouter.connect(subaccount).updateOrder(
  orderKeys[0],
  decimalToFloat(10_000), // sizeDeltaUsd
  expandDecimals(5200, 12), // acceptablePrice
  expandDecimals(5200, 12), // triggerPrice
  expandDecimals(52000, 6) // minOutputAmount
);

const orderAfterFirstUpdate = await reader.getOrder(dataStore.address,

// Check that the order's execution fee is still 0
expect(orderAfterFirstUpdate.numbers.executionFee).eq(0);

// Check that the order has been unfrozen
expect(orderAfterFirstUpdate.flags.isFrozen).eq(false);

// ETH balance for subaccount after first update
const ethBalanceAfterFistUpdate = await ethers.provider.getBalance(suba

// Gas spent (after top-up) in the first update: 0.000641938502995713 E
const EthSpentFirstUpdate = ethers.BigNumber.from("641938502995713");

// The difference between the previous and current balance is due to ga
expect(ethBalancePreUpdate.sub(ethBalanceAfterFistUpdate)).eq(EthSpentF


/**** STEP 5: SUBACCOUNT UPDATES THE ORDER FOR THE SECOND TIME ****/
await subaccountRouter
  .connect(subaccount)
  .multicall(
    [
      subaccountRouter.interface.encodeFunctionData("sendWnt", [orderVa
      subaccountRouter.interface.encodeFunctionData("updateOrder",
         [
          orderKeys[0],
          decimalToFloat(10_000), // sizeDeltaUsd
          expandDecimals(5200, 12), // acceptablePrice
          expandDecimals(5200, 12), // triggerPrice
          expandDecimals(52000, 6) // minOutputAmount
```

```
            ]),
        ],
        { value: expandDecimals(3, 17) }
      );

    const orderAfterSecondUpdate = await reader.getOrder(dataStore.address,

    // Check that the order's execution fee is now 3e17 again, i.e.: 0.3 ETH
    expect(orderAfterSecondUpdate.numbers.executionFee).eq(expandDecimals(3

    // ETH balance for subaccount after second update
    const ethBalanceAfterSecondUpdate = await ethers.provider.getBalance(sub

    // Gas spent (after top-up) in the second update: 0.000667564503115301 E
    const EthSpentSecondtUpdate = ethers.BigNumber.from("667564503115301");

    // The difference between the previous and current balance is due to exe
    // TL;DR: This difference shows that the execution fee is not refunded t
    expect(ethBalanceAfterFistUpdate.sub(ethBalanceAfterSecondUpdate)).eq(o

  });
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:M/Y:N (8.8)

Recommendation

Include the current value of the execution fee when topping up subaccounts during order updates.

Remediation

SOLVED: The **RFX team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/relative-finance/rfx-contracts/pull/50/commits/86f5e9398e3e2fbfb7ff21ed8dbd9fbdaa176170

8.2 NON-UPDATED BORROWING FACTOR LEADS TO FEE MISCALCULATIONS

// HIGH

Description

When a position is updated, the `updateFundingAndBorrowingState` function is triggered to adjust the `CUMULATIVE_BORROWING_FACTOR`. This factor is updated based on the latest borrowing rate (i.e.: borrowing factor per second) and the elapsed time since the last update. The borrowing rate itself depends on the percentage of the pool's liquidity that is currently borrowed: the higher the borrowed percentage, the higher the rate. By updating the `CUMULATIVE_BORROWING_FACTOR` before making any state changes that could influence the rate, the system ensures that borrowing fees are calculated accurately.

However, this process is not consistently applied throughout the code. When users withdraw or deposit funds, they alter the borrowing rate: withdrawals increase it, while deposits decrease it. Since the `CUMULATIVE_BORROWING_FACTOR` is not updated during a deposit or withdrawal, the next time it is updated (e.g.: when increasing / decreasing a position), the borrowing fees will be miscalculated because the rate will use the new value instead of the actual value that was present during the elapsed time. The described situation can result in excessive fees being charged if a withdrawal occurs, or insufficient fees being charged if a deposit occurs. The affected functions are the following:

- `ExecuteDepositUtils.executeDeposit`
- `ExecuteWithdrawalUtils.executeWithdrawal`

It is worth noting that large deposits or withdrawals can significantly amplify this fee inaccuracy. Excessive fees can unexpectedly push near-liquidateable positions into liquidation. Furthermore, this incremental rise in borrowing fees creates arbitrage opportunities, allowing attackers to exploit the inaccurate fee adjustments through well-timed orders and deposits.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:M (8.8)

Recommendation

Update the funding and borrowing state in the `executeDeposit` and `executeWithdrawal` functions before modifying the pool amount.

Remediation

SOLVED: The **RFX team** solved the issue when refactoring the codebase to v2.1.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.3 LACK OF PRICE VALIDATION ALLOWS OPERATIONS AT IMPROPER PRICES

// MEDIUM

Description

When creating a deposit, withdrawal, or order, the user's funds are transferred into a vault. A key referencing the requested operation is attached to the corresponding event upon creation. Once this event is detected, a keeper is triggered. The order keeper can then execute the requested operation by providing the necessary request key and oracle prices.
While prices are validated to ensure they are appropriate at the time of execution, they are not validated to ensure they fall within an acceptable range compared to the time of request creation. As a result, if there is a significant time gap between the request creation and execution, the operation may be executed with inappropriate prices. This scenario can potentially harm both users and disrupt the market's balance.

