

RUJI Pools (BOW)

THORChain / Rujira

HALBORN



Prepared by:  HALBORN

Last Updated 05/02/2025

Date of Engagement: February 3rd, 2025 - February 5th, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

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

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 Precision loss in price calculation leading to underflow error
7.2 Rounding errors in market maker and xyk swap causing insufficient return error
7.3 Rujira - liquidity overwritten on deposit
8. Automated Testing

1. Introduction

THORChain engaged Halborn to conduct a security assessment of the Rujira Pools (BOW) contracts, beginning on February 3rd, 2025 and ending on February 5th, 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 successfully addressed by the Rujira team. The main ones were the following:

- Calculate the price using the total pool values to prevent precision loss and underflow errors.
- Ensure consistent rounding behavior in both MM and XYK swaps.

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

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

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

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

Impact Metric (M_I)	Metric Value	Numerical Value
Integrity (I)	None (I:N)	0
	Low (I:L)	0.25
	Medium (I:M)	0.5
	High (I:H)	0.75
	Critical (I:C)	1
Availability (A)	None (A:N)	0
	Low (A:L)	0.25
	Medium (A:M)	0.5
	High (A:H)	0.75
	Critical (A:C)	1
Deposit (D)	None (D:N)	0
	Low (D:L)	0.25
	Medium (D:M)	0.5
	High (D:H)	0.75
	Critical (D:C)	1
Yield (Y)	None (Y:N)	0
	Low (Y:L)	0.25
	Medium (Y:M)	0.5
	High (Y:H)	0.75
	Critical (Y:C)	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)	1
	Partial (R:P)	0.5
	Full (R:F)	0.25
Scope (s)	Changed (S:C)	1.25
	Unchanged (S:U)	1

Severity Coefficient *C* is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score *S* is obtained by:

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

The score is rounded up to 1 decimal places.

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

5. SCOPE

REPOSITORY
(a) Repository: rujira (b) Assessed Commit ID: 9158579 (c) Items in scope: contracts/rujira-bow/src/events.rs contracts/rujira-bow/src/bin/schema.rs contracts/rujira-bow/src/error.rs contracts/rujira-bow/src/config.rs contracts/rujira-bow/src/lib.rs contracts/rujira-bow/src/contract.rs packages/rujira-rs/src/interfaces/bow/strategy.rs packages/rujira-rs/src/interfaces/bow/arbitrage.rs packages/rujira-rs/src/interfaces/bow/interface.rs packages/rujira-rs/src/interfaces/bow/error.rs packages/rujira-rs/src/interfaces/bow/mod.rs packages/rujira-rs/src/interfaces/bow/xyk.rs 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
0

HIGH
3

MEDIUM
0

LOW
0

INFORMATIONAL
0

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PRECISION LOSS IN PRICE CALCULATION LEADING TO UNDERFLOW ERROR	HIGH	SOLVED - 04/08/2025
ROUNDING ERRORS IN MARKET MAKER AND XYK SWAP CAUSING INSUFFICIENT RETURN ERROR	HIGH	SOLVED - 04/08/2025
RUJIRA - LIQUIDITY OVERWRITTEN ON DEPOSIT	HIGH	SOLVED - 04/29/2025

7. FINDINGS & TECH DETAILS

7.1 PRECISION LOSS IN PRICE CALCULATION LEADING TO UNDERFLOW ERROR

// HIGH

Description

The `swap` function from `XYK pool` incorrectly calculates the price using `return_amount / offer_amount` instead of the total amounts in the pool. This results in **precision loss** that accumulates over swaps.

Eventually, when a user queries a price using `QueryMsg::Quote` and try to verify the minimum price condition, the total pool amounts are used for validation. The accumulated precision loss causes a **mismatch**, leading to an **underflow error** that **reverts all previously executed transactions** (including arbitrage operations and fixed-price orders).

Code Location

Code of `swap` function from `packages/rujira-rs/src/interfaces/bow/xyk.rs` file:

 Copy Code

```
78 pub fn swap(&mut self, offer_amount: Uint128) -> Result<(Uint128, Decimal), StrategyError> {
79     let k = self.k;
80     self.x = self.x.add(offer_amount);
81     let ratio = Decimal256::from_ratio(self.k, self.x);
82     let y = self.y;
83     // Ceil the ask amount in order to ensure rounding maintains new_k > self.k
84     self.y = Decimal256::one().mul(ratio).to_uint_ceil().try_into()?;
85     let return_amount = y.sub(self.y);
86     let price: Decimal = Decimal::from_ratio(return_amount, offer_amount);
87     self.k = Uint256::from(self.x) * Uint256::from(self.y);
88     ensure!(self.k >= k, StrategyError::Underflow {});
89     Ok((return_amount, price))
90 }
```

