Affluent Strategy Vault and RFQ Contracts

Security Assessment

June 13, 2025

Prepared for:
Hyungyeon Lee
Affluent

Prepared by: **Tarun Bansal and Quan Nguyen**

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Mary O'Brien, Project Manager
mary.obrien@trailofbits.com

The following engineering director was associated with this project:

Benjamin Samuels, Engineering Director, Blockchain
benjamin.samuels@trailofbits.com

The following consultants were associated with this project:

Tarun Bansal, Consultant
tarun.bansal@trailofbits.com

Quan Nguyen, Consultant
quan.nguyen@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
April 30, 2025	Pre-project kickoff call
May 15, 2025	Delivery of report draft
May 15, 2025	Report readout meeting
May 23, 2025	Completion of fix review
June 13, 2025	Delivery of final comprehensive report

Executive Summary

Engagement Overview

Affluent engaged Trail of Bits to review the security of its strategy vault and request for quote (RFQ) contracts built on the TON blockchain. The strategy vault contract allows users to deposit assets that vault managers allocate across investment strategies to generate yield, with tokenized shares representing user positions. The RFQ system, comprising the auction and batch contracts, implements an order book exchange for assets. The on-chain data aggregator v2 contract helps other contracts consolidate the pricing data from multiple sources for accurate asset valuation.

A team of two consultants conducted the review from May 5 to May 14, 2025, for a total of three engineer-weeks of effort. With full access to source code and documentation, we performed comprehensive static and dynamic analysis of the strategy vault and RFQ contracts, focusing on the deposit/withdrawal mechanisms, exposure caps validation, fee collection, and trading safeguards. Through our testing efforts, we aimed to identify opportunities for manipulation by privileged roles, unbounded loops, out-of-gas exceptions, and denial-of-service vulnerabilities, and we examined asset valuation during critical operations and risk management across external interactions.

Observations and Impact

The codebase is structured well and is broken down into small files that implement limited functionality to manage complexity. The documentation and inline code comments help developers and reviewers navigate the code and follow user action message flows through different protocol smart contracts.

We identified two high-severity issues related to an unbounded `while` loop that could trigger an out-of-gas exception (TOB-FACTRFQ-1) and incorrect handling of a failure case (TOB-FACTRFQ-2). We discovered two medium-severity issues involving an out-of-order execution of an action (TOB-FACTRFQ-3) and a missing status check in the RFQBatch contract that leads to accepting new orders after settlement (TOB-FACTRFQ-9). We also found informational-severity issues related to input data validation, incorrect arithmetic operations, and inefficient access control and incentive mechanism design.

Recommendations

Based on the findings identified during the security review, Trail of Bits recommends that Affluent take the following steps before deploying the protocol smart contracts in production:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or any refactoring that may occur when addressing other recommendations.

- **Improve the test suite.** Consider failure cases of cross-contract interactions and add test cases to ensure correct behavior and recovery from such failure cases. Consider invalid or malicious user inputs to test protocol stability against them.

Finding Severities and Categories

The following tables provide the number of findings by severity and category.

EXPOSURE ANALYSIS

Severity	*Count*
High	2
Medium	2
Low	0
Informational	4
Undetermined	1

CATEGORY BREAKDOWN

Category	*Count*
Data Validation	3
Denial of Service	4
Timing	1
Undefined Behavior	1

Project Goals

The engagement was scoped to provide a security assessment of the Affluent strategy vault and RFQ contracts. Specifically, we sought to answer the following non-exhaustive list of questions:

- Can an attacker render an RFQ auction or batch permanently unresolvable?
- Can a bidder manipulate the auction settlement process to receive assets without proper payment?
- Can a malicious user prevent the legitimate settlement of an RFQ batch?
- Are there gas-related vulnerabilities that could lead to transaction failures during critical operations?
- Are there vulnerabilities in the composite oracle system that could lead to incorrect price calculations?
- Can the vault manager execute unauthorized strategies that violate investment constraints?
- Can the owner or manager of a strategy vault steal user deposits?
- Can a user cancel another user's bid or order without proper authorization?
- Are the access controls effective across all privileged operations?
- Can the on-chain data aggregator be manipulated to return incorrect price information?
- Can the fee collection mechanisms be exploited to steal or lock funds?
- Are all protocol operations robust against edge case inputs and error conditions?
- Can race conditions be exploited to take malicious actions such as the following?
    - Prevent RFQ settlement
    - Execute multiple conflicting strategies simultaneously
    - Manipulate asset prices during critical operations
    - Increase a user's share value or otherwise corrupt the contract storage