Code Location

Let's take the example of deposits: In order to execute a deposit, an order keeper calls the `executeDeposit` function within the `exchange/DepositHandler` contract:

```
94  function executeDeposit(
95      bytes32 key,
96      OracleUtils.SetPricesParams calldata oracleParams
97  ) external payable
98      globalNonReentrant
99      onlyOrderKeeper
100     withOraclePrices(oracle, dataStore, eventEmitter, oracleParams)
101 {
102 ...
103 }
```

The `OracleModule.withOraclePrices` modifier is implemented as follows: it sets the oracle prices using the provided prices before executing the calling function, and subsequently clears all the prices:

```
26  modifier withOraclePrices(
27      Oracle oracle,
28      DataStore dataStore,
29      EventEmitter eventEmitter,
30      OracleUtils.SetPricesParams memory params
31  ) {
32      oracle.setPrices{value: msg.value}(dataStore, eventEmitter, params);
33      _;
34      oracle.clearAllPrices();
35  }
```

The `Oracle.setPrices` function has the below implementation:

```
199 function setPrices(
200     DataStore dataStore,
201     EventEmitter eventEmitter,
202     OracleUtils.SetPricesParams memory params
203 ) external payable onlyController {
204     if (tokensWithPrices.length() != 0) {
205         revert Errors.NonEmptyTokensWithPrices(tokensWithPrices.length(
206     }
207 
208     _setPricesFromRealtimeFeeds(dataStore, eventEmitter, params);
209 }
```

The `Oracle._setPricesFromRealtimeFeeds` function has the below implementation:

```
396 function _setPricesFromRealtimeFeeds(
397     DataStore dataStore,
398     EventEmitter eventEmitter,
399     OracleUtils.SetPricesParams memory params
400 ) internal returns (OracleUtils.RealtimeFeedReport[] memory) {
401     OracleUtils.RealtimeFeedReport[] memory reports = _validateRealtime
402         dataStore,
403         params.realtimeFeedTokens,
404         params.realtimeFeedData
405     );
406     ....
407 }
```

The `Oracle._validateRealtimeFeeds` function has the below implementation:

```
303 function _validateRealtimeFeeds(
304         DataStore dataStore,
305         address[] memory realtimeFeedTokens,
306         bytes[] memory realtimeFeedData
307     ) internal returns (OracleUtils.RealtimeFeedReport[] memory) {
308         ...
309         uint256 dataCounter;
310         for (uint256 i; i < realtimeFeedTokens.length; i++) {
311             ...
312             uint fee = pyth.getUpdateFee(feedData);
313 
314             /// @dev Reverts if the transferred fee is not sufficient c
315             /// no update for any of the given `priceIds` within the gi
316             uint64 minPublishTime = uint64(Chain.currentTimestamp() - m
317 
```

```
318             uint64 maxPublishTime = uint64(Chain.currentTimestamp() + m
319             PythStructs.PriceFeed[] memory priceFeeds = pyth.parsePrice
320             ...
321         }
322         ...
        }
```

The highlighted part will ensure that the prices provided to `pyth.parsePriceFeedUpdates` must fall within the specified `minPublishTime` and `maxPublishTime`. These timestamps are derived from the current block timestamp, thereby enforcing that the prices are within an acceptable range relative to the time of execution.

Going back to the `exchange/DepositHandler.executeDeposit` function, it later calls the `_executeDeposit` internal function, which calls the `ExecuteDepositUtils.executeDeposit`function, which also fails to confirm that the provided prices align with an acceptable timeframe relative to the deposit creation time:

```
104 function executeDeposit(ExecuteDepositParams memory params, Deposit.Pro
105         // 63/64 gas is forwarded to external calls, reduce the startin
106         params.startingGas -= gasleft() / 63;
107 
108         DepositStoreUtils.remove(params.dataStore, params.key, deposit.
109 
110         ExecuteDepositCache memory cache;
111 
112         if (deposit.account() == address(0)) {
113             revert Errors.EmptyDeposit();
114         }
115 
116         Market.Props memory market = MarketUtils.getEnabledMarket(param
117 
118         _validateFirstDeposit(params, deposit, market);
119 
120         MarketUtils.MarketPrices memory prices = MarketUtils.getMarketP
121 
122         MarketUtils.distributePositionImpactPool(
123             params.dataStore,
124             params.eventEmitter,
125             market.marketToken
126         );
127         ...
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:H/D:H/Y:N (6.3)

Recommendation

Consider validating that the oracle prices provided during the execution of the operation fall within an acceptable range relative to the prices at the time the operation was requested.

Remediation

SOLVED: The **RFX team** solved the issue when refactoring the codebase to v2.1.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.4 DEPOSIT CAP VALIDATION COULD CAUSE UNNECESSARY REVERTS

// MEDIUM

Description

The `_executeDeposit` function in the `ExecuteDepositUtils` library calls the `MarketUtils.validatePoolAmountForDeposit` function if `priceImpactUsd` is greater than zero. However, if the pool's long token is near its deposit cap while the short token is not, depositing short tokens can lead to a positive price impact. This price impact might cause the long token's deposit cap to be exceeded and, as a consequence, the operation could unnecessarily revert.

In such scenarios, it is preferable to allow the pool to be rebalanced even if it results in surpassing the deposit cap for the long token.

Code Location

A call to the `MarketUtils.validatePoolAmountForDeposit` function when `priceImpactUsd` is greater than zero could cause an unnecessary revert:

