RUJI Lending - THORChain / Rujira

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

Date of Engagement: October 20th, 2025 - October 24th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
8	0	1	1	1	5

1. Introduction

`THORChain` engaged Halborn to conduct a security assessment of the `Ghost Vault` smart contract, beginning on October 19th, 2025 and ending on October 23th, 2025. Commit hashes and further details can be found in the **Sources** section of this report.

`Ghost Vault` is a CosmWasm-based lending and borrowing protocol designed to operate on top of the `Rujira` ecosystem. It enables users to deposit native assets into a vault and earn interest, while allowing borrowers to draw liquidity against those deposits. The protocol maintains two internal accounting pools (a deposit pool and a debt pool) and determines interest rates dynamically based on real-time utilization ratios. Interest accrual is applied linearly over time, and all interactions (deposits, borrows, repayments, withdrawals) trigger synchronization of the accumulated interest between pools.

2. Assessment Summary

The team at Halborn assigned a full-time security engineer to evaluate the security of the smart contracts. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract auditing, and deep knowledge of CosmWasm and other Web3 ecosystems.

The purpose of this assessment is to:

- Ensure that core vault functions operate as intended and maintain consistent accounting between depositors and borrowers.
- Identify potential vulnerabilities or logic flaws in the interest calculation, liquidity management, and share issuance mechanisms.

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

- `Convert overpayment shares back to base tokens before refunding to ensure accurate user reimbursements.`
- `Enforce borrower limits in base tokens instead of shares to prevent exceeding the real debt cap.`
- `Apply interest in chunks and update last_updated incrementally to avoid persistent DoS after long idle periods.`
- `Reject or accumulate tiny deposits that would mint zero shares to prevent silent value donation.`
- `Add a hard APR cap to prevent extreme rates or unsafe configurations.`
- `Validate all new interest parameters on sudo SetInterest to reject invalid or unsafe curves.`
- `Add a guard in repay for an empty debt pool to prevent division-by-zero reverts.`
- `Introduce a sudo-controlled pause mechanism to allow emergency stopping of critical operations during incidents.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

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

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:

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 Overpayment refunded in shares instead of base tokens
    7.2 Borrow limit checked in share units allows exceeding base-token cap
    7.3 Silent zero-share issuance from small deposits
    7.4 Persistent dos due to large idle-time interest accrual
    7.5 Lack of hard cap and unsafe default interest parameters
    7.6 Missing validation on sudo setinterest allows invalid interest curves
    7.7 Division-by-zero on repay when the debt pool is empty
    7.8 No sudo-controlled emergency stop

// DOWNLOAD THE FULL REPORT

RUJI LENDING

SAVE AS PDF

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

$$C = rs$$

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: fbf80aa

(c) Items in scope:

- contracts/rujira-ghost-vault/src/events.rs
- contracts/rujira-ghost-vault/src/error.rs
- contracts/rujira-ghost-vault/src/config.rs

↓ Expand ↓

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

REMEDIATION COMMIT ID:

- 394a1c2
- 76428ba
- 0fb88ac
- 008f46c
- 1f6cce7

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	1	1	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
OVERPAYMENT REFUNDED IN SHARES INSTEAD OF BASE TOKENS	HIGH	SOLVED - 10/28/2025
BORROW LIMIT CHECKED IN SHARE UNITS ALLOWS EXCEEDING BASE-TOKEN CAP	MEDIUM	SOLVED - 11/06/2025
SILENT ZERO-SHARE ISSUANCE FROM SMALL DEPOSITS	LOW	SOLVED - 10/08/2025
PERSISTENT DOS DUE TO LARGE IDLE-TIME INTEREST ACCRUAL	INFORMATIONAL	SOLVED - 11/24/2025
LACK OF HARD CAP AND UNSAFE DEFAULT INTEREST PARAMETERS	INFORMATIONAL	ACKNOWLEDGED - 11/28/2025
MISSING VALIDATION ON SUDO SETINTEREST ALLOWS INVALID INTEREST CURVES	INFORMATIONAL	SOLVED - 11/24/2025
DIVISION-BY-ZERO ON REPAY WHEN THE DEBT POOL IS EMPTY	INFORMATIONAL	SOLVED - 11/25/2025
NO SUDO-CONTROLLED EMERGENCY STOP	INFORMATIONAL	ACKNOWLEDGED - 11/28/2025

7. FINDINGS & TECH DETAILS

7.1 OVERPAYMENT REFUNDED IN SHARES INSTEAD OF BASE TOKENS

// HIGH

Description

The `repay` flow computes the borrower's overpayment in **debt-pool shares**, but then refunds that numeric value **directly as base tokens** (1:1), without converting shares back to their base-token value.

