Rujira Staking - THORChain / Rujira

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

Date of Engagement: March 31st, 2025 - April 2nd, 2025

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
5	0	0	0	2	3

1. INTRODUCTION

`THORChain` engaged **Halborn** to conduct a security assessment on their smart contracts beginning on March 31st, 2025 and ending on April 2th, 2025. The security assessment was scoped to the smart contracts provided to the Halborn team. Commit hashes and further details can be found in the Scope section of this report.

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 mostly acknowledged by the `THORChain team`. The main ones were the following:

- `Ensure the new converter address is validated, the swap limit is within a safe range, and the binary message is properly formed. Also, emit an event with the updated values to improve transparency and traceability.`
- `Enforce stricter checks on denom format and validate token metadata fields like name, symbol, and decimals to ensure they follow expected structure and length constraints.`

3. TEST APPROACH AND METHODOLOGY

Halborn performed a combination of manual review of the code and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the smart contract assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of smart contracts and can quickly identify items that do not follow security best practices.

The following phases and associated tools were used throughout the term of the assessment:

- Research into the architecture, purpose, and use of the platform.
- Smart contract manual code review and walkthrough to identify any logic issue.
- Thorough assessment of safety and usage of critical Rust variables and functions in scope that could lead to arithmetic vulnerabilities.
- Scanning of Rust files for common vulnerabilities (`cargo audit`).

4. Caveats

This assessment focused exclusively on the files and logic explicitly defined within the provided scope. While external contract calls and interactions with shared packages were reviewed where relevant, a deeper analysis of their internal behavior was considered out of scope and may warrant a dedicated assessment if those components are critical to the system's security.

5. RISK METHODOLOGY

TABLE OF CONTENTS

1. Introduction
2. Assessment summary
3. Test approach and methodology
4. Caveats
5. Risk methodology
6. Scope
7. Assessment summary & findings overview
8. Findings & Tech Details
    8.1 Lack of parameter restrictions when updating revenue converter via privileged entry point
    8.2 Insufficient validation of instantiation inputs may allow misconfiguration
    8.3 Suppressed validation allows multi-denomination inputs to proceed silently
    8.4 Inconsistent calculation and ambiguous labeling of revenue-related fields
    8.5 Unbonding may succeed without returning any tokens to the user

// DOWNLOAD THE FULL REPORT
RUJIRA STAKING
SAVE AS PDF
* Use Google Chrome for best results
** Check "Background Graphics" in the print settings if needed

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.

5.1 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:

$$E = \prod m_e$$

5.2 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:

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

5.3 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

6. SCOPE

REPOSITORY

(a) Repository: rujira

(b) Assessed Commit ID: 471d744

(c) Items in scope:

- contracts/rujira-staking/src/config.rs
- contracts/rujira-staking/src/contract.rs
- contracts/rujira-staking/src/error.rs

↓ Expand ↓

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

REMEDIATION COMMIT ID:

- bde18fd

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

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	0	2	3

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
LACK OF PARAMETER RESTRICTIONS WHEN UPDATING REVENUE CONVERTER VIA PRIVILEGED ENTRY POINT	LOW	RISK ACCEPTED - 04/04/2025
INSUFFICIENT VALIDATION OF INSTANTIATION INPUTS MAY ALLOW MISCONFIGURATION	LOW	RISK ACCEPTED - 04/04/2025
SUPPRESSED VALIDATION ALLOWS MULTI-DENOMINATION INPUTS TO PROCEED SILENTLY	INFORMATIONAL	ACKNOWLEDGED - 04/04/2025
INCONSISTENT CALCULATION AND AMBIGUOUS LABELING OF REVENUE-RELATED FIELDS	INFORMATIONAL	SOLVED - 04/28/2025
UNBONDING MAY SUCCEED WITHOUT RETURNING ANY TOKENS TO THE USER	INFORMATIONAL	ACKNOWLEDGED - 04/04/2025

8. FINDINGS & TECH DETAILS

8.1 LACK OF PARAMETER RESTRICTIONS WHEN UPDATING REVENUE CONVERTER VIA PRIVILEGED ENTRY POINT

// LOW

Description

The `sudo` function in **contracts/rujira-staking/src/contract.rs** allows updating the `revenue_converter` field of the contract configuration without applying any constraints or validation to the provided values. While access to `sudo` is expected to be limited by the chain environment, the function permits setting an arbitrary contract address, execution message, and token transfer limit without enforcing bounds, verifying the message content, or restricting contract destinations. This could lead to unintended behavior or misconfiguration if the chain governance module is misused or incorrectly configured.