```
370 if (_params.priceImpactUsd > 0) {
371   // when there is a positive price impact factor,
372   // tokens from the swap impact pool are used to mint additional marke
373   // for example, if 50,000 USDC is deposited and there is a positive p
374   // an additional 0.005 ETH may be used to mint market tokens
375   // the swap impact pool is decreased by the used amount
376   //
377   // priceImpactUsd is calculated based on pricing assuming only deposi
378   // was added to the pool
379   // since impactAmount of tokenOut is added to the pool here, the calc
380   // the price impact would not be entirely accurate
381   //
382   // it is possible that the addition of the positive impact amount of
383   // could increase the imbalance of the pool, for most cases this shou
384   // change compared to the improvement of balance from the actual depo
385   int256 positiveImpactAmount = MarketUtils.applySwapImpactWithCap(
386     params.dataStore,
387     params.eventEmitter,
388     _params.market.marketToken,
389     _params.tokenOut,
390     _params.tokenOutPrice,
391     _params.priceImpactUsd
392   );
393
394   // calculate the usd amount using positiveImpactAmount since it may
```

```
395     // be capped by the max available amount in the impact pool
396     // use tokenOutPrice.max to get the USD value since the positiveImpac
397     // was calculated using a USD value divided by tokenOutPrice.max
398     //
399     // for the initial deposit, the pool value and token supply would be
400     // so the market token price is treated as 1 USD
401     //
402     // it is possible for the pool value to be more than zero and the tok
403     // to be zero, in that case, the market token price is also treated c
404     mintAmount += MarketUtils.usdToMarketTokenAmount(
405       positiveImpactAmount.toUint256() * _params.tokenOutPrice.max,
406       poolValue,
407       marketTokensSupply
408     );
409
410     // deposit the token out, that was withdrawn from the impact pool, to
411     MarketUtils.applyDeltaToPoolAmount(
412       params.dataStore,
413       params.eventEmitter,
414       _params.market,
415       _params.tokenOut,
416       positiveImpactAmount
417     );
418
419     MarketUtils.validatePoolAmountForDeposit(
420       params.dataStore,
421       _params.market,
422       _params.tokenOut
423     );
424
425     MarketUtils.validatePoolAmount(
426       params.dataStore,
427       _params.market,
428       _params.tokenOut
429     );
430   }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (5.0)

Recommendation

It is recommended to remove the call to the `MarketUtils.validatePoolAmountForDeposit` function if `priceImpactUsd` is greater than zero.

Remediation

SOLVED: The **RFX team** solved the issue when refactoring the codebase to v2.1.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.5 FLAWED PAYMENT LOGIC

// LOW

Description

The **Oracle** contract relies on the **Pyth Network** for querying price feeds using `pyth.parsePriceFeedUpdates`, which necessitates a pre-calculated fee. However, the current payment logic has several issues:

1. The `_validateRealtimeFeeds` function does not validate whether the caller has provided sufficient ETH to cover the fees when calculating the fee. This oversight allows the caller to utilize contract funds and bypass the payment requirement.

2. The `validateRealtimeFeeds` function, which incurs fees by calling `_validateRealtimeFeeds`, lacks the `payable` modifier. Consequently, it does not allow callers to attach the necessary funds to cover fees. If the contract has insufficient funds, this function becomes temporarily disabled. Since the `Oracle` contract lacks a payable fallback or receive function, funds can only be transferred by either self-destructing from another contract to transfer funds or by attaching a large amount of ETH when calling `setPrices`, the only available payable function.

Code Location

The `_validateRealtimeFeeds` function does not validate whether the caller has provided sufficient ETH to cover the fees:

```
303 function _validateRealtimeFeeds(
304         DataStore dataStore,
305         address[] memory realtimeFeedTokens,
306         bytes[] memory realtimeFeedData
307     ) internal returns (OracleUtils.RealtimeFeedReport[] memory) {
308         ...
309         for (uint256 i; i < realtimeFeedTokens.length; i++) {
310             ...
311             uint fee = pyth.getUpdateFee(feedData);
312 
313             /// @dev Reverts if the transferred fee is not sufficient c
314             /// no update for any of the given `priceIds` within the gi
315             uint64 minPublishTime = uint64(Chain.currentTimestamp() - m
316             uint64 maxPublishTime = uint64(Chain.currentTimestamp() + m
317             PythStructs.PriceFeed[] memory priceFeeds = pyth.parsePrice
318             ...
319         }
320         ...
321     }
```

The `validateRealtimeFeeds` function calls the `_validateRealtimeFeeds`function. However, it lacks the `payable` modifier:

```
295 function validateRealtimeFeeds(
296     DataStore dataStore,
297     address[] memory realtimeFeedTokens,
298     bytes[] memory realtimeFeedData
299 ) external onlyController returns (OracleUtils.RealtimeFeedReport[] mem
300     return _validateRealtimeFeeds(dataStore, realtimeFeedTokens, realti
301 }
```

BVSS

AO:A/AC:L/AX:L/R:P/S:C/C:N/A:M/I:N/D:M/Y:N (3.9)

Recommendation

It is recommended to implement the following changes:

- Validate that the caller has attached sufficient ETH to cover the fee and refund any excess amount within `_validateRealtimeFeeds`.
- Add the `payable` keyword to the `validateRealtimeFeeds` function to allow callers to attach ETH for covering the fees.

