FAILSAFE

Prepared for
Rujira

Prepared by
ret2basic.eth
y4y
FailSafe

13 February 2026

Rujira Fin

Smart Contract Audit Report

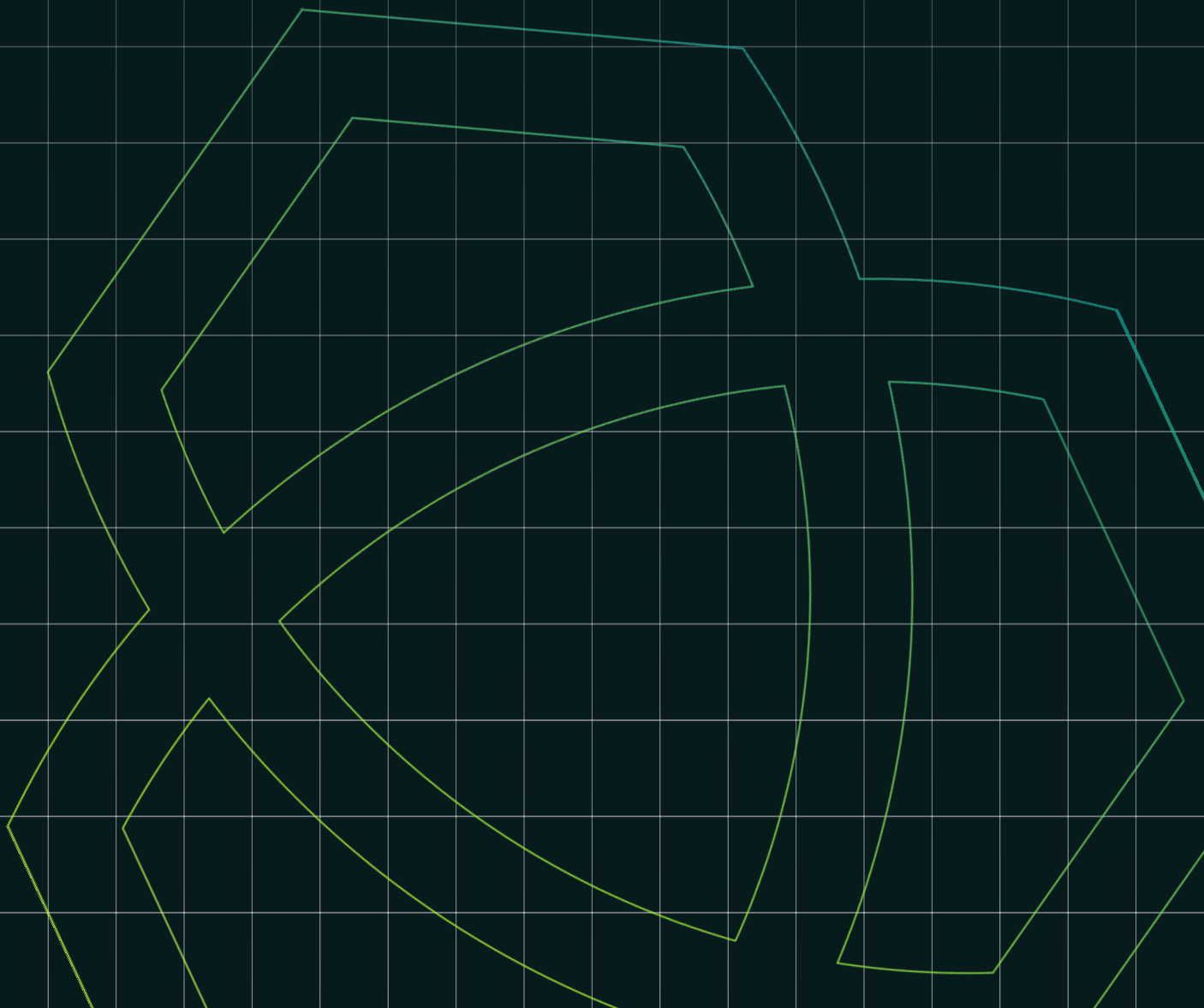

Table of Contents

Executive Summary

FailSafe was engaged by Rujira Fin to conduct a smart contract audit on its THORChain blockchain implementation. Our elite team of security experts delved deeply into the intricacies of the smart contracts to provide a comprehensive security review. The audit process was meticulous, leveraging both automated tools and manual code analysis to identify potential vulnerabilities. Our objective was to ensure the integrity, security, and reliability of Rujira Fin's smart contracts, and to provide actionable recommendations to enhance their security posture.

During the audit, we identified several critical vulnerabilities that could have had significant implications if left unaddressed. The most severe findings involved the potential for direct fund theft through manipulation of zero-rate pools and unauthorized range transfers via spoofing. These vulnerabilities could have allowed attackers to steal assets directly from users or manipulate market conditions to their advantage. Additionally, we discovered issues related to market operations that could lead to denial of service, such as underflows and unbounded iterations. These findings highlight the importance of rigorous input validation and robust transaction handling to prevent exploitation by malicious actors.

We commend the Rujira Fin development team for their proactive approach and dedication to security. Their swift response to our findings and commitment to resolving critical vulnerabilities demonstrate a strong commitment to safeguarding user funds and maintaining the integrity of their platform. By addressing these concerns and implementing our recommendations, Rujira Fin has significantly strengthened its security posture. We encourage the team to continue their diligent efforts in maintaining a secure and resilient blockchain environment, ensuring trust and confidence among their users.

Project Details

Project	Rujira Fin
Website	https://rujira.network/
Repository	https://gitlab.com/thorchain/rujira/-/tree/fin/v1.2/contracts/rujira-fin?ref_type=heads
Blockchain	THORChain
Audit Type	Smart Contract Audit Report
Initial Commit	9e78fabab7d5441743af3e925074beb79912be86
Final Commit	TBD
Timeline	29 January 2026 - 13 February 2026
	Final Report: 13 February 2026

Structure & Organization of The Security Report

Issues are tagged as “Open”, “Acknowledged”, “Partially Resolved”, “Resolved” or “Closed” depending on whether they have been fixed or addressed.

- Open: The issue has been reported and is awaiting remediation from developer team.
- Acknowledged: The developer team has reviewed and accepted the issue but has decided not to fix it.
- Partially Resolved: Mitigations have been applied, yet some risks or gaps still remain.
- Resolved: The issue has been fully addressed and no further work is necessary.
- Closed: The issue is deemed no longer pertinent or actionable.

Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical	The issue affects the platform in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.
High	The issue affects the ability of the platform to compile or operate in a significant way.
Medium	The issue affects the ability of the platform to operate in a way that doesn’t significantly hinder its behavior.
Low	The issue has minimal impact on the platform’s ability to operate.
Info	The issue is informational in nature and does not pose any direct risk to the platform’s operation.

Methodology

Threat Modelling

We will employ a threat modelling approach to identify potential attack vectors and risks associated with the smart contract(s). This involves:

1. Asset Identification: Enumerating the critical assets within the smart contract(s), such as tokens, sensitive data, access controls, and more.
2. Threat Enumeration: Identifying potential threats such as reentrancy, integer overflow/underflow, denial of service, and more.
3. Vulnerability Assessment: Assessing vulnerabilities in the context of the smart contract(s) and its interaction with external components.
4. Risk Prioritization: Prioritizing identified threats based on their severity and potential impact.

Manual Code Review

Our manual analysis involves an in-depth review of the smart contract(s) source code, focusing on:

1. Code Review Line-by-line examination to detect vulnerabilities and ensure compliance with best practices.
2. Logic Analysis: Analyzing the smart contract(s) Business logic for vulnerabilities and inconsistencies.
3. Gas Optimization: Identifying areas for gas optimization and efficiency improvements.
4. Access Control Review: Ensuring proper access controls and permission management.
5. External Dependencies: Assessing the security implications of external dependencies or oracles.

Functional Testing in Hardhat/Foundry

We will perform functional testing using Hardhat/Foundry to ensure the correctness and reliability of the smart contract(s). This includes:

1. Functional Testing: Writing comprehensive tests to cover various functionalities and edge cases.
2. Integration Testing: Verifying the interaction of smart contract(s) with other components.
3. Deployment Verification: Ensuring the correctness of smart contract(s) deployment.

Fuzzing and Invariant Testing

If deemed necessary based on the complexity and criticality of the smart contract(s), we will perform fuzzing and invariant testing to identify vulnerabilities that might not be caught through conventional methods. This includes:

1. Fuzz Testing: Employing fuzzing techniques to generate invalid, unexpected, or random inputs to trigger potential vulnerabilities.
2. Invariant Testing: Verifying invariants and properties to ensure the correctness and consistency of the smart contract(s) across various scenarios.

Edge Cases Scenarios Coverage

Our audit will thoroughly cover a wide spectrum of edge cases, including but not limited to:

1. Extreme Inputs: Testing with extreme and boundary inputs to assess resilience.
2. Exception Handling: Evaluating how the contract(s) handle exceptional scenarios.
3. Concurrency: Assessing the contract(s) behaviour in concurrent or simultaneous interactions.
4. Non-Standard Scenarios: Analyzing non-standard use cases that might impact contract(s) behaviour.

Reporting and Recommendations

A thorough description of the issue, highlighting the potential impact on the system.

1. The location within the codebase where the issue is found.
2. A clear explanation of the vulnerability, its root cause, and its potential exploitation.
3. Code snippets or detailed instructions on how to address the vulnerability.
4. Best practices and coding guidelines to prevent similar issues in the future.
5. We will suggest improvements in the overall system architecture or design, if relevant.
6. Wherever applicable, we'll include a PoC to demonstrate issue severity, aiding effective mitigation.

Report Generation

1. Document all findings, including identified vulnerabilities, their severity, and potential impact.
2. Provide clear and actionable recommendations for addressing security issues.

Remediation Support

1. Collaborate with the project's development team to address and remediate identified vulnerabilities.
2. Review and validate code changes and security fixes.

Final Assessment

Re-evaluate the project's security posture after remediation efforts to ensure vulnerabilities have been adequately addressed.

In-scope

- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/ranges/range.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/contract.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/order_pool/pool.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/order_pool/order_manager.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/ranges/scl.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/ranges/iter.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/ranges/execute.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/order_pool/order.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/ranges/tests.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/market_maker/context.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/config.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/ranges/range_offer.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/order_pool/pool_key.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/market_maker/iter.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/testing/xyk.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/ranges/events.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/ranges/context.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/market_maker/offer.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/swap_iter.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/error.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/market_maker/tests.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/order_pool/query.rs
- rujira-fin-v1.2-contracts-rujira-fin/contracts/rujira-fin/src/order_pool/execute.rs

Summary of Findings

Severity	Total	Open	Acknowledged	Partially Resolved	Resolved
Critical	2	-	-	-	2
High	1	-	-	-	1
Medium	5	-	1	-	4
Low	1	-	-	-	1
Info	1	-	-	-	1
Total	10	0	1	0	9

#	Findings	Severity	Status
1	Oracle(-10000) Zero-Rate Pool Enables Direct Theft of Swap User Funds	Critical	Resolved
2	Unauthorized Range Transfer via Arb/DoRange Sender Spoofing	Critical	Resolved
3	MarketMaker commit() Fee Deduction Creates Impossible BOW min_return	High	Resolved
4	bid_pool::distribute_full Consumed-Offer Exceeds Passed Offer — Swapper Underflow DoS	Medium	Resolved
5	LP sandwich on ‘RangeMsg::Create‘ due to lack of slippage control	Medium	Resolved
6	MarketMaker Netting Underflow Causes DoS	Medium	Resolved
7	Tick Change Strands Fixed-Price Orders, Locking User Funds	Medium	Resolved
8	Unbounded Orderbook Iteration Can Cause DoS	Medium	Acknowledged
9	Recursive RangeOfferIter Can DoS with Many Empty Ranges	Low	Resolved
10	Range Iterator Infinite Loop When range_delta == 0	Info	Resolved

Finding 1: Oracle(-10000) Zero-Rate Pool Enables Direct Theft of Swap User Funds

Severity: Critical

Status: Resolved

Description:

`Premiumable::adjust` maps `bps == -10000` to a rate multiplier of exactly zero. An attacker can exploit the Quote-side order creation path — which does not call `inv()` and therefore does not panic — to store a persistent pool with `rate = 0` in contract state. This pool then silently absorbs subsequent swap users' offer tokens for zero return, allowing the attacker to claim those stolen tokens while keeping their original deposit intact.