Because a share's value may exceed 1 base token after interest accrual (`debt_pool.size / debt_pool.shares ≥ 1`), this causes systematic **under-refunds**: debtors receive fewer tokens than they actually overpaid. This is an economic correctness bug that reduces user funds on every overpayment event and produces persistent accounting asymmetry between shares and token units.

Code Location

Code of `execute_market` / `Repay` handler from `contracts/rujira-ghost-vault/src/contract.rs` file:

Copy Code

```rust
MarketMsg::Repay { delegate } => {
    let owner = delegate
        .map(|x| deps.api.addr_validate(&x))
        .transpose()?
        .unwrap_or_else(|| info.sender.clone());

    let mut borrower = Borrower::load(deps.storage, owner.clone())?;

    let amount = must_pay(&info, config.denom.as_str())?;
    let shares = state.repay(amount)?;
    let overpayment = borrower.repay(shares);
    borrower.save(deps.storage)?;

    let mut response = Response::default().add_event(event_repay(owner, amount, shares));
    if !overpayment.is_zero() {
        response = response.add_message(BankMsg::Send {
            to_address: borrower.addr.to_string(),
            amount: coins(overpayment.u128(), config.denom),
        });
    }
    response
}
```

Proof of Concept

Scenario:

Ghost Vault with 1000 base tokens deposited and two borrowers (B1 and B2). B1 borrows 500 and repays 100, then the block time advances by 90 days to accrue interest. To ensure the pool's total debt exceeds B1's remaining debt (and avoid pool-level overflow), B2 borrows 50. The test queries B1's current debt in base tokens and repays exactly that amount plus an additional 30 tokens to intentionally overpay. The expected behavior is that the full 30-token surplus is refunded.

Test Code:

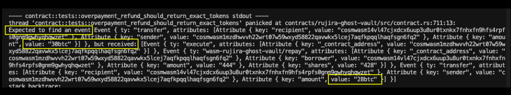
Copy Code

```rust
#[test]
#[should_panic]
fn overpayment_refund_should_return_exact_tokens() {
    let mut app = mock_rujira_app();
    let owner = app.api().addr_make("owner");
    let borrower = app.api().addr_make("borrower");
    let borrower2 = app.api().addr_make("borrower2");

    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &owner, coins(1_000_000, "btc"))
            .unwrap();
        router
            .bank
            .init_balance(storage, &borrower, coins(1_000_000, "btc"))
            .unwrap();
        router
            .bank
            .init_balance(storage, &borrower2, coins(1_000_000, "btc"))
            .unwrap();
    });

    let code = Box::new(ContractWrapper::new(execute, instantiate, query).with_sudo(sudo));
    let code_id = app.store_code(code);
    let contract = app
        .instantiate_contract(
            code_id,
            owner.clone(),
            &InstantiateMsg {
                denom: "btc".to_string(),
                receipt: TokenMetadata {
                    description: "".to_string(),
                    display: "".to_string(),
                    name: "".to_string(),
                    symbol: "".to_string(),
                    uri: None,
                    uri_hash: None,
                },
                interest: Interest {
                    target_utilization: Decimal::from_ratio(8u128, 10u128),
                    base_rate: Decimal::from_ratio(1u128, 10u128),
                    step1: Decimal::from_ratio(1u128, 10u128),
                    step2: Decimal::from_ratio(3u128, 1u128),
                },
            },
            &[],
            "template",
            None,
        )
        .unwrap();

    // Deposit liquidity
    app.execute_contract(
        owner.clone(),
        contract.clone(),
        &ExecuteMsg::Deposit { callback: None },
        &coins(1_000u128, "btc"),
    )
    .unwrap();

    // Whitelist borrower with limit
    app.wasm_sudo(
        contract.clone(),
        &SudoMsg::SetBorrower {
            contract: borrower.to_string(),
            limit: Uint128::from(1_000u128),
        },
    )
    .unwrap();

    // Borrow 500, repay 100 to reach same structure as lifecycle test
    app.execute_contract(
        borrower.clone(),
        contract.clone(),
        &ExecuteMsg::Market(MarketMsg::Borrow {
            callback: None,
            amount: Uint128::from(500u128),
            delegate: None,
        }),
        &[],
    )
    .unwrap();

    app.execute_contract(
        borrower.clone(),
        contract.clone(),
        &ExecuteMsg::Market(MarketMsg::Repay { delegate: None }),
        &[coin(100, "btc")],
    )
    .unwrap();

    // Accrue interest (90 days)
    app.update_block(|x| x.time = x.time.plus_days(90));
```

```
    // Add a second borrower with a small active debt so the pool total debt exceeds borrower1's repay
    app.wasm_sudo(
        contract.clone(),
        &SudoMsg::SetBorrower {
            contract: borrower2.to_string(),
            limit: Uint128::from(1_000u128),
        },
    )
    .unwrap();
    app.execute_contract(
        borrower2.clone(),
        contract.clone(),
        &ExecuteMsg::Market(MarketMsg::Borrow {
            callback: None,
            amount: Uint128::from(50u128),
            delegate: None,
        }),
        &[],
    )
    .unwrap();

    // Read current borrower debt in base tokens
    let b: BorrowerResponse = app
        .wrap()
        .query_wasm_smart(
            contract.clone(),
            &QueryMsg::Borrower {
                addr: borrower.to_string(),
            },
        )
        .unwrap();
    let remaining = b.current.u128();

    // Repay all remaining debt plus an extra 30 tokens to trigger overpayment refund
    let repay_amount = remaining + 30u128;

    let res = app
        .execute_contract(
            borrower.clone(),
            contract.clone(),
            &ExecuteMsg::Market(MarketMsg::Repay { delegate: None }),
            &[coin(repay_amount, "btc")],
        )
        .unwrap();

    // Correct behavior would refund full 30 tokens; current code refunds fewer due to share/token mismatch
    let expected_refund = format!("{}btc", 30u128);
    res.assert_event(
        &Event::new("transfer").add_attributes(vec![
            ("recipient", borrower.as_str()),
            ("sender", contract.as_str()),
            ("amount", expected_refund.as_str()),
        ]),
    );
}
```

Result:

- The repay transaction emits a bank transfer refund to the borrower with amount 28 instead of the expected 30.
- This demonstrates the overpayment is computed in share units but refunded as base tokens 1:1, under-refunding the payer when the debt share ratio > 1.



```
---- contract::tests::overpayment_refund_should_return_exact_tokens stdout ----
thread 'contract::tests::overpayment_refund_should_return_exact_tokens' panicked at contracts/rujira-ghost-vault/src/contract.rs:711:13:
Expected to find an event Event { ty: "transfer", attributes: [Attribute { key: "recipient", value: "cosmwasm14vl47cjxdcx6uup3u8ur0txnkx7fnhxfn9hfs4rpf
s0gnm9gwhyqhqwzet" }, Attribute { key: "sender", value: "cosmwasm1mzdhwvvh22wrt07w59wxyd58822qavwkx5lcej7aqfkpqqlhaqfsgn6fq2" }, Attribute { key: "amou
nt", value: "30btc" }] }, but received: [Event { ty: "execute", attributes: [Attribute { key: "_contract_address", value: "cosmwasm1mzdhwvvh22wrt07w59w
xyd58822qavwkx5lcej7aqfkpqqlhaqfsgn6fq2" }] }, Event { ty: "wasm-rujira-ghost-vault/repay", attributes: [Attribute { key: "_contract_address", value: "
cosmwasm1mzdhwvvh22wrt07w59wxyd58822qavwkx5lcej7aqfkpqqlhaqfsgn6fq2" }, Attribute { key: "borrower", value: "cosmwasm14vl47cjxdcx6uup3u8ur0txnkx7fnhxfn
9hfs4rpfs0gnm9gwhyqhqwzet" }, Attribute { key: "amount", value: "444" }, Attribute { key: "shares", value: "428" }] }, Event { ty: "transfer", attribut
es: [Attribute { key: "recipient", value: "cosmwasm14vl47cjxdcx6uup3u8ur0txnkx7fnhxfn9hfs4rpfs0gnm9gwhyqhqwzet" }, Attribute { key: "sender", value: "c
osmwasm1mzdhwvvh22wrt07w59wxyd58822qavwkx5lcej7aqfkpqqlhaqfsgn6fq2" }, Attribute { key: "amount", value: "28btc" }] }]
stack backtrace:
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:H/Y:M (8.8)

Recommendation

It is recommended to convert any `overpayment_shares` into **base tokens** before issuing the refund. Concretely, after `overpayment_shares` is computed, call the debt pool conversion helper (or compute `overpayment_shares * debt_pool.size / debt_pool.shares`) to obtain `refund_tokens` and refund that value.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by converting the overpayment value from shares to base tokens before the transfer.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/394a1c274ff6f281c5e6a1849bc88418a068a9f1

7.2 BORROW LIMIT CHECKED IN SHARE UNITS ALLOWS EXCEEDING BASE-TOKEN CAP

// MEDIUM

Description

The borrower limit (`Borrower.limit`) is conceptually defined and surfaced **in base tokens**, but the **execution path** validates it using **share units**. In `Borrower::borrow`, the check compares `borrower.shares + amount` against `borrower.limit` even though `amount` is already expressed **in shares** (converted from base tokens earlier).

When the **price-per-share > 1** (after interest accrual), each share corresponds to more than one base token, so a borrower can receive **more tokens than the nominal cap** while the share-based condition still passes.

This interpretation is backed by the **query path**, which explicitly treats the limit and current debt **in base tokens**:

```
// QueryMsg::Borrower { ... }
let current = state.debt_pool.ownership(borrower.shares); // shares -> base tokens
available = min(
    borrower.limit.checked_sub(current).unwrap_or_default(), // limit - current (both in tokens)
    state.deposit_pool.size() - state.debt_pool.size(),
);
```

This confirms the **intended unit** for `limit` is base tokens, making the **share-based execution check** a unit mismatch that can **permit over-borrowing** relative to the token-denominated cap.

Code Location

Code of `borrow` function from `contracts/rujira-ghost-vault/src/borrowers.rs` file:

```
pub fn borrow(&mut self, amount: Uint128) -> Result<(), ContractError> {
    if self.shares.add(amount).gt(&self.limit) {
        return Err(ContractError::BorrowLimitReached { limit: self.limit });
    }
    self.shares += amount;

    Ok(())
}
```

BVSS

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

Recommendation

It is recommended to enforce borrow limits in **consistent units**. Compute the borrower's current debt in base tokens and validate that the total will not exceed the limit.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue through the implementation of the recommended fix.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/76428ba5b9c6898d02afc6b8217010480391cc84

7.3 SILENT ZERO-SHARE ISSUANCE FROM SMALL DEPOSITS

// LOW

Description

When the pool already has issued shares, `SharePool::join` computes how many shares to mint with integer arithmetic like `minted_shares = floor(amount * total_shares / total_size)`.

For very small deposit `amount` this formula can round down to zero, so the contract **adds the deposited base tokens to the pool size but mints 0 shares** to the depositor. The depositor silently loses the deposited tokens and they become an immediate, implicit gain for existing share holders because the price-per-share increases.

This behaviour can be abused to skim value slowly from inattentive users or faulty integrations that perform micro-deposits, and it degrades UX (depositor believes they have a position but receives no receipt shares).

Code Location

Code of `join` function from `packages/rujira-rs/src/share_pool.rs` file:

```
pub fn join(&mut self, amount: Uint128) -> Result<Uint128, SharePoolError> {
    if amount.is_zero() {
        return Err(SharePoolError::ZeroAmount {});
    }
    if self.shares.is_zero() {
        self.size.add_assign(amount);
        self.shares.add_assign(amount);
        return Ok(amount);
    }

    let issuance = self.shares.multiply_ratio(amount, self.size);
    self.shares.add_assign(issuance);
    self.size.add_assign(amount);
    Ok(issuance)
}
```

BVSS

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

Recommendation

It is recommended to prevent silent zero-share deposits by one of the following approaches:
1. **Reject** deposits that would mint zero shares with a clear error (e.g., `MinDepositNotMet`) so the caller is informed and the funds are not silently accepted.
2. **Accumulate residuals** (the fractional part) per depositor and only mint when accumulated residuals reach at least one share.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by handling the error in case of minting 0 shares.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/0fb88acfed58d82f17b2de6ad0c13e3c7a9d50c1

7.4 PERSISTENT DOS DUE TO LARGE IDLE-TIME INTEREST ACCRUAL

// INFORMATIONAL

Description

Since the vault contract computes accrued interest using an **unbounded** time since last update (`dt = now - last_updated`), a very large `dt` (node downtime, long inactivity, or clock skew) can produce an extremely large `interest` amount:

`I = debt × rate(utilization) × (dt / YEAR)`

When the code then tries to credit that `I` to the deposit pool via `SharePool::deposit(I)`, the internal anti-dilution probe (simulating a `join` of `D = 1000` units and requiring round-trip loss ≤1%) can fail and **revert** (error `InvalidDeposit`).

Crucially, the revert happens **before** `last_updated` is advanced. Because `last_updated` remains unchanged, the pending `dt` (and thus `I`) is the same or larger on the next attempt. Repeating the call therefore repeatedly re-triggers the exact same revert, so the contract remains stuck in a state where no operation that first calls `distribute_interest()` can progress. This is a **persistent DoS**: the interest accrual that should be applied prevents any state change because the guard reverts and `last_updated` is never updated.

`NOTE`: Given the implausible preconditions required (multi-year inactivity, tiny fully utilized pool, and/or extreme sudo interest settings), the Δt/anti-dilution DoS is not reproducible under realistic operations and has been downgraded to **INFORMATIONAL**.

Code Location

Code of `calculate_interest` and `distribute_interest` functions from `contracts/rujira-ghost-vault/src/state.rs`:

```rust
pub fn calculate_interest(
    &mut self,
    interest: &Interest,
    to: Timestamp,
) -> Result<Uint128, ContractError> {
    let rate = self.debt_rate(interest)?;
    let seconds = to.seconds().sub(self.last_updated.seconds());
    let part = Decimal::from_ratio(seconds, 31_536_000u128);
    Ok(Decimal::from_ratio(self.debt_pool.size(), 1u128)
        .mul(rate)
        .mul(part)
        .to_uint_floor())
}

pub fn distribute_interest(&mut self, env: &Env, config: &Config) -> Result<(), ContractError> {
    // Calculate interest charged on total debt since last update
    let interest = self.calculate_interest(&config.interest, env.block.time)?;

    // Allocate it to the deposit pool
    self.deposit_pool.deposit(interest)?;
    // Charge it to the debt pool, so that outstanding debt tokens are required to
    // pay this interest on return
    self.debt_pool.deposit(interest)?;
    self.last_updated = env.block.time;

    Ok(())
}
```

Code of `deposit` function from `packages/rujira-rs/src/share_pool.rs`:

```rust
pub fn deposit(&mut self, amount: Uint128) -> Result<(), SharePoolError> {
    let mut checked = self.clone();
    checked.size.add_assign(amount);
    let deposit = Uint128::from(1000u128);
    let test = checked.join(deposit)?;
    let value = checked.ownership(test);
    let ratio = Decimal256::from_ratio(value, deposit);