Project Targets

The engagement involved reviewing and testing the following target.

factorial-core

Repository	https://github.com/factorial-finance/factorial-core
Version	Commit `0741a058`
Type	FunC
Platform	TVM

factorial-core

Repository	https://github.com/factorial-finance/factorial-core
Version	PR #4
Type	FunC
Platform	TVM

Project Coverage

This section provides an overview of the analysis coverage of the review, as determined by our high-level engagement goals. Our approaches included the following:

- **General:** We reviewed the whole codebase for common FunC and TON blockchain flaws, such as missing or incorrect gas checks, issues with bounced message handling, and message parsing and building errors. We checked the use of modifying functions, message and storage cell limits, error handling logic, maintenance of the contracts' TON balance, and contract deployment and initialization, all for correctness.

- **`StrategyVault` contract:** We analyzed all user-facing operations (deposit, withdraw) and manager actions (supply, borrow, repay, liquidate, create RFQ). We verified the exposure cap calculation mechanism, asset whitelist enforcement, management and protocol fee collection, and interaction with external protocols. We assessed the security of oracle integration for price discovery and evaluated the contract's robustness against potential manipulation by privileged roles.

- **RFQ auction and batch contracts:** We performed a comprehensive review of all RFQ operations across both the auction and batch contracts, including the creation, initialization, deposit handling, bid and order placement, locking, matching, settlement, payout, and cancellation workflows. We analyzed the security of the auction modes (amount-bid and slippage-bid), evaluated the oracle price deviation checks, verified the whitelist controls, examined the fee collection logic, and assessed potential denial-of-service vectors. We paid particular attention to edge cases in the settlement processes that could lead to fund locking or manipulation, and we investigated possible attack vectors through order or bid submission.

- **On-chain data aggregator v2:** We reviewed the entire contract, examining the asynchronous data collection process from multiple external sources (DeDust, STON.fi, Storm, Affluent pools, Affluent vaults). We verified the contract's self-destruct life cycle, evaluated its resilience against fake deployment attacks, analyzed its access control mechanisms, and investigated potential denial-of-service vulnerabilities. We also assessed whether index manipulation could affect the contract's operation or create race conditions across multiple data requests.

- **Composite oracle:** We examined the RedStone signature verification process and multi-source price data calculation logic. We assessed how on-chain price data is aggregated, validated, and used to compute asset values and vault compositions. We verified that errors are properly handled during validation failure and assessed the vault price calculation mechanisms for robustness against manipulation.

Coverage Limitations

Because of the time-boxed nature of testing work, it is common to encounter coverage limitations. During this project, we were unable to perform comprehensive testing of the following system elements, which may warrant further review:

- Contract deployment and initialization scripts
- Off-chain components interacting with the smart contracts
- Race conditions arising in complex scenarios involving the `StrategyVault` contract integration with other contracts
- Owner actions of all the smart contracts

Summary of Findings

The table below summarizes the findings of the review, including details on type and severity.

ID	Title	Type	Severity
1	Missing validation and an unbounded while loop in the RFQAuction contract enable denial-of-service attacks	Denial of Service	**High**
2	The StrategyVault contract can get locked for strategy execution actions	Denial of Service	**High**
3	The collect_fee function of the RFQBatch contract can lock assets	Timing	**Medium**
4	The RFQBatch contract is vulnerable to denial-of-service attacks	Denial of Service	**Informational**
5	Lack of order value validation enables denial-of-service attacks against the RFQBatch contract	Denial of Service	**Informational**
6	An incorrect total value is used in the exposure cap check in the StrategyVault contract	Undefined Behavior	**Informational**
7	Redundant whitelist check in the take_wallet_address handler of the StrategyVault contract	Data Validation	**Informational**
8	The on-chain data aggregator contract waits for a response from an unknown source type	Data Validation	**Undetermined**
9	The RFQBatch contract accepts new orders after settlement	Data Validation	**Medium**

Detailed Findings

1. Missing validation and an unbounded while loop in the RFQAuction contract enable denial-of-service attacks	
Severity: **High**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-FACTRFQ-1
Target: `contracts/rfq/rfq_auction/handlers/jetton/transfer_notification/bid.fc`	

Description

An attacker can exploit missing validation in the `handle::bid()` function and an unbounded `while` loop in the `get_settleable_max_bid` function to trigger an out-of-gas error, ultimately causing a denial-of-service on auctions.