Remediation

SOLVED: The **RFX team** solved the issue when refactoring the codebase to v2.1.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.6 ORACLE FEED REPORTS MISSING FEED ID

// LOW

Description

In the `_validateRealtimeFeeds` function of the **Oracle** contract, price feeds are queried for each token using the respective feed ID to generate real-time feed reports, of type `OracleUtils.RealtimeFeedReport`, containing price details.

However, currently, the `feedId` field within each generated report is not being set. Consequently, each report lacks the necessary feed ID information that was used to query the underlying price details. This omission can lead to ambiguity or incorrect interpretation of the price data retrieved.

Code Location

Below is the definition of the `OracleUtils.RealtimeFeedReport` structure:

```
75  struct RealtimeFeedReport {
76      // The feed ID the report has data for
77      bytes32 feedId;
78      // Min price
79      uint256 minPrice;
80      // Max price
81      uint256 maxPrice;
82      // Price exponent
83      int32 expo;
84      // Unix timestamp describing when the price was published
85      uint publishTime;
86  }
```

In the `_validateRealtimeFeeds` function, the `feedId` field is not set for each report:

```
349 OracleUtils.RealtimeFeedReport memory report;
350     if(baseFeedId != bytes32(0)) {
351         report = OracleUtils.getQuoteBaseReport(priceFeeds[0].price, pr
352     } else {
353         report.publishTime = priceFeeds[0].price.publishTime;
354         report.expo = priceFeeds[0].price.expo;
355         report.minPrice = uint256(uint64(priceFeeds[0].price.price) - p
356         report.maxPrice = uint256(uint64(priceFeeds[0].price.price) + p
357     }
358     if (report.publishTime + maxPriceAge < Chain.currentTimestamp()) {
359         revert Errors.RealtimeMaxPriceAgeExceeded(token, report.publish
360     }
```

```
361
362         reports[i] = report;
```

BVSS

AO:A/AC:L/AX:L/R:N/S:C/C:N/A:N/I:L/D:N/Y:N (3.1)

Recommendation

Consider setting each report's `feedId`:

```
report.feedId = feedId;
reports[i] = report;
```

Remediation

SOLVED: The **RFX team** solved the issue when refactoring the codebase to v2.1.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.7 ROLE SET IS NOT CORRECTLY UPDATED WHEN REVOKING ONE OF THE ROLES

// LOW

Description

The `_revokeRole` function in the `RoleStore` contract does not remove `roleKey` from the `role` set when there are no more members (i.e.: `roleMembers[roleKey].length()` is **0**). As a consequence, the number of roles stored in the contract calculated in the `getRoleCount` function could be distorted from the current value.

Code Location

The `_revokeRole` function does not remove `roleKey` from `role` set when there are no more members:

```
117  function _revokeRole(address account, bytes32 roleKey) internal {
118    roleMembers[roleKey].remove(account);
119    roleCache[account][roleKey] = false;
120
121    if (roleMembers[roleKey].length() == 0) {
122      if (roleKey == Role.ROLE_ADMIN) {
123        revert Errors.ThereMustBeAtLeastOneRoleAdmin();
124      }
125      if (roleKey == Role.TIMELOCK_MULTISIG) {
126        revert Errors.ThereMustBeAtLeastOneTimelockMultiSig();
127      }
128    }
129  }
```

BVSS

AO:A/AC:L/AX:L/R:P/S:U/C:N/A:N/I:M/D:N/Y:N (2.5)

Recommendation

It is recommended to remove keys from the `role` set when there are no more members.

Remediation

SOLVED: The **RFX team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/relative-finance/rfx-contracts/pull/50/commits/a4b457671fd048a706db420e790cc73c2e4ce85c

8.8 UNTIMELY BALANCE CHECKS WHEN TOPPING UP SUBACCOUNTS

// LOW

Description

The `_autoTopUpSubaccount` function in the `SubaccountRouter` contract verifies if the balance of wrapped native tokens for the `account` is sufficient before further processing. However, this validation is carried out **before capping** the top-up `amount` to the amount of native tokens used, which could result in some subaccounts not being correctly topped up.