Root Cause

`Premiumable::adjust` in `packages/rujira-rs/src/premium.rs`:

```
i16::MIN..=-1 => self * Decimal::from_ratio(10000u16 - bps.unsigned_abs(), 10000u16)
```

For `bps == -10000`: numerator = `10000 - 10000 = 0` → rate = `oracle * 0 = 0`.

For `bps < -10000`: overflow-checks are enabled in release profile (`Cargo.toml`), so the `u16` subtraction panics — only `bps == -10000` reaches the zero-rate path without reverting.

The Quote-Side Asymmetry (Why Order Creation Succeeds)

In `execute_new_order` (order_manager.rs):

```
let opposite = side.other();
let mut swapper = Swapper::new(
    ...,
    SwapRequest::Limit {
        price: match opposite {
            Side::Base => pool.rate(),             // ← used when side == Quote
            Side::Quote => pool.rate().inv().unwrap(), // ← used when side == Base
        },
        ...
    },
    ...
);
```

- `Side::Base` path: `opposite = Quote` → calls `pool.rate().inv().unwrap()` → rate is 0 → `inv()` returns `None` → `.unwrap()` panics → tx reverts → order NOT stored.
- `Side::Quote` path: `opposite = Base` → calls `pool.rate()` directly → returns 0 → no panic. The swapper runs with `limit = 0`, immediately breaks on all Base-side pools (`achieved > 0 = limit`), and returns with `consumed_offer = 0`, `remaining_offer = target`. Then `pool.create_order(storage, ..., target)` stores the order. The rate-0 pool is now persistently in state.

How the Stored Rate-0 Pool Steals Funds During Swaps

When any user swaps by sending base tokens (→ `side = Side::Quote`), `Pool::iter` includes the stored rate-0 pool. In `Pool::swap` (SwapItem impl, pool.rs):

```
let rate = match self.side {
    Side::Base => self.rate.inv().unwrap(),
    Side::Quote => self.rate,  // ← 0, no inv(), no panic
};
let res = self.pool.distribute(offer.into(), &Decimal256::from(rate))?;
```

`distribute(remaining_offer, rate=0)` calls `distribute_partial(bids_value=0, offer)`:

- `consumed_offer = offer` → victim's entire remaining offer consumed
- `consumed_bids = 0` → attacker's bid untouched
- `ratio = 1 - 0/total = 1` → pool product unchanged, attacker's bid amount stays the same
- Pool `sum` updated with `offer / total` → attacker's `filled` amount increases by victim's tokens

Back in `Swapper::swap` (swapper.rs):

```
if let SwapRequest::Limit { price: limit, .. } = self.req {
    if !bids.is_zero() {  // bids == 0 → SKIPPED
        ...
    }
}
self.context = next_context;  // COMMITTED
```

The `!bids.is_zero()` guard skips the limit-price check whenever bids is zero. For `Yolo`/`Min` requests, there is no per-step guard at all. The step is unconditionally committed.

After the loop, `SwapRequest::Yolo` performs no return check. `SwapRequest::Min` only reverts if `returned < min_return` — but if legitimate pools already filled enough to meet the minimum, the rate-0 theft goes through silently.

Attacker Recovery

The attacker sends a subsequent `Order` message targeting the same `(Side::Quote, Price::Oracle(-10000))` pool:

1. `maybe_withdraw` → `pool.claim_order` claims `filled` tokens → returned in `ask` denom (base tokens). These are the victim's stolen tokens.
2. Retract to `target = 0` → `pool.retract_order` returns the attacker's original quote tokens, since `consumed_bids = 0` during the theft.

Net result: attacker keeps their quote tokens AND gains the victim's base tokens.

Impact:

- **Direct fund theft**: attacker steals base tokens from every swap user whose transaction reaches the rate-0 pool.
- **Persistent**: the pool stays in state indefinitely, affecting all future `Side::Quote` swaps until governance intervention.
- **No special privileges required**: any user can create the malicious order.
- **Yolo swaps fully exploitable**: no return check protects the victim.
- **Min swaps partially exploitable**: theft succeeds whenever the min_return is met by earlier legitimate pools.

Proof of Concept:

Step 1 — Attacker plants the rate-0 pool

```
// Attacker sends 1_000_000 quote tokens alongside:
ExecuteMsg::Order((
    vec![
        (
            Side::Quote,                    // ← Quote side, NOT Base
            Price::Oracle(-10000),          // ← rate = oracle * 0 = 0
            Some(Uint128::from(1_000_000u128)),
        ),
    ],
    None,
))
```

Execution trace (succeeds, no panic):

```
ExecuteMsg::Order
  → Arb { then: DoOrder }
    → DoOrder
      → execute_orders
        → Pool::load(Price::Oracle(-10000), Side::Quote, oracle)
            → price.to_rate(oracle) = 0
        → execute_new_order(side=Quote)
            → opposite = Base
            → limit price = pool.rate() = 0     (no inv(), so won't revert)
            → Swapper runs, breaks immediately on all Base pools
            → swap result: consumed=0, remaining=1_000_000
            → pool.create_order(storage, ..., 1_000_000)  // STORED IN STATE
```

Result: pool at `(Side::Quote, PoolType::Oracle, Price::Oracle(-10000))` with `total = 1_000_000` and `rate = 0` is now in `POOLS` + `BID_POOLS`.

Step 2 — Victim swaps (sends base tokens)

```
// Victim sends 500 base tokens alongside:
ExecuteMsg::Swap(SwapRequest::Yolo {
    to: None,
    callback: None,
})
```

Execution trace:

```
ExecuteMsg::Swap → Arb { then: DoSwap }
  → DoSwap → execute_swap
    → side = Side::Quote  (ask_side of base token)
    → Pool::iter(storage, Side::Quote, oracle)
      → [... legitimate pools ...] [rate-0 pool at end]
    → Swapper iterates:
      → <legitimate pools fill some or all of the offer>
      → rate-0 pool hit with remaining_offer:
        → Pool::swap: rate = 0 (Side::Quote, no inv())
        → distribute(remaining_offer, 0)
          → distribute_partial(bids_value=0, remaining_offer)
          → consumed_offer = remaining_offer  ← ALL CONSUMED
          → consumed_bids = 0                 ← NOTHING RETURNED
        → Swapper: bids=0, limit check SKIPPED, step COMMITTED
        → remaining_offer = 0, returned += 0
    → Yolo: no return check → Ok
    → Victim receives 0 for the stolen portion
```

Step 3 — Attacker extracts stolen funds

```
// Attacker resubmits with target = 0 to claim + retract:
ExecuteMsg::Order((
    vec![
        (
            Side::Quote,
            Price::Oracle(-10000),
            Some(Uint128::from(0u128)),  // retract all
        ),
    ],
    None,
))
```

```
→
 execute_existing_order
  → maybe_withdraw: bid.filled > 0 → claims victim's base tokens
  → retract to target=0: returns 1_000_000 quote tokens (bid was never consumed)
```

Attacker wallet after: 1,000,000 quote tokens + victim's base tokens.