Code Location

This code updates the revenue converter configuration using unchecked input from the `SudoMsg` without applying validation on value bounds, message structure, or target contract, and does not emit any event to record the change:

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn sudo(deps: DepsMut, _env: Env, msg: SudoMsg) -> Result<Response, ContractError> {
    let mut config = Config::load(deps.storage)?;

    match msg {
        SudoMsg::SetRevenueConverter {
            contract,
            msg,
            limit,
        } => {
            config.revenue_converter = (deps.api.addr_validate(&contract)?, msg, limit);
            config.save(deps.storage)?;
            Ok(Response::default())
        }
    }
}
```

BVSS

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

Recommendation

It is recommended to add strict validation to the values accepted by the `SetRevenueConverter` variant of the `SudoMsg`, including reasonable bounds for the limit, validation of the binary message content, and allowlisting of expected contract addresses. Additionally, the function should emit events that clearly record the new configuration values for traceability and auditability. This ensures configuration integrity even in the event of governance misuse or misconfiguration.

Remediation Comment

RISK ACCEPTED: The **THORChain team** accepted this risk and stated the following:

> *Revenue converter validation is inherently difficult. The contract address and payload are deliberately arbitrary to support any type of converter. Even the amount is hard to bounds check, since a token with a very low price (e.g., 0.00001) will have vastly different thresholds compared to something like BTC. To properly validate the contract and message, you would need access to some of the designated funds and actually execute the contract with them.*

8.2 INSUFFICIENT VALIDATION OF INSTANTIATION INPUTS MAY ALLOW MISCONFIGURATION

// LOW

Description

The `instantiate` function in **contracts/rujira-staking/src/contract.rs** does not adequately validate all user-provided fields in the `InstantiateMsg`, which may lead to configuration errors during deployment. While critical fields like addresses and fee percentages are validated, others such as `bond_denom`, `revenue_denom`, and `receipt_token_metadata` are only partially checked. In particular, `bond_denom` and `revenue_denom` are only validated for non-emptiness, allowing incorrect formats such as uppercase letters, extra spaces, or special characters. Additionally, the metadata for the receipt token is passed to `TokenFactory::create_msg(...)` without validation, which could result in user-facing display issues or malformed tokens. These shortcomings could cause operational problems despite the assumption that the deployer is non-malicious.

Code Location

The `instantiate` function invokes `config.validate()` and constructs the token creation message using unchecked user input; however, it lacks proper format validation for denoms and metadata fields, which may lead to misconfiguration or malformed token definitions:

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn instantiate(
    deps: DepsMut,
    env: Env,
    _info: MessageInfo,
    msg: InstantiateMsg,
) -> Result<Response, ContractError> {
    set_contract_version(deps.storage, CONTRACT_NAME, CONTRACT_VERSION)?;
    let config = Config::new(deps.api, msg.clone())?;
    config.validate()?;
    config.save(deps.storage)?;
    init(deps.storage)?;
    let share_denom = TokenFactory::new(&env, format!("staking-{}", config.bond_denom).as_str());
    Ok(Response::default().add_message(share_denom.create_msg(msg.receipt_token_metadata)))
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (2.1)

Recommendation

It is recommended to implement stricter validation in the `validate` method of the `Config` struct within the **rujira-staking** contract, enforcing a proper format for denoms using a regex such as `^[a-z0-9]{3,64}$` and validating the contents of `receipt_token_metadata` to ensure that the token name, symbol, and decimals meet expected formatting and length constraints.

Remediation Comment

RISK ACCEPTED: The **THORChain team** accepted this risk and stated the following:

> *Regarding the validation comments, the emitted Cosmos SDK message is validated, specifically using the logic in* `cosmos-sdk/x/bank/types/metadata.go`*. This includes validation of the denom string via* `sdk.ValidateDenom(denom)`*, which ensures that only permitted SDK tokens are accepted.*

8.3 SUPPRESSED VALIDATION ALLOWS MULTI-DENOMINATION INPUTS TO PROCEED SILENTLY

// INFORMATIONAL

Description

The `execute` function in **contracts/rujira-staking/src/contract.rs** attempts to read the amount of `bond_denom` sent by the user using the `may_pay` helper. However, the result is wrapped in `unwrap_or_default()`, which causes the contract to silently fallback to zero if an error occurs. This suppresses important validation — specifically, if the user sends multiple coins (a case correctly rejected by `may_pay`), the fallback allows the transaction to continue and consume gas before ultimately failing in deeper message branches.

Code Location

The following line bypasses validation by ignoring the result of `may_pay`, defaulting to zero even when multiple coins are sent:

Copy Code

```
#[cfg_attr(not(feature = "library"), entry_point)]
pub fn execute(
    deps: DepsMut,
    env: Env,
    info: MessageInfo,
    msg: ExecuteMsg,
) -> Result<Response, ContractError> {
    let config = Config::load(deps.storage)?;
    let bond_amount_sent = may_pay(&info, config.bond_denom.as_str()).unwrap_or_default();
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:L/Y:N (1.7)

Recommendation

It is recommended to remove `unwrap_or_default()` and propagate the result of `may_pay` to avoid masking fund validation failures. If this change is made, the `may_pay` function must also be adjusted to return an error only when multiple coins are sent, and to return zero when no funds are provided or when a single coin with a different denom is sent. This ensures early rejection of invalid transactions while maintaining compatibility with valid branches such as `LiquidMsg::Unbond`.