Code Location

The `_autoTopUpSubaccount` function verifies the balance of wrapped native tokens for the `account` **before capping** the top-up `amount`:

```
198 function _autoTopUpSubaccount(address account, address subaccount, uint
199   uint256 amount = SubaccountUtils.getSubaccountAutoTopUpAmount(dataSto
200   if (amount == 0) {
201     return;
202   }
203
204   IERC20 wnt = IERC20(dataStore.getAddress(Keys.WNT));
205
206   if (wnt.allowance(account, address(router)) < amount) { return; }
207   if (wnt.balanceOf(account) < amount) { return; }
208
209   // cap the top up amount to the amount of native tokens used
210   uint256 nativeTokensUsed = (startingGas - gasleft()) * tx.gasprice +
211   if (nativeTokensUsed < amount) { amount = nativeTokensUsed; }
212
213   router.pluginTransfer(
214     address(wnt), // token
215     account, // account
216     address(this), // receiver
217     amount // amount
218   );
219
220   TokenUtils.withdrawAndSendNativeToken(
221     dataStore,
222     address(wnt),
223     subaccount,
224     amount
225   );
226
```

```
227     EventUtils.EventLogData memory eventData;
228
229     eventData.addressItems.initItems(2);
230     eventData.addressItems.setItem(0, "account", account);
231     eventData.addressItems.setItem(1, "subaccount", subaccount);
232
233     eventData.uintItems.initItems(1);
234     eventData.uintItems.setItem(0, "amount", amount);
235
236     eventEmitter.emitEventLog2(
237       "SubaccountAutoTopUp",
238       Cast.toBytes32(account),
239       Cast.toBytes32(subaccount),
240       eventData
241     );
242 }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (2.5)

Recommendation

It is recommended to verify the balance of wrapped native tokens for the `account` **after capping** the top-up `amount`.

Remediation

SOLVED: The **RFX team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/relative-finance/rfx-contracts/pull/50/commits/86f5e9398e3e2fbfb7ff21ed8dbd9fbdaa176170

8.9 EMPTY EVENT LOG DATA IS RETURNED WHEN PROCESSING ORDER INCREASES

// INFORMATIONAL

Description

When executing market orders, the `processOrder` function within each order type manages order execution and returns event log data. This data can later be utilized within the `afterOrderExecution` function inside the order's callback contract.
However, when executing order increases via `IncreaseOrderUtils.processOrder`, the function returns empty event log data, thereby preventing the callback from utilizing the execution results.

Code Location

The `IncreaseOrderUtils.processOrder` function returns an empty event log data after processing each order:

```
20  function processOrder(BaseOrderUtils.ExecuteOrderParams memory params)
21      ...
22
23      EventUtils.EventLogData memory eventData;
24      return eventData;
25  }
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:N/D:N/Y:N (1.7)

Recommendation

It is recommended to include the collateral token address and the corresponding increment amount used to increase the user's position in the event log data. These parameters are returned from the `SwapUtils.swap` call:

```
23  (address collateralToken, uint256 collateralIncrementAmount) = SwapUtil
24  ...
```

Remediation

SOLVED: The **RFX team** has solved this issue by including the collateral token address and the corresponding increment amount used to increase the user's position in the event log data.

Remediation Hash

https://github.com/relative-finance/rfx-contracts/pull/50/commits/351992fc4cf1e2eca6cfa96b07c3b0636e9654bb

8.10 LACK OF ZERO ADDRESS CHECK

// INFORMATIONAL

Description

Some functions in the codebase do not include a **zero address check** for their parameters. If one of those parameters is mistakenly set to zero, it could affect the correct operation of the protocol. The affected functions are the following:

- `DataStore.setAddress`
- `BaseOrderHandler.constructor`
- `DepositHandler.constructor`
- `FeeHandler.constructor`
- `MarketFactory.constructor`
- `OracleStore.constructor`
- `OracleStore.addSigner`
- `RoleModule.constructor`
- `SubaccountRouter.constructor`

BVSS

AO:A/AC:L/AX:H/R:P/S:U/C:N/A:N/I:M/D:N/Y:N (0.8)

Recommendation

It is recommended to add a zero address check in the functions mentioned above.

Remediation

ACKNOWLEDGED: The **RFX team** acknowledged this issue.

Remediation Comment

Acknowledged

8.11 POTENTIAL VARIABLE DECLARATION MIGHT GO UNUSED

// INFORMATIONAL

Description

In the `SwapUtils._swap` function, there is a risk that the declared `cache` variable might not be utilized if one of the subsequent validations fails. This could lead to unnecessary gas consumption.

Code Location

The `cache` variable declaration might remain unused if one of the subsequent validations fails:

```
184  function _swap(SwapParams memory params, _SwapParams memory _params) ir
185      SwapCache memory cache;
186
187      if (_params.tokenIn != _params.market.longToken && _params.tokenIn
188          revert Errors.InvalidTokenIn(_params.tokenIn, _params.market.ma
189      }
190
191      MarketUtils.validateSwapMarket(params.dataStore, _params.market);
192
193      cache.tokenOut = MarketUtils.getOppositeToken(_params.tokenIn, _par
```

Score

Impact:

Likelihood:

Recommendation

Consider declaring the `cache` variable immediately after the `MarketUtils.validateSwapMarket` check to ensure it is used in the intended context.

Remediation

SOLVED: The **RFX team** has solved this issue by implementing the recommended change.