Numeric Example

Actor	Before	Action	After
Attacker	1M USDC	Order(Quote, Oracle(-10000), 1M)	0 USDC (in pool)
Victim	500 BTC	Swap(Yolo), sends 500 BTC	0 BTC (stolen)
Attacker	—	Order(Quote, Oracle(-10000), 0)	1M USDC + 500 BTC

Remediation:

1. **Reject** `bps <= -10000` in `OrderManager::execute_orders` (or in `Price::to_rate`) before the pool is loaded.
2. **Guard** `distribute_partial` against `bids_value == 0`: return `Ok(DistributionResult::default())` when rate produces zero bids — zero-rate distributions should never consume offer.
3. **Guard** `Swapper::swap` against zero-bids steps: skip/error when `bids == 0 && offer > 0` instead of silently committing.
4. **Defense-in-depth**: replace `inv().unwrap()` with checked error handling in all execution paths.

Discussion:

> *Developer:*
>
> Hey, just wanted to let you know that we've sorted out the issue flagged in the audit. The fix has been implemented, and you can check it out in the commit here: https://gitlab.com/thorchain/rujira/-/commit/e9beff4a1c2a12eac3eca9df62014861b1e25370. If there's anything else you need from us or any further questions, just give us a shout!
>
> Fix URL: https://gitlab.com/thorchain/rujira/-/commit/e9beff4a1c2a12eac3eca9df62014861b1e25370

> *Auditor:*
>
> Great news! We've seen that you addressed the concern with a fix in place. We'll take a look at the commit you shared to verify everything's sorted. If we have any feedback or further questions after reviewing, we'll get back to you. Thanks for the quick turnaround on this!

Finding 2: Unauthorized Range Transfer via Arb/DoRange Sender Spoofing

Severity: Critical

Status: Resolved

Description:

The `ExecuteMsg::Arb` entry point accepts an arbitrary `then` message. Internal handlers (`DoRange`, `DoOrder`, `DoSwap`) only check that the caller is the contract itself and then trust the embedded `sender/recipient`. This allows spoofing of user identity for internal actions, enabling direct theft via ranges and unauthorized order management.

In `execute` (contract entry), the `DoRange` branch only enforces `info.sender == env.contract.address`.

- The `RangeMsg::Transfer` path in `execute_range` checks `sender == range.owner`.
- Because `sender` is passed as a parameter rather than derived from `info.sender`, a user can call `ExecuteMsg::Arb { then: Some(DoRange((victim, RangeMsg::Transfer{ idx, to })) ) }`.
- `ExecuteMsg::Arb` does not authenticate the `then` message against the original caller, so the spoofed `sender` passes the ownership check.

The same unauthenticated `then` mechanism applies to `DoOrder((recipient, ...))`, allowing arbitrary modification/retraction of another user's orders.

Impact:

Attackers can steal ownership of arbitrary ranges and then withdraw/close/claim to drain assets (critical theft). They can also cancel or resize other users' orders (griefing/market manipulation).

Source:

rujira-fin-v1.2/contrats/rujira-fin/src/contract.rs, execute() rujira-fin-v1.2/contrats/rujira-fin/src/ranges/execute.rs, execute_range()

Proof of Concept:

1. Assume a victim owns range `idx = 7`.
2. Attacker submits `ExecuteMsg::Arb` with a crafted `then` message:

```
ExecuteMsg::Arb {
  then: Some(to_json_binary(&ExecuteMsg::DoRange((
    victim_addr,
    RangeMsg::Transfer { idx: 7u128.into(), to: attacker_addr.to_string() }
  )))?)
}
```

1. The contract validates only that `info.sender` is the contract itself and then trusts the embedded `victim_addr`.
2. `RangeMsg::Transfer` succeeds because `sender == range.owner`, and ownership moves to the attacker.

Remediation:

Remove the embedded sender/recipient parameters from internal handlers and derive ownership checks from the original caller. Alternatively, store and verify the authenticated original sender inside Arb and reject any mismatched internal call.

Discussion:

> *Developer:*
>
> Hey, we really appreciate you catching that issue! We've already taken care of it by implementing a fix, and we've migrated the changes to production. You can check out the details in our latest commit here: https://gitlab.com/thorchain/rujira/-/commit/6a02edb25c9df83c550cceb7158466c6009a54d1. Thanks again for the heads up!
>
> Fix URL: https://gitlab.com/thorchain/rujira/-/commit/6a02edb25c9df83c550cceb7158466c6009a54d1

> *Auditor:*
>
> We found an issue during our review and flagged it for the development team. It looks like they've been quick to address the concern by pushing a fix and updating their production environment. It's always great to see such prompt action when vulnerabilities are identified!

Finding 3: MarketMaker commit() Fee Deduction Creates Impossible BOW min_return

Severity: High

Status: Resolved

Description:

In `MarketMakerContext::commit()`, the AMM fee is deducted from the `offer` tokens before sending them to the BOW contract, but `min_return` is set to the original unmodified `ask` amount. Because market maker prices are derived directly from BOW's own `Quote` query (which already embeds BOW's internal fee), there is zero margin to absorb the additional `fee_amm` deduction. BOW receives fewer input tokens but must meet the unchanged output target, causing every BOW swap to revert when `fee_amm > 0`.

Market maker offer prices originate from `bow::QueryMsg::Quote`, which computes an XYK swap and deducts BOW's own fee before returning the price. `MarketMakerIter::query_next()` (iter.rs) stores this price directly as `MarketMakerOffer::price`. When takers fill against the offer, `MarketMakerOffer::swap()` (offer.rs) uses that price to compute the `(offer, ask)` pair committed to the shared context. The `ask` therefore represents exactly what BOW should return for the full `offer`.