In the `RFQAuction` contract, users place bids by transferring the buy asset directly to the auction contract's buy asset wallet, which triggers the `handle::bid()` function. While this function verifies that the `bid_value`, a numerical value specified by the user in the message body, is higher than the `min_buy_asset_value`, it does not validate the actual amount of tokens transferred, allowing attackers to submit valid bids by transferring an insignificant number of tokens:

```
(int bid_value, slice response_address, int forward_ton_amount, cell
forward_payload) = in_msg_body~load_bid_params();

;; Validate the bid value
int min_buy_asset_value = storage.get_min_buy_asset_value();
int max_buy_asset_value = storage.get_max_buy_asset_value();
throw_unless(error::bid_value_too_low, min_buy_asset_value <= bid_value);
```

Figure 1.1: Excerpt of the `bid` function (`contracts/rfq/rfq_auction/handlers/jetton/transfer_notification/bid.fc`)

During the settlement, the `get_settleable_max_bid` function iterates through all the bids stored in the dictionary using an unbounded `while` loop. Because the loop is unbounded, an attacker can cause an out-of-gas exception from this function by flooding the dictionary with a large number of bids. And if the `allow_bidder_cancel` flag is set to `false`, users would not be able to cancel bids or clear the auction, so this attack would permanently lock the auction in an unresolvable state.

```
;; Iterate through all bids
while(bid_found?) {
    ;; Check if the bid value is the highest or if it matches but has a smaller LT
    if (bid_value > max_bid_value) | ((bid_value == max_bid_value) & (lt <
max_bid_lt)) {
        ;; Calculate the bid amount based on the auction mode
        int bid_amount = auction_mode == auction_mode::amount_bid ? bid_value :
buy_asset_amount_by_oracle.muldiv(bid_value, DECIMALS_OF_SLIPPAGE);
        ;; Calculate the gap between the oracle amount and the bid amount
        int gap = abs(buy_asset_amount_by_oracle - bid_amount);
        ;; If the gap is within tolerance and escrow is sufficient, update the max
bid
        if (gap <= limit_amount_gap) & (escrow_amount >= bid_amount) {
            (max_bid_value, settle_amount, max_bid_lt, max_bidder) = (bid_value,
bid_amount, lt, bidder);
        }
    }
    ;; Move to the next bid
    (bidder, bid_value, escrow_amount, lt, _, _, _, bid_found?) =
bids.bids::next?(bidder);
}
```

Figure 1.2: Excerpt of the `get_settleable_max_bid` function
(`contracts/rfq/rfq_auction/types/bids.fc`)

Exploit Scenario

Eve identifies a valuable RFQ auction and proceeds to submit 1 million bids from 1 million different accounts by transferring only 1 nanoton and specifying a valid `bid_value` for each bid. When the settlement period begins, the `get_settleable_max_bid` function attempts to process all 1 million entries sequentially, exceeding the gas limit and causing an execution failure. If `allow_bidder_cancel` is set to `false`, users have no mechanism to cancel bids or reset the auction. The contract becomes permanently locked, trapping the seller's assets and all legitimate bids.

Recommendations

Short term, add a validation check on the actual token amount in the `handle::bid` function to ensure that the total escrow amount is more than the specified `bid_value` for `auction_mode::bid_amount` auctions. Add a mechanism to allow the seller to cancel an `auction_mode::slippage_bid` type auction in the event of an out-of-gas error in the settlement function.

Long term, refactor the bid management system to use an optimized data structure that maintains bid order without requiring full iteration; alternatively, have the settlement function process bids in batches. Carefully analyze and mitigate the risk of out-of-gas errors arising from unbounded loops.

2. The StrategyVault contract can get locked for strategy execution actions	
Severity: **High**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-FACTRFQ-2
Target: `contracts/vault/strategy_vault/logics/manager/create_rfq.fc`	

Description

The `StrategyVault` contract's deposit to the `RFQAuction` contract can fail, leaving the `StrategyVault` contract with the `is_executing_strategy` flag set to `true` forever, causing all strategy execution actions to fail, and locking deposited funds in the contract.