Remediation Hash

https://github.com/relative-finance/rfx-contracts/pull/50/commits/166bccf8beda744ea57dbae1c9ec86ec6511c10f

8.12 DUPLICATED FUNCTIONS

// INFORMATIONAL

Description

Some functions in the codebase are duplicated (i.e.: they contain the same code logic), which could make it harder to apply future changes in the common logic of the functions and can even introduce potential bugs. The affected functions are the following:

- `DataStore.applyDeltaToUint` function has duplicated code with `DataStore.incrementUint`.
- `DataStore.applyDeltaToInt` function has duplicated code with `DataStore.incrementInt`.

This situation is not security-related, but mentioned in the report as part of the DRY (Don't Repeat Yourself) principle used as a best practice in software development to improve the maintainability of code during all phases of its lifecycle.

Score

Impact:

Likelihood:

Recommendation

It is recommended to update the codebase to call only one version of the duplicated functions and remove the other ones to avoid potential mistakes.

Remediation

SOLVED: The **RFX team** solved the issue in the specified commit id.

Remediation Hash

https://github.com/relative-finance/rfx-contracts/pull/50/commits/2684b1511e0c85f621bfdb2d9d987b799ff9e52f

8.13 OPTIMIZATION IN CONDITIONAL LOGIC

// INFORMATIONAL

Description

In the `ExecuteDepositUtils._executeDeposit` function, the deposit logic is segregated based on the price impact factor. The initial `if` clause manages scenarios with a positive price impact, whereas the subsequent `if` clause handles those with a negative price impact. Given that the first `if` clause checks `params.priceImpactUsd > 0`, it logically follows that the second `if` clause checks `params.priceImpactUsd < 0`. However, since the conditions are mutually exclusive, the second `if` clause will never evaluate to true if the first condition is satisfied.

Code Location

The `ExecuteDepositUtils._executeDeposit` function contains an inefficient `if` clause:

```
370 if (_params.priceImpactUsd > 0) {
371     ...
372 }
373
374 if (_params.priceImpactUsd < 0) {
375     ...
376 }
```

Score

Impact:

Likelihood:

Recommendation

It is recommended to use the `else if` construct as follows: `else if (_params.priceImpactUsd < 0)` to avoid redundant checks for a negative price impact after confirming a positive one.

Remediation

SOLVED: The **RFX team** has solved this issue by implementing the recommended change.

Remediation Hash

https://github.com/relative-finance/rfx-contracts/pull/50/commits/ceda2c6def9ecc2fa5cb91d0b9ba67cf3b9ffb99

8.14 OPTIMIZATIONS IN LOOPS

// INFORMATIONAL

Description

Within the existing `for` loops, the following optimization principles can be applied to reduce gas:

Increment loop counter in an unchecked block:

In Solidity, most `for` loops use a `uint256` counter variable that starts at 0 and increments by `1`. Checking for over/underflow in these increments is unnecessary because the variable will exhaust gas long before reaching the maximum capacity of `uint256`.

Use ++i instead of i++:

With `i++`, the value of `i` (its old value) is returned before incrementing `i` to its new value. This results in two values being stored on the stack, regardless of whether you intend to use both. In contrast, `++i` evaluates the increment operation first, updating `i` to its incremented value, and then returns this new value. This approach requires only one item to be stored on the stack.

Cache array lengths outside of loop:

When the length of an array is not cached outside of a loop, the Solidity compiler reads the length of the array during each iteration. For storage arrays, this leads to an additional `SLOAD` operation, incurring 100 extra gas for each iteration except the first. For memory arrays, an extra `MLOAD` operation is performed, adding 3 additional gas for each iteration except the first. Identifying loops where the length of a storage array is accessed in the loop condition without being modified inside the loop can highlight optimization opportunities.

The optimizations mentioned above can be implemented in the following cases:

- `contracts/fee/FeeHandler.sol#L38`
- `contracts/oracle/OracleModule.sol#L56`
- `contracts/oracle/Oracle.sol#L316, L407`
- `contracts/swap/SwapUtils.sol#L136, L165`

Score

Impact:

Likelihood:

Recommendation

It is recommended to implement the adequate `for` loop optimizations in the following way:

Change:

```
for (uint256 i; i < array.length; i++) {..}
```

To:

```
length = array.length
for (uint256 i; i < length;) {
  ...
  unchecked{++i;}
}
```

Remediation

ACKNOWLEDGED: The **RFX team** acknowledged this issue.

Remediation Comment

Acknowledged

8.15 UNLOCKED PRAGMA

// INFORMATIONAL

Description

Every Solidity file specifies in the header a version number of the format pragma solidity `(^)0.8.*`. The caret (`^`) before the version number implies an unlocked pragma, meaning that the compiler will use the specified version and above, hence the term "unlocked".

Contracts should be deployed using the same compiler version and flags that were used during their development and testing phases. Locking the pragma ensures consistency and prevents unintended deployment with a different pragma version. Using an outdated pragma version can introduce bugs that may adversely affect the contract system.

Code Location

All contracts within the codebase use an unlocked pragma:

```
pragma solidity ^0.8.0;
```

Score

Impact:

Likelihood:

Recommendation

Consider locking the pragma version in the smart contracts. It is not recommended to use a floating pragma in production.
For example: `pragma solidity 0.8.21;`

Remediation

ACKNOWLEDGED: The **RFX team** acknowledged this issue.