Code of `quote` function from `packages/rujira-rs/src/interfaces/bow/xyk.rs` file:

 Copy Code

```
134 fn quote(
135     &self,
136     deps: Deps,
137     env: Env,
138     req: QuoteRequest,
139 ) -> Result<Option<QuoteResponse>, StrategyError> {
140     let mut state: PoolState = match req.data {
141         Some(binary) => from_json(binary)?,
142         None => self.state(
143             deps,
144             env,
145             // Quote is always a query, no need for info.funds
146             vec![],
147             req.offer_denom.as_str(),
148             req.ask_denom.as_str(),
149         )?,
150     };
151     if state.k.lt(&self.min_k) {
152         return Ok(None);
153     }
154 }
155
```

```

155     let offer_size = match req.min_price {
156         Some(price) => {
157             let offer_dec = Decimal::from_ratio(state.x, 1u128);
158             let ask_dec = Decimal::from_ratio(state.y, 1u128);
159             offer_dec.mul(price).sub(ask_dec).div(price).to_uint_floor()
160         }
161         None => Decimal::from_ratio(state.x, 1u128)
162             .mul(self.step)
163             .to_uint_floor(),
164     };

```

Proof of Concept

Scenario:

This test simulates a **full distribution swap** in a **market-making (MM) environment**, utilizing **custom setup functions** and **balance query utilities**.

It begins by setting up a liquidity pool and initializing multiple users with balances. Three users create sell orders at a fixed price, and the market maker (MM) is also included as a liquidity provider.

A large swap operation is executed, aiming to distribute 8,100,000 units of eth-usdc, with 8,000,000 allocated to fixed-price orders and 100,000 to the MM. After the swap, the test queries remaining user orders and compares balance changes for the owner, contract, and MM to verify the distribution.

Test:

 Copy Code

```

#[test]
fn test_full_distribution_with_mm() {
    let (mut app, contract, contract_mm, fee_address) = setup_with_mm();

    let owner = app.api().addr_make("owner");
    let funds = vec![
        coin(500_000_000_000, "btc-btc"),
        coin(500_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();
    });

```

```
});
```

```
let owner_balance_before = app.wrap().query_all_balances(owner.clone()).unwrap();
let mm_balance_before = app.wrap().query_all_balances(contract_mm.clone()).unwrap();
let contract_balance_before = app.wrap().query_all_balances(contract.clone()).unwrap();
```

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

```
let funds = vec![
    coin(100_000_000, "btc-btc"),
    coin(100_000_000, "eth-usdc"),
];
```

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

```
app.execute_contract(
    user2.clone(),
    contract.clone(),
    &ExecuteMsg::Order(vec![
        (
            Side::Base,
            Price::Fixed(Decimal::from_str("1000.00").unwrap()),
            Uint128::from(2000u128),
        ),
    ]),
    &funds,
)
.unwrap();
```

```
app.execute_contract(
    user3.clone(),
    contract.clone(),
    &ExecuteMsg::Order(vec![
        (
            Side::Base,
            Price::Fixed(Decimal::from_str("1000.00").unwrap()),
            Uint128::from(5000u128),
        ),
    ]),
    &funds,
)
.unwrap();
```

```
println!("----> Swap: Full Distribution of 8_100_000 : 8_000_000 for Fixed, 100_000 for MM ----");
```

```
let swap_amount = coin(8_100_000, "eth-usdc");
```

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

```
println!("-----> Check Orders -----");
```

```

let orders_user: OrdersResponse = app
.wrap()
.query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user.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!("User Orders:");
for (i, order) in orders_user.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);
}

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

// Extract the balances for "btc-btc" and "eth-usdc" for each account.

// Owner balances.
let owner_btc_before = get_balance(&owner_balance_before, "btc-btc");
let owner_usdc_before = get_balance(&owner_balance_before, "eth-usdc");
let owner_btc_after = get_balance(&owner_balance_after, "btc-btc");
let owner_usdc_after = get_balance(&owner_balance_after, "eth-usdc");

// Contract balances.
let contract_btc_before = get_balance(&contract_balance_before, "btc-btc");
let contract_usdc_before = get_balance(&contract_balance_before, "eth-usdc");
let contract_btc_after = get_balance(&contract_balance_after, "btc-btc");
let contract_usdc_after = get_balance(&contract_balance_after, "eth-usdc");

//Market Maker balances
let mm_btc_before = get_balance(&mm_balance_before, "btc-btc");
let mm_usdc_before = get_balance(&mm_balance_before, "eth-usdc");
let mm_btc_after = get_balance(&mm_balance_after, "btc-btc");
let mm_usdc_after = get_balance(&mm_balance_after, "eth-usdc");

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

// Owner outcome.
println!("Owner balance changes:");
print_difference("btc-btc", owner_btc_before, owner_btc_after);
print_difference("eth-usdc", owner_usdc_before, owner_usdc_after);

// Contract outcome.
println!("Contract balance changes:");
print_difference("btc-btc", contract_btc_before, contract_btc_after);
print_difference("eth-usdc", contract_usdc_before, contract_usdc_after);

// Market maker outcome.
println!("Market Maker balance changes:");