Remediation Comment

ACKNOWLEDGED: The **THORChain team** acknowledged this issue and stated the following:

> *The distribution must occur before any user action to ensure that new joiners are not allocated existing fees. Not all user actions involve sending funds. This balance check was simply a convenient way to retrieve the token amount supplied. It is only used to deduct from the current contract balance in order to compute the net effect of the new deposit. This logic could have been implemented using an* `iter().find(...)` *with a default value of 0.*

8.4 INCONSISTENT CALCULATION AND AMBIGUOUS LABELING OF REVENUE-RELATED FIELDS

// INFORMATIONAL

Description

The `status` function in **contracts/rujira-staking/src/state.rs** exposes two revenue-related fields: `account_revenue` and `pending_revenue`. However, both the calculation and the naming of these fields raise concerns:

1. The computation of `revenue_surplus` is inconsistent with internal accounting logic:

It is defined as:

```
let revenue_surplus = revenue_balance.checked_sub(account.pending)?;
```

This omits subtracting `PENDING_SWAP`, which is properly accounted for in the `distribute` function. As a result, `status()` may report more revenue than is actually available for future distribution, especially in periods of high activity or when swaps are queued.

2. The field names `account_revenue` and `pending_revenue` appear semantically inverted or misleading:

- `account_revenue` shows `account.pending`, which reflects revenue already distributed to users but not yet claimed — a more accurate label might be `pending_rewards` or `assigned_rewards`.
- `pending_revenue` shows `revenue_surplus`, which represents undistributed contract-held revenue — better described as `unallocated_rewards` or `undistributed_revenue`.

If these values are queried by external systems, such as dashboards, indexers, or accounting tools, these inconsistencies may lead to **integrity issues**, incorrect assumptions, or flawed financial reporting. For example, if the contract holds 1,000 units of `revenue_denom`, with 600 units assigned to accounts (`account.pending`) and 200 units marked as `PENDING_SWAP`, the true unallocated surplus is only 200. However, the current logic would report a `pending_revenue` of 400, which is twice the actual amount that could be distributed.

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (1.7)

Recommendation

It is recommended to:

- Adjust the computation of `revenue_surplus` in `status()` to also subtract `PENDING_SWAP`, mirroring the internal logic used in `distribute()`.
- Rename the `StatusResponse` fields to clearly reflect their meanings, such as `pending_rewards` for user-assigned but unclaimed revenue, and `undistributed_rewards` for revenue still held by the contract but not yet allocated.
- Optionally document these fields explicitly if they are intended to be consumed by off-chain systems.

These changes will ensure consistency with the contract's internal logic, reduce ambiguity for developers, and avoid integrity issues in external integrations.

Remediation Comment

SOLVED: The **THORChain team** solved the issue in the specified commit id.

Remediation Hash

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

8.5 UNBONDING MAY SUCCEED WITHOUT RETURNING ANY TOKENS TO THE USER

// INFORMATIONAL

Description

The `LiquidMsg::Unbond` branch inside the `execute_liquid` function of **contracts/rujira-staking/src/contract.rs** allows unbonding operations to proceed even when the returned token amount is zero.

In this case, the contract burns the user's share tokens but performs no actual transfer of the underlying bond tokens. This could happen in rare edge cases — for example, due to rounding errors or unexpected internal state. Although unlikely, it may lead to silent value loss for the user without any visible error.

Code Location

The `returned` value is not validated before constructing the response:

```rust
fn execute_liquid(
    storage: &mut dyn Storage,
    env: &Env,
    info: MessageInfo,
    config: &Config,
    msg: LiquidMsg,
) -> Result<Response, ContractError> {
    let share_denom = TokenFactory::new(env, format!("staking-{}", config.bond_denom).as_str());

    match msg {
        LiquidMsg::Bond {} => {
            let amount = must_pay(&info, config.bond_denom.as_str())?;
            let shares = execute_liquid_bond(storage, amount)?;
            Ok(Response::default()
                .add_event(event_liquid_bond(info.sender.clone(), amount, shares))
                .add_message(share_denom.mint_msg(shares, info.sender)))
        }
        LiquidMsg::Unbond {} => {
            let shares = must_pay(&info, share_denom.denom().as_str())?;
            let returned = execute_liquid_unbond(storage, shares)?;
            Ok(Response::default()
                .add_event(event_liquid_unbond(info.sender.clone(), shares, returned))
                .add_message(share_denom.burn_msg(shares))
                .add_message(BankMsg::Send {
                    to_address: info.sender.to_string(),
                    amount: coins(returned.u128(), config.bond_denom.clone()),
                }))
        }
    }
}
```

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:N/I:L/D:N/Y:N (0.8)

Recommendation

It is recommended to verify that the `returned` value is strictly greater than zero before proceeding. If the amount is zero, the transaction should be canceled early by returning an appropriate error. This prevents value loss and avoids executing transactions that have no meaningful effect.

Remediation Comment

ACKNOWLEDGED: The **THORChain team** acknowledged this issue and stated the following:

> *If a zero amount is sent, the SDK throws an "invalid coins" error. Therefore, it should be impossible to receive a zero return when performing an unbond operation.*

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.

Privacy Policy Terms of Use Vulnerability Disclosure Policy Cookie Preferences