RUJI Trade FIN v1.1 - THORChain / Rujira

Prepared by: **HALBORN** Last Updated 12/08/2025

Date of Engagement: November 12th, 2025 - November 20th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
6	0	0	0	2	4

1. Introduction

`THORChain` engaged Halborn to conduct a security assessment of the `Rujira FIN` contracts, beginning on **November 10th, 2025** and ending on **November 18th, 2025**. This security assessment was scoped exclusively to the updates introduced in version **1.1.0** of `Rujira FIN`, as the core system had already been reviewed in a previous audit. Commit hashes and further details can be found in the **Sources** section of this report.

`Rujira FIN` is a decentralized order book protocol built on top of Thorchain, enabling spot trading of native assets without wrapped tokens. It handles order creation, matching, and settlement fully on-chain.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to verify the security of the updated 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 was to:

- Ensure that the updated smart contract functions operate as intended
- Identify potential security issues introduced or affected by the new logic in version 1.1.0

In summary, Halborn identified a few improvements to further harden the system and reduce the likelihood or impact of potential risks, which have been completely addressed by the `Rujira` `development team`. The most relevant ones were the following:

- `Forbid denom changes during migration unless the contract is fully shut down or enforce a pre-migration check ensuring no active orders or pools exist.`
- `Deduplicate or reject repeated MarketMaker addresses during initialization before storing them.`
- `Replace inv().unwrap() with a checked inversion that errors on zero or tiny prices, and validate MM price inputs before use.`
- `Reject zero or negative tick values during configuration and validate ticks explicitly in Tick::new and Tick::validate.`
- `Explicitly reject Fixed <= 0 before tick validation, add guards to prevent underflow in truncation, and ensure validation returns a contract error instead of panicking.`
- `Replace the intermediate multiplication with amount.multiply_ratio(...) to avoid overflow and add tests with values near Uint128::MAX.`

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 essential to uncover flaws in logic, process, and implementation, automated techniques enhance coverage and quickly detect deviations from best practices.

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 Denoms can be changed during migrate with an active order book
    7.2 Duplicate market makers addresses allow liquidity double-counting
    7.3 Zero tick size collapses price grid and blocks non-zero fixed prices
    7.4 Swap operation panics on zero or tiny marketmaker prices
    7.5 Accepting fixed(0) prices triggers unhandled panic during tick validation
    7.6 Potential uint128 overflow in eitherorboth proportional split

// DOWNLOAD THE FULL REPORT

RUJI TRADE FIN V1.1

SAVE AS PDF

* Use Google Chrome for best results
** Check "Background Graphics" in the print settings if needed

The following phases and associated tools were used during the assessment:

- Research into architecture, version 1.1.0 updates, and overall protocol purpose.
- Manual code read and walkthrough of the updated Rust modules.
- Manual assessment of critical arithmetic operations, rate handling, and overflow/underflow protections.
- Logical review of order matching, bucket aggregation, and MarketMaker calculation flow.
- Validation of safe handling for zero and invalid price conditions.
- Review and verification of integration and unit tests covering the updated logic.

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.

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: 306dc1e

(c) Items in scope:

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

↓ Expand ↓

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

REMEDIATION COMMIT ID:

- 1fceec1
- 8bbc335
- d7499fa
- a9e4350
- d553ef6
- 09d0e39

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DENOMS CAN BE CHANGED DURING MIGRATE WITH AN ACTIVE ORDER BOOK	LOW	SOLVED - 11/26/2025
DUPLICATE MARKET MAKERS ADDRESSES ALLOW LIQUIDITY DOUBLE-COUNTING	LOW	SOLVED - 11/26/2025
ZERO TICK SIZE COLLAPSES PRICE GRID AND BLOCKS NON-ZERO FIXED PRICES	INFORMATIONAL	SOLVED - 11/26/2025

SWAP OPERATION PANICS ON ZERO OR TINY MARKETMAKER PRICES	INFORMATIONAL	SOLVED - 11/26/2025
ACCEPTING FIXED(0) PRICES TRIGGERS UNHANDLED PANIC DURING TICK VALIDATION	INFORMATIONAL	SOLVED - 11/26/2025
POTENTIAL UINT128 OVERFLOW IN EITHERORBOTH PROPORTIONAL SPLIT	INFORMATIONAL	SOLVED - 11/26/2025

7. FINDINGS & TECH DETAILS

7.1 DENOMS CAN BE CHANGED DURING MIGRATE WITH AN ACTIVE ORDER BOOK

// LOW

Description

The `migrate` flow in `rujira-fin` rewrites the entire `Config`, including the denomination fields (`base_denom`, `quote_denom`), without validating whether active pools or open orders currently exist.