```

```
print_difference("btc-btc", mm_btc_before, mm_btc_after);
print_difference("eth-usdc", mm_usdc_before, mm_usdc_after);
}
```

Result:

The swap execution introduces **precision loss**, which accumulates over multiple transactions. When querying the **quote function** in the XYK pool, an **underflow condition** occurs when the `min_price` check is performed in the contract. This triggers a **reversal of previous operations**, disrupting arbitrage, fixed-price orders, and market-making functions.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (7.5)

Recommendation

It is recommended to calculate the price using the total pool values and review the `min_price` verification in the `quote` function to ensure proper precision adjustments.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by modifying the XYK Strategy code, specifically the `quote` and `swap` functions, addressing the precision loss problem. The previously failing POC test is now passing successfully.

This issue was fixed across several commits included in this [pull request](#).

Remediation Hash

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

7.2 ROUNDING ERRORS IN MARKET MAKER AND XYK SWAP CAUSING INSUFFICIENT RETURN ERROR

// HIGH

Description

The Market Maker (MM) swap function rounds the `offer_value` upward (`to_uint_ceil()`), while the XYK pool swap function also rounds upward when calculating `self.y`. This double rounding behavior causes a discrepancy in the expected return amount.

When the final return amount is calculated as `y_original - self.y`, it results in a slightly smaller value than expected. When validated against `ask_amount`, this difference will trigger the `StrategyError::InsufficientReturn` error in the contract.

Code Location

Code of `swap` function from `packages/rujira-rs/src/interfaces/bow/xyk.rs` file:

 Copy Code

```
78 pub fn swap(&mut self, offer_amount: Uint128) -> Result<(Uint128, Decimal), StrategyError> {
79     let k = self.k;
80     self.x = self.x.add(offer_amount);
81     let ratio = Decimal256::from_ratio(self.k, self.x);
82     let y = self.y;
83     // Ceil the ask amount in order to ensure rounding maintains new_k > self.k
84     self.y = Decimal256::one().mul(ratio).to_uint_ceil().try_into()?;
85     let return_amount = y.sub(self.y);
86     let price: Decimal = Decimal::from_ratio(return_amount, offer_amount);
87     self.k = Uint256::from(self.x) * Uint256::from(self.y);
88     ensure!(self.k >= k, StrategyError::Underflow {});
89     Ok((return_amount, price))
90 }
```

Code of `swap` function from `contracts/rujira-fin/src/market_maker.rs` file:

 Copy Code

```
78 fn swap(&mut self, offer: Uint128) -> Result<(Uint128, Uint128), SwapError> {
79     let offer_value = Decimal::from_ratio(offer, 1u128)
80         .mul(self.price)
81         .to_uint_ceil();
82     let pool_value = Decimal::from_ratio(self.total, 1u128)
83         .div(self.price)
84         .to_uint_floor();
85     let res = match self.total.cmp(&offer_value) {
86         Ordering::Greater => {
87             self.total -= offer_value;
88             (offer, offer_value)
89         }
90         // Complete fill
91         _ => {
92             let size = self.total;
93             self.total = Uint128::zero();
94             (pool_value, size)
95         }
96     };
97     self.commitment = res;
98
99
100 }
```

```
Ok(res)
```

```
}
```

Proof of Concept

Scenario:

This test simulates a **full distribution swap** in a **market-making (MM) environment**, utilizing **custom setup functions** and **balance query utilities**.

A large swap operation is executed, aiming to distribute 8,100,000 units of eth-usdc, with 8,000,000 allocated to fixed-price orders and 100,000 to the MM.

After the swap, the test queries remaining user orders and compares balance changes for the owner, contract, and MM to verify the distribution.

NOTE: The `min_price` check from `Quote` function has been skipped in order to continue with this test.