In context.rs, the `commit()` function:

1. Iterates compiled `(offer, ask)` pairs produced by `compile()`
2. Deducts `fee_amm` from `offer`:

```
let offer_fee = coin(
    offer.amount.multiply_ratio(fee.numerator(), fee.denominator()).u128(),
    offer.denom.clone(),
);
let offer = coin(offer.amount.sub(offer_fee.amount).u128(), offer.denom);
```

3. Sends the reduced offer to BOW but keeps `ask` (the original pre-fee amount) as `min_return`:

```
messages.push(CosmosMsg::Wasm(WasmMsg::Execute {
    contract_addr: addr.to_string(),
    msg: to_json_binary(&bow::ExecuteMsg::Swap {
        min_return: ask.clone(),   // <-- NOT adjusted for fee
        ...
    })?,
    funds: vec![offer.clone()],    // <-- reduced by fee
}));
```

The existing test `test_market_maker_commit_fee` in tests.rs confirms:

- Two Quote-side swaps: 1000 @ 1.25 = 1250, and 1000 @ 1.5 = 1500.
- After compile: `offer = 2000 BTC`, `ask = 2750 USDC`.

- With 5% fee: BOW receives 1900 BTC but must return ≥ 2750 USDC.
- At the original weighted rate of 1.375, 1900 input yields ~2612 USDC < 2750. BOW reverts.
- The test comment reads *"with a correct min return"*, indicating the developer believed this behavior was intended.

The test only asserts message construction (passes), but in on-chain execution the BOW `min_return` enforcement causes the transaction to revert. The reduced input cannot satisfy the original `min_return` because the MM quoted price was already the best rate BOW can offer for that swap size.

Impact:

When fee_amm > 0 (the expected production configuration, typically Decimal::permille(5) / 0.5%), every swap routed through market makers reverts. This disables the entire BOW liquidity layer — one of the three core liquidity sources — severely degrading orderbook depth and execution quality. All user swaps, arb operations, order placements, and range operations that trigger market maker fills will fail.

Remediation:

Adjust `min_return` proportionally to the fee deducted from the offer:

```
let ask = coin(
    ask.amount.sub(ask.amount.multiply_ratio(fee.numerator(), fee.denominator())).u128(),
    ask.denom,
);
```

Discussion:

> *Developer:*
>
> Hey, just wanted to update you that we've addressed the issue with the FailSafe Admin. We've mitigated the problem and acknowledged it on our end. Let us know if there's anything else you need from us.

> *Auditor:*
>
> Great to hear that you've taken care of the FailSafe Admin issue. We'll check out the mitigation steps you've implemented and get back to you if we have any more questions or need further clarification. Thanks for your quick response!

Finding 4: bid_pool::distribute_full Consumed-Offer Exceeds Passed Offer — Swapper Underflow DoS

Severity: Medium

Status: Resolved

Description:

`bid_pool::Pool::distribute` uses a `+1` tolerance check to decide whether to fully consume the pool via `distribute_full`. The full-consumption path recomputes `consumed_offer` from `total / rate` rather than capping it at the caller's original `offer`, which can yield a `consumed_offer` greater than the offer that was passed in. The `Swapper` then executes `self.remaining_offer -= offer` with the inflated value, underflowing a `Uint128` and panicking. This is reachable during `execute_swap` against thinly-filled order tick pools at fractional rates, causing per-transaction reverts for affected users.

Note: the automatic arbitrage (`Arb`) path is **not** affected because it calls `SwapItem::swap` directly (not through `Swapper`) and guards profit checks with `checked_sub`, handling inflated values gracefully. Therefore this bug cannot cause a persistent market-wide DoS.

Root cause

In `packages/rujira-rs/src/bid_pool/pool.rs`, `distribute`:

```
pub fn distribute(
    &mut self,
    offer: Uint256,
    rate: &Decimal256,
) -> Result<DistributionResult, BidPoolError> {
    // ...
    let bids_value = offer.mul_floor(*rate);
    // +1 tolerance check:
    if bids_value + Uint256::one() >= self.total {
        return self.distribute_full(rate);  // <-- ignores `offer`
    }
    self.distribute_partial(bids_value, offer)
}
```

When `bids_value` is exactly 1 unit short of `self.total` (which commonly happens due to integer truncation of `offer * rate`), the +1 check passes and `distribute_full` is called. But `distribute_full` does not receive the original `offer`:

```
fn distribute_full(&mut self, rate: &Decimal256) -> Result<DistributionResult, BidPoolError> {
    let consumed_offer = self
        .total
        .multiply_ratio(rate.denominator(), rate.numerator());
    // ...
    Ok(DistributionResult {
        consumed_offer,  // <-- can exceed original offer
        consumed_bids: self.total,
        snapshots,
    })
}
```

`consumed_offer = total / rate`. Because `total` was compared against `bids_value + 1` (not `bids_value`), the back-calculation can produce a value larger than the caller's `offer`.

Flow to panic

1. `Pool::swap` (at `contracts/rujira-fin/src/order_pool/pool.rs`) calls `self.pool.distribute(offer.into(), &rate)` and returns `res.consumed_offer` as the offer amount consumed.

2. `Swapper::swap` (at `packages/rujira-rs/src/exchange/swapper.rs`) does:

```
self.remaining_offer -= offer;  // Uint128 subtraction panics on underflow
```

   If `offer > remaining_offer`, this panics.

Why the Arb path is safe

The `Arb` mechanism uses a separate code path from `Swapper`. In `Arbitrage::arbitrage` (at `packages/rujira-rs/src/exchange/arb.rs`), the arber calls `SwapItem::swap` directly on `RootItem` (a nested `EitherOrBoth<..>` structure). The `EitherOrBoth::swap` implementation only **adds** consumed offers from sub-items — it never subtracts `consumed_offer` from the input `amount`. The inflated value simply propagates upward to the profit check:

```
match (b.1.checked_sub(a.0), a.1.checked_sub(b.0)) {
    (Ok(a), Ok(b)) => { /* commit profitable arb */ }
    _ => Ok(None),  // graceful exit, no panic, no state commit
}
```

The `checked_sub` catches any overflow and returns `Ok(None)`, causing `Arber::run` to break. No panic occurs and no state is committed.