Allowing denom changes while there are active positions can lead to severe inconsistencies: funds may be misrouted to incorrect tokens, side detection logic may break due to swapped or undefined denoms, and the system can become permanently bricked.

This vulnerability compromises both **fund safety** and **protocol integrity** during migrations or upgrades.

Code Location

Code of `migrate` function from contracts/rujira-fin/src/contract.rs file:

Copy Code

```rust
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn migrate(deps: DepsMut, _env: Env, msg: InstantiateMsg) -> Result<Response, ContractError> {
    set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;
    let config = Config::new(deps.api, msg)?;
    config.validate(deps.as_ref())?;
    config.save(deps.storage)?;
    Ok(Response::default())
}
```

BVSS

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

Recommendation

It is recommended to **forbid denom changes during migration** unless the protocol is in a fully inactive state. Specifically, require an explicit shutdown procedure that ensures there are **no active orders, pools, or Market Makers** before allowing denom updates, or alternatively, restrict migration to non-denom configuration fields only.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by disabling the Config replacement during the migration.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/1fceec11adbeee23658c3a41bcea012c9b9592fc

7.2 DUPLICATE MARKET MAKERS ADDRESSES ALLOW LIQUIDITY DOUBLE-COUNTING

// LOW

Description

The `MarketMakers` constructor accepts a `Vec<String>` of Market Maker addresses and validates only their format, without checking for duplicates.

As a result, supplying the same address multiple times creates redundant entries in the internal `contracts` list. This causes **liquidity to be double-counted**, skewing proportional fills within equal-rate buckets, biasing commitment aggregation toward duplicated Market Makers, and increasing gas consumption.

Code Location

Code of `new` function from contracts/rujira-fin/src/market_makers.rs file:

Copy Code

```rust
pub fn new(api: &dyn Api, market_makers: Vec<String>) -> StdResult<Self> {
    Ok(Self {
        contracts: market_makers
            .into_iter()
            .map(|x| api.addr_validate(&x))
            .collect::<StdResult<Vec<Addr>>>()?,
    })
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:C/I:H/D:N/Y:N (2.4)

Recommendation

It is recommended to **deduplicate** Market Maker addresses during initialization by inserting validated entries into a set before collection, or to reject duplicates explicitly with an appropriate validation error.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by replacing the Vector by a `Set`.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/8bbc335c9bff642a7b95fede479280382bc24a5d

7.3 ZERO TICK SIZE COLLAPSES PRICE GRID AND BLOCKS NON-ZERO FIXED PRICES

// INFORMATIONAL

Description

The `Tick` logic accepts a tick size of `0`. With `tick == 0`, the truncation routines collapse any non-zero `Decimal` price to `0` (or otherwise behave nonsensically), which makes `validate()` only accept `Fixed(0)` and reject any `Fixed > 0`.

This breaks normal order placement and distorts rate truncation wherever tick-based rounding is assumed sane.

Code Location

Code of `Tick::validate` and `Tick::truncate_floor` functions from packages/rujira-rs/src/interfaces/fin/tick.rs file:

Copy Code

```rust
pub fn new(size: u8) -> Self {
    Self(size)
}

pub fn validate(&self, v: &Decimal) -> Result<(), TickError> {
    if v == self.truncate_floor(v) {
        return Ok(());
    }
    Err(TickError::Invalid { v: *v })
}

pub fn truncate_floor(&self, v: &Decimal) -> Decimal {
    self.do_truncate(v, |x, y| x.mul_floor(y))
}

pub fn truncate_ceil(&self, v: &Decimal) -> Decimal {
    self.do_truncate(v, |x, y| x.mul_ceil(y))
}

fn do_truncate<F>(&self, v: &Decimal, fn_trunc: F) -> Decimal
where
    F: Fn(Uint128, Decimal) -> Uint128,
{
    let int = v.numerator();
    let len = int.to_string().as_str().bytes().len() as u32;
    let decimals: u32 = len - self.0 as u32;
    let pow = Uint128::from(10u128).pow(decimals);
    let truncated = fn_trunc(Uint128::one(), Decimal::from_ratio(int, pow));
    Decimal::from_ratio(truncated * pow, v.denominator())
}
```

BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.0)

Recommendation

It is recommended to **disallow** `tick == 0` by enforcing a **minimum** `tick ≥ 1` (or the smallest positive unit representable) in configuration (e.g., in `Config::validate`).

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by disallowing `tick`=0.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/d7499faef102f58595c1ef90372a7e3d61eeb086

7.4 SWAP OPERATION PANICS ON ZERO OR TINY MARKETMAKER PRICES

// INFORMATIONAL

Description

If a Market Maker returns a price of `0` or an extremely small value, the code path computing `1 / price` via `Decimal::inv().unwrap()` in the Base-side rate calculation **panics** during execution.

This occurs when constructing or updating MarketMaker offers: for `Side::Base`, the inverse of the quoted price is computed without validating that it is non-zero or within a sane range.

A misconfigured or malicious Market Maker can therefore trigger a **runtime panic**, halting the swap or arbitrage flow and degrading system availability.

Code Location

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