Remediation Comment

Acknowledged

8.16 UNUSED COMPONENTS IN THE CODEBASE

// INFORMATIONAL

Description

Within the codebase, the following components remain unused:

- `Errors.CouldNotSendNativeToken` error message
- `Errors.HasRealtimeFeedId` error message
- `Errors.InvalidRealtimeFeedId` error message
- `Errors.InvalidRealtimeBidAsk` error message
- `Errors.InvalidRealtimeBlockHash` error message
- `Errors.InvalidRealtimePrices` error message
- `Errors.MaxPriceAgeExceeded` error message
- `Errors.MinOracleSigners` error message
- `Errors.MaxOracleSigners` error message
- `Errors.BlockNumbersNotSorted` error message
- `Errors.MinPricesNotSorted error` message
- `Errors.MaxPricesNotSorted` error message
- `Errors.InvalidFeedPrice` error message
- `Errors.PriceFeedNotUpdated` error message
- `Errors.MaxSignerIndex` error message
- `Errors.InvalidOraclePrice` error message
- `Errors.InvalidSignerMinMaxPrice` error message
- `Errors.InvalidMedianMinMaxPrice` error message
- `Errors.EmptyPriceFeed` error message
- `Errors.UnsupportedOracleBlockNumberType` error message
- `oracleParams` argument in the `BaseOrderHandler._getExecuteOrderParams` function
- `Oracle.ValidatedPrice` struct
- `Oracle.SetPricesCache` struct
- `Oracle.getStablePrice` function
- `Oracle.getPriceFeedMultiplier` function
- `Oracle._getSalt` function

These elements are present within the codebase but are not currently utilized.

Score

Impact:

Likelihood:

Recommendation

It is recommended to remove unused components in the codebase to save gas.

Remediation

PARTIALLY SOLVED: The **RFX team** solved most of the issue when refactoring the codebase to v2.1. The only remaining unused component is the `Errors.BlockNumbersNotSorted` error message.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.17 INCORRECT OR MISSING NATSPEC COMMENTS

// INFORMATIONAL

Description

The following issues were identified in the NatSpec documentation:

OracleUtils.SetPricesParams struct**:**

- The `compactedOracleBlockNumbers` parameter is outdated and no longer exists.
- The `realtimeFeedTokens` and `realtimeFeedData` parameters lack NatSpec documentation.

OracleUtils.validateSigner function**:**

- The `minOracleBlockNumber` and `maxOracleBlockNumber` parameters are no longer used but are still documented.

ExecuteDepositUtils.ExecuteDepositParams struct**:**

- The `oracleBlockNumbers` parameter is deprecated and requires removal from NatSpec documentation.
- No NatSpec documentation exists for the `depositVault` parameter.

SwapUtils.SwapCache struct**:**

- The `priceImpactUsd` and `priceImpactAmount` parameters are undocumented.

Withdrawal.Addresses struct**:**

- The `longTokenSwapPath` and `shortTokenSwapPath` parameters lack NatSpec documentation.

Withdrawal.Flags struct**:**

- The `shouldUnwrapNativeToken` parameter's NatSpec comment is incomplete and needs revision:`"whether to unwrap the native token when"`.

ExecuteWithdrawalParams struct**:**

- The `minOracleBlockNumbers` and `maxOracleBlockNumbers` parameters are outdated and should be removed from NatSpec documentation.

ExecuteWithdrawalUtils.executeWithdrawal function**:**

- The `withdrawal` parameter lacks NatSpec documentation.

Order.Numbers struct**:**

- The `decreasePositionSwapType` parameter is not documented within the NatSpec comments.

Market.Props struct**:**

- The `data` parameter is documented in NatSpec comments but does not exist and should be removed.

Score

Impact:

Likelihood:

Recommendation

Consider updating the NatSpec comments by removing outdated parameters, documenting the missing ones, and completing unfinished comments.

Remediation

PARTIALLY SOLVED: The **RFX team** has partially solved this issue when refactoring the codebase to v2.1. However, there are still some NatSpec comments that are either missing or incorrect:
SwapUtils.SwapCache struct**:**

- The `priceImpactUsd` and `priceImpactAmount` parameters are undocumented.

Withdrawal.Addresses struct**:**

- The `longTokenSwapPath` and `shortTokenSwapPath` parameters lack NatSpec documentation.

Withdrawal.Flags struct**:**

- The `shouldUnwrapNativeToken` parameter's NatSpec comment is incomplete and needs revision:`"whether to unwrap the native token when"`.

ExecuteWithdrawalUtils.executeWithdrawal function**:**

- The `withdrawal` parameter lacks NatSpec documentation.

Order.Numbers struct**:**

- The `decreasePositionSwapType` parameter is not documented within the NatSpec comments.

Market.Props struct**:**

- The `data` parameter is documented in NatSpec comments but does not exist and should be removed.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.18 TYPO IN COMMENT

// INFORMATIONAL

Description

Within the NatSpec documentation for the `Oracle.getPriceFeedMultiplier` function, the word "**multiplier**" was misspelled as "**multipler**":

```
274 | // @return the price feed multipler
```

Score

Impact:

Likelihood:

Recommendation

It is recommended to fix the present typo.