Concrete example

Parameter	Value
`pool.total`	`10`
`rate`	`0.1` (= numerator 10^17, denominator 10^18)
Swapper `remaining_offer`	`99`
`bids_value = 99 * 0.1`	`9` (truncated)
`bids_value + 1 = 10 >= 10`	`true` → routes to `distribute_full`
`consumed_offer = 10 / 0.1`	`100`
`remaining_offer - consumed_offer`	`99 - 100` → underflow panic

In this scenario, the pool contains 10 units at rate 0.1. A swap with 99 offer arrives. The integer-truncated `bids_value` of 9 plus the +1 tolerance equals `total`, so `distribute_full` fires. It back-computes `consumed_offer = 100`, exceeding the actual offer of 99, and the Swapper panics.

The overflow window for a given `rate` and `total` is approximately `1/rate - 1` values wide. For `rate = 0.1, total = 10`, any offer in `[90, 99]` triggers the bug. For `rate = 0.01, total = 1`, the window is ~99 values wide.

Trigger conditions

This is reachable whenever:

1. A limit-order pool has a small `total` at a fractional rate (common for low-liquidity ticks)
2. A swap offer slightly undershoots the pool's back-computed offer requirement
3. The +1 tolerance bridges the gap, routing to `distribute_full`

Affected code paths

Entry Point	SwapRequest Type	Affected?
`execute_swap`	`Yolo` / `Min` / `Exact`	**Yes** — no in-loop price guard, unconditional panic
`execute_swap`	`Limit`	**Sometimes** — the limit price check may break before the underflow if the achieved price exceeds the limit
`execute_new_order`	`Limit` (auto-set to order price)	**Conditional** — the limit check may protect
`QueryMsg::Simulate`	`Yolo`	**Yes** — query fails (no state change)
`ExecuteMsg::Arb`	N/A (uses `SwapItem::swap` directly)	**No** — `checked_sub` handles gracefully

Impact:

- **Per-swap DoS**: Individual `execute_swap` transactions (`Yolo`/`Min`/`Exact`) that exhaust better liquidity and reach the affected pool will panic and revert. No funds are lost (transaction reverts cleanly).
- **Griefing vector**: An attacker can create a limit order at a carefully selected fractional price with a small amount. Any user swap large enough to reach that tick in the book will revert. The attacker's pool sits deep in the book (low rate), so only swaps exhausting all better liquidity are affected.

- **No market-wide DoS**: The automatic arbitrage mechanism is unaffected because it uses `checked_sub` for profit validation. All `Swap`, `Order`, and `Range` operations that don't hit the poisoned pool through `Swapper` continue to function normally.
- **Query impact**: `Simulate` queries for large offers covering the affected tick will also fail.

Remediation:

Cap the `consumed_offer` returned by `distribute_full` to the original `offer` value. The simplest fix is to thread `offer` into `distribute_full`:

```
fn distribute_full(
    &mut self,
    offer: Uint256,
    rate: &Decimal256,
) -> Result<DistributionResult, BidPoolError> {
    let consumed_offer = self
        .total
        .multiply_ratio(rate.denominator(), rate.numerator())
        .min(offer);  // <-- cap at original offer
    // ...
}
```

Discussion:

> *Developer:*
>
> Hey there! We've taken care of the issue you pointed out. You can check out the fix in our latest commit at https://gitlab.com/thorchain/rujira/-/commit/970b137861042cbb9477610d7918ea1a7e013eb1. Let us know if there's anything else we should address!
>
> Fix URL: https://gitlab.com/thorchain/rujira/-/commit/970b137861042cbb9477610d7918ea1a7e013eb1

> *Auditor:*
>
> Hi team, thanks for jumping on that fix so quickly. We'll review the changes at the commit link you provided to ensure everything's squared away. If we spot anything else, we'll reach out. Appreciate the swift response!

Finding 5: LP sandwich on `RangeMsg::Create` due to lack of slippage control

Severity: Medium

Status: Resolved

Description:

`RangeMsg::Create` balances deposits using `Range::mid_price()` derived from existing ranges (not the order-book). Attackers can front-run with dust ranges to skew the mid-price and force a victim's deposit split, then trade against the newly created range.

Impact:

Victims can receive an adverse base/quote composition and suffer MEV losses when attackers trade against the skewed range.

Remediation:

Add min-amount or price-bound parameters to `RangeMsg::Create`.

Discussion:

> *Developer:*
>
> We've taken a good look at the FailSafe Admin aspect. While there's not a huge risk of malicious front-running, we figured adding a safety check for the UI is a smart move. You can check out the changes we made here: https://gitlab.com/thorchain/rujira/-/commit/0475cf1df0bab03a8e9e9ff2f077261611dd2a1f.
>
> Fix URL: https://gitlab.com/thorchain/rujira/-/commit/0475cf1df0bab03a8e9e9ff2f077261611dd2a1f

> *Auditor:*
>
> It's great to see you've addressed the potential issue with front-running by implementing a safety check in the UI. This adds an extra layer of protection, which is always a good thing. Thanks for taking that step and sharing the update with us.

Finding 6: MarketMaker Netting Underflow Causes DoS

Severity: Medium

Status: Resolved

Description:

`MarketMakerContext::compile` can underflow when netting opposite-direction entries for the same market maker, causing a panic and reverting the entire `Arb` execution.

- Netting logic in contracts/rujira-fin/src/market_maker/context.rs subtracts `ask_coin.amount` and `offer_coin.amount` when directions oppose.
- If opposite-direction entries have mismatched ratios (common when prices differ), `a.amount -= ask_coin.amount` (or the inverse) can underflow `Uint128`, panicking.
- This can occur when `Arb` consumes both sides from the same market maker in a single run, or when a malicious market maker returns skewed quotes.

Impact:

Panics during Arb execution revert swaps, order management, and range operations (all are wrapped in Arb). This enables a denial of service under adversarial or volatile conditions.

Remediation:

Use checked arithmetic and net by value in a single currency before converting, or enforce invariants that guarantee no underflow when offsetting opposite-direction entries.

Discussion:

> *Developer:*
>
> Hey, we took a look at the situation and while we think it's pretty unlikely to happen since we've got full control over the market makers, we decided it's better to be safe than sorry. So, we went ahead and added some checks to prevent any unnecessary panic. You can check out the changes we made in this commit: https://gitlab.com/thorchain/rujira/-/commit/061c342f1cbc9a89320937254bd4d8c696f9357d.
>
> Fix URL: https://gitlab.com/thorchain/rujira/-/commit/061c342f1cbc9a89320937254bd4d8c696f9357d

> *Auditor:*
>
> Great to hear you've taken steps to add those checks despite the low likelihood of an issue. We'll review the recent updates in the commit you provided to ensure everything's locked down nicely. Thanks for being proactive and keeping things secure.