If the deposit from the `StrategyVault` contract to the `RFQAuction` contract fails, the `RFQAuction` contract does not send any transfer notification with the jetton transfer for returning the deposited tokens. The `StrategyVault` contract gets back the deposited tokens, but it cannot update its internal balance of the deposited asset or set the `is_executing_strategy` flag to `false`.

```
} catch(_, int error_code) {
    ;; On failure, emit the error code
    emit_log_simple_error(op::transfer_notification, error_code);
    ;; On failure, refund the transferred tokens to the sender
    send::jetton_transfer(ctx, storage, 0, CARRY_REMAINING_GAS, 0,
ctx.sender_address(), from_address, from_address, amount, null());
    return ();
}
```

Figure 2.1: The exception handler of the deposit action in the `RFQAuction` contract (`contracts/rfq/rfq_auction/handlers/jetton/transfer_notification.fc#L32–L38`)

As a result, no other strategy execution action can be executed by the manager. Users cannot withdraw the deposited assets because of an incorrect value of the asset balance stored in the contract. There is no way to recover from this state.

Exploit Scenario

Alice, the strategy vault manager, mistakenly initializes the `RFQAuction` contract with the wrong `sell_asset_wallet_address`. As a result, the sell asset deposit from the `StrategyVault` contract to the `RFQAuction` contract fails, locking the `StrategyVault` contract for all strategy execution actions without updating the sell asset balance to the original value.

Recommendations

Short term, have the `RFQAuction` contract send a transfer notification when returning funds if the deposit from `StrategyVault` fails. Implement a way to set the `is_executing_strategy` flag to `true` in the event of an unexpected failure.

Long term, consider failure cases for all cross-contract interactions and implement measures to recover from unexpected transaction failures.

3. The collect_fee function of the RFQBatch contract can lock assets	
Severity: **Medium**	Difficulty: **High**
Type: Timing	Finding ID: TOB-FACTRFQ-3
Target: `contracts/rfq/rfq_batch/handlers/fee_manager/collect_fee.fc`	

Description

The lack of a gas check in the `collect_fee` function of the `RFQBatch` contract can cause the last batch of the `settle` function to fail, locking user funds in the contract.

The `handle::payout` function iterates through all the bid orders in batches of 100 orders in a single transaction; it computes the amount to be sent to the bidder based on the settlement price, deducts the manager fee, and transfers the buy asset to the bidder. The `RFQBatch` contract collects the gas fee for the jetton transfers triggered in the `handle::payout` function from the bidders.

The fee manager can execute the `collect_fee` action to transfer the accrued management fee from the `RFQBatch` contract anytime:

```
_ handle::collect_fee(tuple ctx, tuple storage, slice in_msg_body) impure inline {
    throw_unless(error::unauthorized, equal_slices(ctx.sender_address(),
storage.get_fee_manager()));

    int asset0_collected_fee = storage.get_asset0_collected_fee();
    int asset1_collected_fee = storage.get_asset1_collected_fee();

    send::jetton_transfer(ctx, storage, fee::jetton_transfer, PAY_FEES_SEPARATELY,
0, storage.get_asset0_wallet_address(), ctx.sender_address(), ctx.sender_address(),
asset0_collected_fee, null());
    send::jetton_transfer(ctx, storage, fee::jetton_transfer, PAY_FEES_SEPARATELY,
0, storage.get_asset1_wallet_address(), ctx.sender_address(), ctx.sender_address(),
asset1_collected_fee, null());

    storage~set_asset0_collected_fee(0);
    storage~set_asset1_collected_fee(0);
    storage::save(storage);

    send::excesses(ctx, ctx.sender_address(), NANOTON_1e8);
    return ();
}
```

Figure 3.1: The `collect_fee` function of the `RFQBatch` contract (`contracts/rfq/rfq_batch/handlers/fee_manager/collect_fee.fc#L1–L16`)

However, there is no gas check in the `handle::collect_fee` function to ensure that the fee manager sends enough value to pay for two jetton transfers. Additionally, the `handle::collect_fee` function sends the remaining TON balance of the contract to the message sender. If the fee manager executes the `collect_fee` function before the execution of all payout batches, then `collect_fee` will send the contract TON balance reserved for processing the payout function to the sender. It will cause the remaining batches of the payout to fail with an out-of-gas error, locking the buy assets in the contract.