```
fn attributes(&self) -> Vec<Attribute> {
    let price = match self.side {
        Side::Base => self.price.inv().unwrap().to_string(),
        Side::Quote => self.price.to_string(),
    };
    vec![
        Attribute::new("price", format!("{}:{price}", self.contract)),
        Attribute::new("side", self.side.to_string()),
    ]
}
```

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

```
match res {
    Ok(Some(res)) => {
        self.bid += Decimal::from_ratio(res.size, 1u128)
            .mul(res.price)
            .to_uint_ceil();
        self.ask += res.size;
        self.data = res.data;
        let price = res.price;
        self.last_price = Some(price);
        // Rate is used for presentation, and for sorting and grouping `Iter<impl Swappable>`
        // Truncate to "worse" price so that execution price is fractionally better than
        // the price rendered
        let rate = match self.side {
            Side::Base => self.tick.truncate_ceil(&res.price.inv().unwrap()),
            Side::Quote => self.tick.truncate_floor(&res.price),
        };
```

BVSS

AO:S/AC:L/AX:H/R:N/S:U/C:N/A:C/I:N/D:N/Y:N (0.7)

Recommendation

It is recommended to replace `inv().unwrap()` with a **checked inversion** that safely returns an error on zero or near-zero input. Additionally, enforce **price validation** (non-zero, non-negative, within a reasonable range) before any MarketMaker rate computation. This prevents misconfigured or malicious Market Makers from triggering panics and ensures graceful error handling in all price paths.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue through implementation of appropriate input validation.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/a9e43508afeb067c2401a8cb68c3f1bf192652df

7.5 ACCEPTING `FIXED(0) PRICES TRIGGERS UNHANDLED PANIC DURING TICK VALIDATION

// INFORMATIONAL

Description

Submitting an order with `Price::Fixed(0)` causes a **panic** during tick-based truncation instead of returning a handled contract error.

In the execution path, `order_manager::execute_orders` calls `tick.validate(&x)` for fixed prices; the validation invokes the truncation routine (e.g., `do_truncate` / `truncate_floor`) which performs unsigned arithmetic that underflows when `v == 0` , aborting the transaction.

Although the order is not persisted (no lasting state corruption), user input can trigger per-request DoS and yields inconsistent error semantics (panic vs. `ContractError`).

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 **explicitly reject** `Fixed = 0` at input level (prior to tick validation) with a clear `ContractError` (e.g., `InvalidPrice`).

Remediation Comment

SOLVED: The **Rujira team** has solved this issue through the implementation of the proper input validation.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/d553ef6c9d7063ec3aba30ab12e1ba9679c2603e

7.6 POTENTIAL UINT128 OVERFLOW IN EITHERORBOTH PROPORTIONAL SPLIT

// INFORMATIONAL

Description

The proportional split logic in `EitherOrBoth::swap` calculates the offer for side A using

Copy Code

```
Decimal::from_ratio(a.total() * amount, a.total() + b.total())
```

This operation multiplies two `Uint128` values (`a.total()` and `amount`) **before** constructing the ratio, which can overflow at the intermediate step even when the final ratio result would fit within `Uint128`.

Although in normal market conditions amounts are unlikely to approach `u128::MAX`, this arithmetic pattern constitutes a **latent overflow hazard** that can lead to panics or transaction reverts under adversarial or stress-testing inputs.

Because this code path executes as part of swap execution, such an overflow could cause **unexpected failures or DoS in extreme liquidity or large-trade scenarios**.

Code Location

Code of `EitherOrBoth::swap` implementation from packages/rujira-rs/src/exchange/swappable.rs file:

Copy Code

```
fn swap(&mut self, amount: Uint128) -> Result<(Uint128, Uint128), SwapError> {
    match self {
        EitherOrBoth::Both(a, b) => {
            let offer_a = Decimal::from_ratio(a.total() * amount, a.total().add(b.total()))
                .to_uint_floor();
            let offer_b = amount - offer_a;
            let (consumed_offer_a, consumed_bids_a) = a.swap(offer_a)?;
            let (consumed_offer_b, consumed_bids_b) = b.swap(offer_b)?;
```

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 eliminate the intermediate `Uint128` product and use a safe rational multiplication instead, which performs the same proportional calculation without risking overflow. For example:

Copy Code

```
amount.multiply_ratio(a.total(), a.total() + b.total()),
```

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by using the `checked_multiply_ratio` operation.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/09d0e395b0ab37a9ad0c16da29cc7491f84fd81a

---

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.