Remediation

SOLVED: The **RFX team** solved the issue when refactoring the codebase to v2.1.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.19 INCORRECT COMMENTS

// INFORMATIONAL

Description

Incorporating comments into the codebase is essential for elucidating key aspects and facilitating comprehension of the developer's intentions by others. Nevertheless, several errors were identified within these comments:

In the `OracleUtils` contract, the `RealtimeFeedReport` structure's comments incorrectly referenced `bid` and `ask` parameters, which have since been deprecated and renamed to `minPrice` and `maxPrice` respectively.

```
73  // bid: min price, highest buy price
74  // ask: max price, lowest sell price
75  struct RealtimeFeedReport {
76      // The feed ID the report has data for
77      bytes32 feedId;
78      // Min price
79      uint256 minPrice;
80      // Max price
81      uint256 maxPrice;
82      // Price exponent
83      int32 expo;
84      // Unix timestamp describing when the price was published
85      uint publishTime;
86      }
```

In the `Oracle` contract, it was stated that the `setPrices` function checked the number of signers and then called `_setPrices` and `_setPricesFromPriceFeeds` to set tokens prices. However, this logic no longer applies in the current implementation.

```
113 // bits contain the index of each signer in the oracleStore. The functi
114 // of signers is greater than or equal to the minimum number of signers
115 // the signer indices are unique and within the maximum signer index. T
116 // _setPrices and _setPricesFromPriceFeeds to set the prices of the tok
```

Below is the implementation of the `setPrices` function:

```
199 function setPrices(
200         DataStore dataStore,
201         EventEmitter eventEmitter,
202         OracleUtils.SetPricesParams memory params
```

```
203     ) external payable onlyController {
204         if (tokensWithPrices.length() != 0) {
205             revert Errors.NonEmptyTokensWithPrices(tokensWithPrices.ler
206         }
207 
208         _setPricesFromRealtimeFeeds(dataStore, eventEmitter, params);
209     }
```

Furthermore, examples within the comments of the `Oracle.setPrices` function contained incorrect exponent values, leading to inaccurate prices:
In the comment below, `(2 ^ 64)` should have been `(2 ^ 32)` instead:

```
// USDC prices maximum value: `(2 ^ 64) / (10 ^ 6) => 4,294,967,296 / (10 ^ 6) =>
4294.967296`.
```

In the comment below, `(10 ^ 3)` should have been `(10 ^ 4)` instead:

```
// Price would be stored as `1 * (10 ^ -26) * (10 ^ 30) => 1 * (10 ^ 3)`.
```

In the comment below, `(2 ^ 64)` should have been `(2 ^ 32)` instead:

```
// DG prices maximum value: `(2 ^ 64) / (10 ^ 11) => 4,294,967,296 / (10 ^ 11) =>
0.04294967296`.
```

Score

Impact:

Likelihood:

Recommendation

It is recommended to remove deprecated comments and correct any comments that contain errors.

Remediation

SOLVED: The **RFX team** solved the issue when refactoring the codebase to v2.1.

Remediation Comment

Fixed/Refactored in RFX2.1

Remediation Hash

https://github.com/relative-finance/rfx-contracts/tree/develop-v2

8.20 UNUSED IMPORT STATEMENTS

// INFORMATIONAL

Description

Within the codebase, the following unused imports were identified:

oracle/OracleUtils.sol:

- `import "../utils/Printer.sol";`

oracle/Oracle.sol:

- `import "@openzeppelin/contracts/utils/math/SafeCast.sol";`
- `import "@openzeppelin/contracts/utils/Strings.sol";`
- `import "./IPriceFeed.sol";`
- `import "./IRealtimeFeedVerifier.sol";`
- `import "../utils/Bits.sol";`
- `import "../utils/Array.sol";`
- `import "hardhat/console.sol";`

utils/Precision.sol:

- `import "@openzeppelin/contracts/utils/math/SafeMath.sol";`

deposit/ExecuteDepositUtils.sol:

- `import "../error/ErrorUtils.sol";`

swap/SwapUtils.sol:

- `import "../token/TokenUtils.sol";`

withdrawal/ExecuteWithdrawalUtils.sol:

- `import "../adl/AdlUtils.sol";`
- `import "../nonce/NonceUtils.sol";`
- `import "../oracle/OracleUtils.sol";`
- `import "../utils/AccountUtils.sol";`

order/SwapOrderUtils.sol:

- `import "../order/OrderStoreUtils.sol";`

order/DecreaseOrderUtils.sol:

- `import "../order/OrderStoreUtils.sol";`

order/IncreaseOrderUtils.sol:

- `import "../order/OrderStoreUtils.sol";`
- `import "../callback/CallbackUtils.sol";`

Score

Impact:

Likelihood:

Recommendation

Consider removing the unused import statements.

Remediation

PARTIALLY SOLVED: The **RFX team** has partially solved this issue by removing most of the unused import statements. However, there is still one unused import that remains in the **order/DecreaseOrderUtils.sol** contract:

- `import "../order/OrderStoreUtils.sol";`

Remediation Hash

https://github.com/relative-finance/rfx-contracts/pull/49/commits/e715715e56576349601cf24fee317bffd7d8ca60

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.