Exploit Scenario

Bob has settled an `RFQBatch` auction contract. Alice starts processing payouts by executing the `payout` action in batches. Alice needs to process 10 batches. Eve, the fee manager, executes the `collect_fee` action after six batches of the `payout` action have been processed. As a result, the `collect_fee` function drains the contract's TON balance. The remaining four batches of the `payout` action fail with an out-of-gas error.

Recommendations

Short term, ensure that the fee manager can collect the fee only after all the bid orders have been processed.

Long term, improve the test suite to include test cases that check smart contract TON balance after user actions to ensure that users cannot drain a contract's TON balance.

4. The RFQBatch contract is vulnerable to denial-of-service attacks

Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-FACTRFQ-4
Target: `contracts/rfq/rfq_batch/handlers/public/lock.fc`	

Description

The `RFQBatch` contract's settlement process is vulnerable to denial-of-service attacks when no settler whitelist is configured.

The settlement process begins with a user locking the `RFQBatch` contract to safely execute the `settle` function, with a whitelist check to verify eligible settlers. When no whitelist exists, any user can lock an `RFQBatch` contract without any commitment or penalty for failing to settle, creating an opportunity for malicious actors to indefinitely prevent legitimate settlements.

```
;; Whitelist check logic: if whitelist exists, caller must be on it
ifnot (whitelisted_settlers.cell_null?()) {
    ;; Whitelist is configured - verify sender is on the whitelist
    (_, int found?) = whitelisted_settlers.adict_get?(ctx.sender_address());
    throw_unless(error::not_whitelisted_settler, found?);
}
;; If no whitelist exists, proceed (open to anyone)
```

Figure 4.1: Excerpt of the `lock` function
(`contracts/rfq/rfq_batch/handlers/public/lock.fc`)

Exploit Scenario

Eve repeatedly locks an `RFQBatch` contract and intentionally lets the lock expire without settling, effectively blocking legitimate sellers from using the contract. Considering the lock duration is 1 hour, Eve keeps sending a new lock transaction every hour, preventing settlements for days or weeks with minimal gas expenditure.

Recommendations

Short term, add a lock fee that can be refunded in the event of a successful settlement but is not refunded if a settlement is not initiated before the lock expiry time.

Long term, design the incentive mechanisms for public actions to prevent malicious behavior. Analyze the cost versus the impact of every malicious public action, and implement measures to prevent such attacks.

5. Lack of order value validation enables denial-of-service attacks against the RFQBatch contract

Severity: **Informational**	Difficulty: **High**
Type: Denial of Service	Finding ID: TOB-FACTRFQ-5
Target: `contracts/rfq/rfq_batch/handlers/jetton/transfer_notification/add_order.fc`	

Description

The `RFQBatch` contract is vulnerable to denial-of-service attacks due to a lack of minimum order value validation checks.

While the `RFQBatch` contract enforces a maximum order count per batch, it allows orders of any value to be placed. The code checks only if the order count limit has been reached:

```
throw_if(error::order_limit_reached, storage.get_order_count() >=
storage.get_order_limit());
```

Figure 5.1: The order count limit check in the `add_order` function (`contracts/rfq/rfq_batch/handlers/jetton/transfer_notification/add_order.fc`)

The lack of an order value validation check allows attackers to submit numerous minimal-value orders (e.g., 1 nanoton), quickly exhausting the order count limit and preventing legitimate users from participating in the batch at an insignificant cost.

Recommendations

Short term, implement a minimum order value check that rejects orders below a reasonable economic threshold. This threshold should be set high enough to make the attack economically unfeasible while remaining accessible to legitimate small traders.

Long term, design the incentive mechanisms for public actions to prevent malicious behavior. Analyze the cost versus the impact of every malicious public action, and implement measures to prevent such attacks.

6. An incorrect total value is used in the exposure cap check in the StrategyVault contract

Severity: **Informational**	Difficulty: **Low**
Type: Undefined Behavior	Finding ID: TOB-FACTRFQ-6
Target: `contracts/vault/strategy_vault/logics/manager/create_rfq.fc`	

Description

The `create_rfq` function of the `StrategyVault` contract fails to properly account for portfolio value changes when validating asset exposure caps, leading to overexposure to an asset.