Test:

[Copy Code](#)

```
#[test]
fn test_full_distribution_with_mm() {
    let (mut app, contract, contract_mm, fee_address) = setup_with_mm();

    let owner = app.api().addr_make("owner");
    let funds = vec![
        coin(500_000_000_000, "btc-btc"),
        coin(500_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();
    });

    let owner_balance_before = app.wrap().query_all_balances(owner.clone()).unwrap();
    let mm_balance_before = app.wrap().query_all_balances(contract_mm.clone()).unwrap();
    let contract_balance_before = app.wrap().query_all_balances(contract.clone()).unwrap();

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

```

let funds = vec![
    coin(100_000_000, "btc-btc"),
    coin(100_000_000, "eth-usdc"),
];

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

app.execute_contract(
    user2.clone(),
    contract.clone(),
    &ExecuteMsg::Order(vec![
        (
            Side::Base,
            Price::Fixed(Decimal::from_str("1000.00").unwrap()),
            Uint128::from(2000u128),
        ),
    ]),
    &funds,
)
.unwrap();

app.execute_contract(
    user3.clone(),
    contract.clone(),
    &ExecuteMsg::Order(vec![
        (
            Side::Base,
            Price::Fixed(Decimal::from_str("1000.00").unwrap()),
            Uint128::from(5000u128),
        ),
    ]),
    &funds,
)
.unwrap();

println!("-----> Swap: Full Distribution of 8_100_000 : 8_000_000 for Fixed, 100_000 for MM -----");

let swap_amount = coin(8_100_000, "eth-usdc");

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

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

let orders_user: OrdersResponse = app
    .wrap()
    .query_wasm_smart(contract.clone(), &QueryMsg::Orders{owner:user.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!("User Orders:");
for (i, order) in orders_user.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);
}

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

// Extract the balances for "btc-btc" and "eth-usdc" for each account.

// Owner balances.
let owner_btc_before = get_balance(&owner_balance_before, "btc-btc");
let owner_usdc_before = get_balance(&owner_balance_before, "eth-usdc");
let owner_btc_after = get_balance(&owner_balance_after, "btc-btc");
let owner_usdc_after = get_balance(&owner_balance_after, "eth-usdc");

// Contract balances.
let contract_btc_before = get_balance(&contract_balance_before, "btc-btc");
let contract_usdc_before = get_balance(&contract_balance_before, "eth-usdc");
let contract_btc_after = get_balance(&contract_balance_after, "btc-btc");
let contract_usdc_after = get_balance(&contract_balance_after, "eth-usdc");

//Market Maker balances
let mm_btc_before = get_balance(&mm_balance_before, "btc-btc");
let mm_usdc_before = get_balance(&mm_balance_before, "eth-usdc");
let mm_btc_after = get_balance(&mm_balance_after, "btc-btc");
let mm_usdc_after = get_balance(&mm_balance_after, "eth-usdc");

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

// Owner outcome.
println!("Owner balance changes:");
print_difference("btc-btc", owner_btc_before, owner_btc_after);
print_difference("eth-usdc", owner_usdc_before, owner_usdc_after);

// Contract outcome.
println!("Contract balance changes:");
print_difference("btc-btc", contract_btc_before, contract_btc_after);
print_difference("eth-usdc", contract_usdc_before, contract_usdc_after);

// Market maker outcome.
println!("Market Maker balance changes:");
print_difference("btc-btc", mm_btc_before, mm_btc_after);
print_difference("eth-usdc", mm_usdc_before, mm_usdc_after);

```

```

}
```

Result:

The swap fails due to accumulated precision loss from double rounding, causing the **return amount** to be lower than expected and triggering **StrategyError::InsufficientReturn**, which results in a transaction revert and disrupts liquidity operations. This triggers a **reversal of previous operations**, disrupting arbitrage, fixed-price orders, and market-making functions.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:N/D:N/Y:N (7.5)

Recommendation

It is recommended to ensure consistent rounding behavior in both MM and XYK swaps to mitigate rounding effects and prevent precision loss.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by modifying the XYK Strategy code, specifically the `quote` and `swap` functions, addressing the precision loss problem. The previously failing POC test is now passing successfully.

This issue was fixed across several commits included in this [pull request](#).

Remediation Hash

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

7.3 RUJIRA - LIQUIDITY OVERWRITTEN ON DEPOSIT

// HIGH

Description

The `deposit` function incorrectly replaces the pool's internal balances with the new deposit amounts, rather than adding them to the existing totals.

Specifically, the call to `state.set(x, y)` stores only the current deposit in the pool state, discarding any previous liquidity.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:N (7.5)

Recommendation

It is recommended to update the deposit logic to **accumulate** the new liquidity instead of overwriting.

Remediation Comment

SOLVED: The Rujira team resolved this issue by updating the logic to add the deposited amounts instead of overwriting the existing pool balances.

```
state.set(state.x.add(x), state.y.add(y));
```

 Copy Code

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.