Finding 7: Tick Change Strands Fixed-Price Orders, Locking User Funds

Severity: Medium

Status: Resolved

Description:

When governance changes the tick size via `SudoMsg::UpdateConfig` or `migrate`, existing Fixed-price orders at prices valid for the old tick but invalid for the new tick become permanently inaccessible. The order owner cannot withdraw filled amounts, retract remaining bids, or resize the order — yet the order continues to be filled by incoming swaps and arbs, accumulating unretrievable proceeds.

Root cause: `OrderManager::execute_orders()` validates every Fixed price against the current tick before any operation, including withdrawal/retract of existing orders.

In contracts/rujira-fin/src/order_pool/order_manager.rs:

```
for (side, price, target) in o {
    if let Price::Fixed(x) = price {
        self.config.tick.validate_price(&x)?;
    }
    let mut pool = Pool::load(storage, &price, &side, oracle);
    match pool.load_order(storage, &self.owner) {
        Ok(mut order) => {
            self.execute_existing_order(storage, &mut pool, &mut order, &side, target)?
        }
        // ...
    }
}
```

The tick gate fires before `Pool::load` and `load_order`, preventing all downstream operations. There is no alternative code path to interact with orders outside `execute_orders`.

Tick change entry points — governance can change the tick via:

1. `SudoMsg::UpdateConfig` in contracts/rujira-fin/src/contract.rs:

```
config.update(deps.api, &update)?;  // may change tick
config.validate(deps.as_ref())?;
config.save(deps.storage)?;
```

2. `migrate` in contracts/rujira-fin/src/contract.rs:

```
config.update(deps.api, &msg)?;  // may change tick
```

Neither migrates existing orders to the new tick.

The stranded orders keep filling. `Pool::iter()` in contracts/rujira-fin/src/order_pool/pool.rs iterates POOLS by storage prefix without any tick validation:

```
let fixed = POOLS
    .prefix((side.clone(), PoolType::Fixed))
    .range(storage, None, None, order)
    .filter_map(populate);
```

This means `SwapIter::iter()`, used by swaps and arbs, includes stranded orders. Incoming trades fill these orders, converting bid tokens into fill proceeds. The `BidPool::distribute` mechanism credits `filled` amounts to the order's `Bid`, but the owner can never call `maybe_withdraw` → `claim_order` to retrieve them.

Queries still work. `query_order` and `query_orders` in contracts/rujira-fin/src/order_pool/query.rs load orders directly via `Pool::load` → `load_order` without tick validation. The user can see their stranded order and its growing filled balance, but cannot interact with it.

Example scenario:

1. Contract deployed with `Tick(4)`. User places a Side::Quote order at `Price::Fixed(1.234)` — valid for 4 significant figures.
2. Governance changes tick to `Tick(2)` via `SudoMsg::UpdateConfig`.
3. `Tick(2).validate_price(&1.234)` fails — `1.234` truncated to 2 sig figs is `1.2 != 1.234`.
4. User attempts `Order([(Side::Quote, Price::Fixed(1.234), None)])` to withdraw fills → reverts at tick validation.
5. User attempts `Order([(Side::Quote, Price::Fixed(1.234), Some(0))])` to fully retract → reverts at tick validation.
6. Meanwhile, swaps continue filling the order at price 1.234, converting user's quote tokens to base proceeds that accumulate unretrievably.

Impact:

Permanent fund lockup for all users with Fixed-price orders at prices incompatible with the new tick. The severity scales with:

- The magnitude of the tick change (e.g., `Tick(6)` → `Tick(2)` strands many more prices than `Tick(4)` → `Tick(3)`)
- The total value locked in affected price levels
- The continued filling of stranded orders wastes the user's bid tokens while accumulating unretrievable proceeds

Oracle-priced orders (`Price::Oracle`) are unaffected since they bypass the tick validation entirely.

Remediation:

Skip tick validation when the user's intent is withdrawal/retract only. Add a dedicated code path or guard:

```
for (side, price, target) in o {
    let is_retract_or_withdraw = target.map_or(true, |t| t.is_zero());
    if let Price::Fixed(x) = price {
        if !is_retract_or_withdraw {
            self.config.tick.validate_price(&x)?;
        }
    }
    // ...
}
```

Discussion:

> *Developer:*
>
> Hey there! We've taken care of the issue with the FailSafe Admin. You can check out the fix at this link: https://gitlab.com/thorchain/rujira/-/commit/86f779bc4643c9abbe1b0774ebb9d1ddc76e18fd. Let us know if everything looks good on your end!
>
> Fix URL: https://gitlab.com/thorchain/rujira/-/commit/86f779bc4643c9abbe1b0774ebb9d1ddc76e18fd

> *Auditor:*
>
> Great to hear the FailSafe Admin issue has been addressed. We'll take a look at the latest commit link you provided to ensure everything is in order. Thanks for the quick response!

Finding 8: Unbounded Orderbook Iteration Can Cause DoS

Severity: Medium

Status: Acknowledged

Description:

`SwapIter` merges pools, ranges, and market makers into a single iterator that is consumed during swap and arb execution. There are no minimum deposit sizes or limits on the number of price levels/ranges, so an attacker can create many dust positions and force long iterations that risk out-of-gas failures.

- The swap path iterates the merged sources in contracts/rujira-fin/src/swap_iter.rs.
- `Swapper::swap` consumes this iterator in a `for` loop until the offer is fully consumed or a limit breaks in packages/rujira-rs/src/exchange/swapper.rs.
- Range creation only enforces non-empty funds, with no minimum size in contracts/rujira-fin/src/ranges/execute.rs.
- Order creation uses the provided target amount with no minimum size checks in contracts/rujira-fin/src/order_pool/order_manager.rs.
- This allows an attacker to create a large number of dust pools/ranges across many price levels, inflating the iterator length and swap execution cost.

Impact:

Swaps and arbs may exceed gas limits when the orderbook is flooded with many small pools/ranges, resulting in a denial of service for normal users. The impact depends on the attacker's willingness to lock funds but can still materially degrade market availability.