The `create_rfq` function of the `StrategyVault` contract uses the original `total_value` for both sell and buy asset exposure cap validations, without adjusting for the impact of the swap on the portfolio's total value. The `total_value` may decrease because of the fee and slippage of the swap operation, and use of an incorrect `total_value` can lead to overexposure to an asset through execution of a strategy.

```
logic::validate_exposure_cap(assets, sell_asset, -1 * sell_asset_value, total_value,
per_asset_values);
logic::validate_exposure_cap(assets, buy_asset, min_buy_asset_value, total_value,
per_asset_values);
```

Figure 6.1: Excerpt from the function `create_rfq`
(`contracts/vault/strategy_vault/logics/manager/create_rfq.fc`)

Recommendations

Short term, use an adjusted `total_value` - `sell_asset_value` + `min_buy_asset_value` calculation in both exposure cap validations.

Long term, identify and document all functions requiring exposure cap validation and ensure they consistently use post-transaction total values for accurate exposure checks.

7. Redundant whitelist check in the take_wallet_address handler of the StrategyVault contract

Severity: **Informational**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FACTRFQ-7
Target: `contracts/vault/strategy_vault/handlers/jetton/take_wallet_address.fc`	

Description

The `handle::take_wallet_address` function of the `StrategyVault` contract throws an exception if the sender address is not a whitelisted asset. Because of this, if the owner adds an asset with the `is_whitelisted` flag set to `false`, then the asset wallet address will not be stored in the `StrategyVault` contract. Later, if the owner marks the asset as whitelisted, then it will be unusable because the asset wallet address does not exist in the contract storage. The owner will need to remove it and add it again as a whitelisted asset to use it.

```
_ handle::take_wallet_address(tuple ctx, tuple storage, slice in_msg_body) impure
inline {
    slice wallet_address = in_msg_body~load_msg_addr();
    var assets = storage.get_assets();
    (_, _, int is_whitelisted?, _, _, _) = assets.assets::get(ctx.sender_address());
    throw_unless(error::invalid_asset, is_whitelisted?);

    ;; dict(asset -> wallet)
    var asset_wallet_dict = storage.get_asset_wallet_dict();
    asset_wallet_dict~adict_set(ctx.sender_address(), wallet_address);
    storage~set_asset_wallet_dict(asset_wallet_dict);

    ;; dict(wallet -> asset)
    var wallet_asset_dict = storage.get_wallet_asset_dict();
    wallet_asset_dict~adict_set(wallet_address, ctx.sender_address());
    storage~set_wallet_asset_dict(wallet_asset_dict);

    storage::save(storage);
    return ();
}
```

Figure 7.1: The `handle::take_wallet_address` function of the `StrategyVault` contract (`contracts/vault/strategy_vault/handlers/jetton/take_wallet_address.fc#L15–L33`)

Recommendations

Short term, update the `handle::take_wallet_address` function to throw an exception if the sender address does not exist in the configured assets list.

Long term, expand the test suite to test the effect of all the state changes made by the owner.

8. The on-chain data aggregator contract waits for a response from an unknown source type

Severity: **Undetermined**	Difficulty: **High**
Type: Data Validation	Finding ID: TOB-FACTRFQ-8
Target: `contracts/onchain_data_aggregator_v2/onchain_data_aggregator_v2.fc`	

Description

The `onchain_data_aggregator_v2` contract iterates through the list of all the request sources and sends a message to the source address based on the source type in order to get the data and add the source address to a dictionary tracking the list of pending responses:

```
if (source_type == source_type::stonfi) {
    send::provide_stonfi_pool_data(ctx, source_address);
} elseif (source_type == source_type::dedust) {
    send::provide_dedust_pool_data(ctx, source_address);
} elseif (source_type == source_type::storm) {
    send::provide_storm_pool_data(ctx, source_address);
}
;; add source_address to waiting_for_response
waiting_for_response~adict_set(source_address, asset_address);
```

Figure 8.1: A snippet of the `provide_aggregated_data` action handler (`contracts/onchain_data_aggregator_v2/onchain_data_aggregator_v2.fc#L100–L108`)

However, it adds an entry for the source address in the `waiting_for_response` dictionary even if the `source_type` is not recognized and a message is not sent to the `source_address`. This results in the `onchain_data_aggregator_v2` contract waiting for a response from a source address that will not send any message, making the contract unable to send the collected data to the owner.

Recommendations

Short term, have the `onchain_data_aggregator_v2` contract add the `source_address` to the `waiting_for_response` dictionary only if a message is sent to the `source_address`.

Long term, improve the test suite to check for malicious or invalid user inputs.

9. The RFQBatch contract accepts new orders after settlement

Severity: **Medium**	Difficulty: **Low**
Type: Data Validation	Finding ID: TOB-FACTRFQ-9
Target: `contracts/rfq/rfq_batch/handlers/jetton/transfer_notification/add_order.fc`	

Description

The `handle::add_order` function of the `RFQBatch` contract allows new orders to be added when the contract `status` is set to `status::closed`. This can lead to a loss of funds for some users.

The `handle::add_order` function of the `RFQBatch` contract allows users to add a new order or modify their existing order in the order book:

```
() handle::add_order(tuple ctx, tuple storage, int amount, slice from_address, slice
in_msg_body) impure inline {
    ;; 1. Validate contract state
    ;; Check if the position is locked - execution fails if locked timestamp is
future
    throw_if(error::locked, storage.get_locked_timestamp() > now());
    throw_if(error::order_limit_reached, storage.get_order_count() >=
storage.get_order_limit());

    ;; 2. Parse incoming message data
    ;; Extract response_address (where to send responses), forwarding details
    (slice response_address, int forward_ton_amount, cell forward_payload) =
in_msg_body~load_order_params();
```

Figure 9.1: Validation checks in the `add_order` action handler (`contracts/rfq/rfq_batch/handlers/jetton/transfer_notification/add_order.fc#L9–L17`)

The `RFQBatch` contract's `status` is set to `status::open` when the contract is initialized. Users add orders by executing the `add_order` action. The settler locks the contract and executes the `settle` action by transferring enough funds to settle all the orders stored at the time of locking the contract. The `logic::settle` function sets the `locked_timestamp` to zero and `status` to `status::closed`, along with the asset price. The `payout` action is used to process all the orders in order of user address to send the required asset amount based on the asset price set at the time of the settlement.

However, the `handle::add_order` function does not check the contract `status` and allows users to add new orders or modify their existing orders even when the `status` is set to `status::closed`. An attacker can use it to add a new order with an address lower than the address of the last user in the order book or to increase the order value of an existing order before all the payouts are processed. These new orders will consume the fixed asset amount transferred at the time of settlement and will lead to the last user getting a lower amount than the one calculated based on the asset price.

Exploit Scenario

Alice places an order to buy 1,000 USDT from the `RFQBatch` contract. After the RFQ batch has been settled, Eve adds a new order to buy 900 USDT. The `payout` action processes Eve's order before Alice's order and transfers 900 USDT to Eve and only 100 USDT to Alice. Alice loses 900 USDT from this order.

Recommendations

Short term, add a validation check in the `handler::add_order` function to ensure that the contract `status` is set to `status::opened`.

Long term, improve the test suite to check restrictions on user actions in every contract state.

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories

Category	Description
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. However, implementing them can enhance the code's readability and may prevent the introduction of vulnerabilities in the future.

1. **Remove unused code.** The `handle::take_wallet_address` function of the `RFQAuction` contract loads the state variable from storage but never uses it before saving it back to storage unchanged. This creates unnecessary gas consumption with no functional benefit, as the function reads and writes the same value without any manipulation.

2. **Use accurate parameter naming.** In the `get_asset_address` function of the `StrategyVault` contract storage, the input parameter should be named `wallet_address` instead of `asset_address` to accurately reflect its purpose and avoid confusion.

3. **Remove unused variables.** The `logic::create_rfq` function of the `StrategyVault` contract declares the variables `exposure_cap` and `buy_asset_exposure_cap` but never uses them, adding unnecessary complexity to the code.

4. **Remove redundant validation checks.** In the `set_max_leverage_ratio` owner action handler of the `StrategyVault` contract, there is a duplicated check for the `MAX_LEVERAGE_RATIO`, which unnecessarily increases gas costs and adds no additional security.

5. **Fix a discrepancy between comment and code.** In the `handle::cancel_bid` function, the bid cancellation implementation contradicts its documentation. While comments state cancellation requires both `allow_bidder_cancel` to be `true` and the auction to be open, the code throws an error only when cancellation is not allowed and the auction is open. This discrepancy creates potential security risks where bids might be cancelable in unauthorized states.

C. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On May 23, 2025, Trail of Bits reviewed the fixes and mitigations implemented by the Affluent team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, Affluent has resolved all eight issues described in this report. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Severity	Status
1	Missing validation and an unbounded while loop in the RFQAuction contract enable denial-of-service attacks	High	Resolved
2	The StrategyVault contract can get locked for strategy execution actions	High	Resolved
3	The collect_fee function of the RFQBatch contract can lock assets	Medium	Resolved
4	The RFQBatch contract is vulnerable to denial-of-service attacks	Informational	Resolved
5	Lack of order value validation enables denial-of-service attacks against the RFQBatch contract	Informational	Resolved
6	An incorrect total value is used in the exposure cap check in the StrategyVault contract	Informational	Resolved
7	Redundant whitelist check in the take_wallet_address handler of the StrategyVault contract	Informational	Resolved
8	The on-chain data aggregator contract waits for a response from an unknown source type	Undetermined	Resolved

9	The RFQBatch contract accepts new orders after settlement	**Medium**	**Resolved**

Detailed Fix Review Results

TOB-FACTRFQ-1: Missing validation and an unbounded while loop in the RFQAuction contract enable denial-of-service attacks

Resolved in commit 47c28e7. The bid action handler now checks that the bid_value is lower than the total escrow amount. The Affluent team has also added a new settlement period feature, which allows users to cancel an auction if it is not settled within the settlement_period. The settlement_period begins after the auction bid deadline. If an auction is stuck because of an out-of-gas error, then users can cancel the auction after the settlement_period expires.

TOB-FACTRFQ-2: The StrategyVault contract can get locked for strategy execution actions

Resolved in commit 300bb2a. The deposit action handler function now uses a try/catch block and sends the forward_payload with the jetton transfer in the event of an error. This forward_payload sets the correct balance of the StrategyVault contract. The cancel_rfq function has been updated to allow cancellation even if the sell_asset_deposited flag is set to false, allowing the StrategyVault contract manager a way to reset the value of the is_executing_strategy flag to false if the deposit fails.

TOB-FACTRFQ-3: The collect_fee function of the RFQBatch contract can lock assets

Resolved in commit 7f2dc1e. The handle::collect_fee function now checks that the auction has been closed and all the bids have been paid out.

TOB-FACTRFQ-4: The RFQBatch contract is vulnerable to denial-of-service attacks

Resolved in commit 5832564. The RFQ batch contract now allows only the whitelisted addresses to lock the contract.

TOB-FACTRFQ-5: Lack of order value validation enables denial–of-service attacks against the RFQBatch contract

Resolved in commit 1e92af3. The handle::add_order function of the RFQ batch contract now validates the deposit amount to be more than the configured minimum order amount for the asset.

TOB-FACTRFQ-6: An incorrect total value is used in the exposure cap check in the StrategyVault contract

Resolved in commit c17f2c5. The exposure cap check in the create_rfq action handler of the StrategyVault contract now uses the minimum expected total value after the RFQ settlement.

TOB-FACTRFQ-7: Redundant whitelist check in the take_wallet_address handler of the StrategyVault contract

Resolved in commit 607da5f. The take_wallet_address handler of the StrategyVault contract now checks if the sender exists in the assets dictionary instead of checking if the asset corresponding to the sender is whitelisted.

TOB-FACTRFQ-8: The on-chain data aggregator contract waits for a response from an unknown source type
Resolved in commit ab15a3c. The on-chain data aggregator contract now adds the source_address to the waiting_for_response dictionary only when the source_type is a known source type.

TOB-FACTRFQ-9: The RFQBatch contract accepts new orders after settlement
Resolved in commit fe1708f. The handle::add_order function now throws an error if the RFQBatch contract's status is set to status::closed.

D. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review assessments, supporting client organizations in the technology, defense, blockchain, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, Uniswap, Solana, Ethereum Foundation, Linux Foundation, and Zoom.

To keep up to date with our latest news and announcements, please follow @trailofbits on X or LinkedIn, and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact or email us at info@trailofbits.com.

Trail of Bits, Inc.
228 Park Ave S #80688
New York, NY 10003
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

Test Coverage Disclaimer

Trail of Bits performed all activities associated with this project in accordance with a statement of work and an agreed-upon project plan.

Security assessment projects are time-boxed and often rely on information provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test software controls and security properties. These techniques augment our manual security review work, but each has its limitations. For example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. A project's time and resource constraints also limit their use.