Remediation:

- Enforce minimum order and range sizes.
- Limit the number of active price levels or ranges per user and/or globally.
- Prune dust positions and consider expirations for inactive entries.
- Consider gas-bounded execution strategies that limit iteration per call.

Discussion:

> *Developer:*
>
> Hey, we've actually known about this vector since way back in mid-2022 when we first rolled out the fin. It's something we accepted as part of the process, and honestly, it hasn't caused any problems since then.

> *Auditor:*
>
> So, you've been aware of this vector since mid-2022 and decided it wasn't a problem for your setup. We just want to make sure it's still not posing any risk, given your current environment and any updates since then. Let's keep an eye on it just in case.

Finding 9: Recursive RangeOfferIter Can DoS with Many Empty Ranges

Severity: Low

Status: Resolved

Description:

`RangeOfferIter::next()` uses recursion when an offer slice has zero liquidity. If many ranges have zero liquidity, the recursion depth can grow large, risking stack overflow or excessive gas usage.

- `Range::withdraw` can reduce base/quote to zero but does not remove the range from storage.
- `RangeOfferIter::next()` calls itself recursively when `offer.total.is_zero()`.
- With a large number of empty ranges, the iterator can recurse many times before finding non-empty liquidity, leading to deep call stacks and wasted gas.
- Each empty range still required a non-zero initial deposit (potentially dust) before withdrawing 100% to reach zero liquidity.

Impact:

Excessive recursion can waste gas and, in extreme cases with many empty ranges, cause swap execution to fail. It does not inherently freeze the market but can degrade reliability under adversarial spam.

Remediation:

Replace recursion with an explicit loop and consider pruning ranges with zero liquidity from storage to keep the iterator bounded.

Discussion:

> *Developer:*
>
> Hey there! So, we realized that we can't just prune ranges from storage since they might only be empty in one direction. Instead, what we're doing is moving the secondary index "out of the way." This way, those empty ranges won't mess with the iterator. You can check out the fix we made over here: https://gitlab.com/thorchain/rujira/-/blob/fin/v1.2/contracts/rujira-fin/src/ranges/range.rs?reftype=heads#L98.
>
> Fix URL: https://gitlab.com/thorchain/rujira/-/blob/fin/v1.2/contracts/rujira-fin/src/ranges/range.rs?reftype=heads#L98

> *Auditor:*
>
> We've gone over the feedback and appreciate the insight. It makes sense that pruning ranges could cause issues if they're only empty one way. Your approach to reposition the secondary index seems like a solid workaround to prevent any iterator issues. Thanks for the update and providing the fix URL. We'll take a closer look at the changes you've made.

Finding 10: Range Iterator Infinite Loop When range_delta == 0

Severity: Info

Status: Resolved

Description:

If `range_delta` is zero, `RangeOfferIter` fails to advance price steps and can recurse indefinitely during swap iteration, leading to a gas-exhaustion DoS.

- `range_delta` is configurable and not validated in `Config::validate`.
- `RangeOfferIter::candidates` computes `step = start * delta`.
- With `delta == 0`, `step == 0`, so the next price candidate can remain equal to `start`.
- `RangeOfferIter::next()` builds a `RangeOffer` with `start == end`, yielding `total == 0`, and recursively calls `next()` again without advancing state.

Impact:

Any swap or arb that touches range liquidity can loop until out-of-gas, causing a denial of service.

Remediation:

Enforce range_delta > 0 (and ideally a minimum tick-safe value) in config validation and updates. Consider also adding a guard in the iterator to prevent zero-step progress.

Discussion:

> *Developer:*
>
> Hey, we've taken care of the issue by adding FailSafe Admin to our config validation. You can check out the changes we made here: https://gitlab.com/thorchain/rujira/-/commit/6f2a162687a180d3af903b499ba6790bc7322d04. Let us know if this addresses everything on your end.
>
> Fix URL: https://gitlab.com/thorchain/rujira/-/commit/6f2a162687a180d3af903b499ba6790bc7322d04

> *Auditor:*
>
> Great to hear you've added FailSafe Admin to the config validation. We'll review the changes at the provided

> link and make sure everything looks good from our side. Thanks for tackling this!

Disclaimer

This security report (“Report”) is provided by FailSafe (“Tester”) for the exclusive use of the client (“Client”). The scope of this assessment is limited to the security testing services performed against the systems, applications, or environments supplied by the Client. This Report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer, and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you (“Customer” or the “Company”) in connection with the Agreement. This Report, provided in connection with the Services set forth in the Agreement, shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This Report may not be transmitted, disclosed, referred to, or relied upon by any person for any purpose, nor may copies be delivered to any other person other than the Company, without FailSafe’s prior written consent in each instance.

This Report is not, nor should it be considered, an “endorsement” or “disapproval” of any particular project, system, or team. This Report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts FailSafe to perform security testing. This Report does not provide any warranty or guarantee regarding the absolute security or bug-free nature of the technology analyzed, nor does it provide any indication of the technology’s proprietors, business, business model, or legal compliance.

This Report should not be used in any way to make decisions around investment or involvement with any particular project. This Report in no way provides investment advice, nor should it be leveraged as investment advice of any sort. This Report represents an extensive testing process intended to help our customers identify potential security weaknesses while reducing the risks associated with complex systems and emerging technologies.

Technology systems, applications, and cryptographic assets present a high level of ongoing risk. FailSafe’s position is that each company and individual are responsible for their own due diligence and continuous security practices. FailSafe’s goal is to help reduce attack vectors and the high level of variance associated with utilizing new and evolving technologies, and in no way claims any guarantee of security or functionality of the systems we agree to test.

The security testing services provided by FailSafe are subject to dependencies and are under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. The testing process may include false positives, false negatives, and other unpredictable results. The services may access and depend upon multiple layers of third-party technologies.

ALL SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED “AS IS” AND “AS AVAILABLE” AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, FAILSAFE HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RE-

SPECT TO THE SERVICES, TESTING REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, FAILSAFE SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, FAILSAFE MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE TESTING REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE.

WITHOUT LIMITATION TO THE FOREGOING, FAILSAFE PROVIDES NO WARRANTY OR DISCLAIMER UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER FAILSAFE NOR ANY OF FAILSAFE'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. FAILSAFE WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, TESTING REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, TESTING REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO ANY OTHER PERSON WITHOUT FAILSAFE'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH SERVICES, TESTING REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF FAILSAFE CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST FAILSAFE WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED TESTING REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.