    // If a deposit causes a new `join` of 1000 units to lose more than 1% of its value,
    // the share pool is in an invalid state, error and don't change state
    if ratio < Decimal256::from_ratio(99u128, 100u128) {
        return Err(SharePoolError::InvalidDeposit {});
    }
    self.size.add_assign(amount);
    Ok(())
}
```

Proof of Concept

Scenario

The PoC initializes a Ghost Vault with minimal liquidity and an active debt position, then advances the chain clock by a very long idle period. On the next interaction, the contract attempts to accrue a very large amount of interest and credit it via SharePool::deposit, triggering the anti-dilution probe and reverting. Because `last_updated` is not advanced on revert, repeating any operation that calls `distribute_interest` first (queries and executes) reverts again, demonstrating a persistent DoS under extreme, unrealistic conditions.

Test Code

```rust
#[test]
fn interest_accrual_unbounded_dt_causes_persistent_invalid_deposit() {
    let mut app = mock_rujira_app();
    let owner = app.api().addr_make("owner");
    let borrower = app.api().addr_make("borrower");

    app.init_modules(|router, _, storage| {
        router
            .bank
            .init_balance(storage, &owner, coins(1_000_000, "btc"))
            .unwrap();
        router
            .bank
            .init_balance(storage, &borrower, coins(1_000_000, "btc"))
            .unwrap();
    });

    let code = Box::new(ContractWrapper::new(execute, instantiate, query).with_sudo(sudo));
    let code_id = app.store_code(code);
    let contract = app
        .instantiate_contract(
            code_id,
            owner.clone(),
            &InstantiateMsg {
                denom: "btc".to_string(),
                receipt: TokenMetadata {
                    description: "".to_string(),
                    display: "".to_string(),
                    name: "".to_string(),
                    symbol: "".to_string(),
                    uri: None,
                    uri_hash: None,
                },
                // Initial interest values won't matter; we'll override via sudo
                interest: Interest {
                    target_utilization: Decimal::from_ratio(8u128, 10u128),
                    base_rate: Decimal::zero(),
                    step1: Decimal::zero(),
                    step2: Decimal::from_ratio(2u128, 1u128),
                },
            },
            &[],
            "ghost-vault",
            None,
        )
        .unwrap();

    // Minimal liquidity
    app.execute_contract(
        owner.clone(),
        contract.clone(),
        &ExecuteMsg::Deposit { callback: None },
        &coins(1u128, "btc"),
    )
    .unwrap();

    // Whitelist and create minimal debt (so debt_pool.shares > 0)
    app.wasm_sudo(
        contract.clone(),
        &SudoMsg::SetBorrower {
            contract: borrower.to_string(),
            limit: Uint128::from(1_000u128),
        },
    )
    .unwrap();
    app.execute_contract(
        borrower.clone(),
        contract.clone(),
        &ExecuteMsg::Market(MarketMsg::Borrow {
            callback: None,
            amount: Uint128::from(1u128),
            delegate: None,
        }),
        &[],
    )
    .unwrap();

    // Crank interest to extreme so that large dt produces huge accrual
    app.wasm_sudo(
        contract.clone(),
        &SudoMsg::SetInterest(Interest {
            target_utilization: Decimal::from_ratio(8u128, 10u128),
            base_rate: Decimal::zero(),
            step1: Decimal::zero(),
            step2: Decimal::from_ratio(1000u128, 1u128), // very steep above target
        }),
    )
    .unwrap();

    // Large idle period → very large dt
    app.update_block(|x| x.time = x.time.plus_days(2000));

    // Any path that triggers distribute_interest() should now revert with SharePool::InvalidDeposit
    // because the simulated join(1000) issuance becomes 0 due to enormous size vs shares.
    let err1: Result<StatusResponse, _> = app
        .wrap()
        .query_wasm_smart(contract.clone(), &QueryMsg::Status {});
    assert!(err1.is_err());

    // Persistent: last_updated was not advanced, so repeating yields the same error again
    let err2: Result<StatusResponse, _> = app
        .wrap()
        .query_wasm_smart(contract.clone(), &QueryMsg::Status {});
    assert!(err2.is_err());

    // Also demonstrate DoS on execute paths that call distribute_interest() first
    // 1) Deposit should fail
    let res = app.execute_contract(
        owner.clone(),
        contract.clone(),
        &ExecuteMsg::Deposit { callback: None },
        &coins(1u128, "btc"),
    );
    assert!(res.is_err());

    // 2) Repay should fail
    let res = app.execute_contract(
        borrower.clone(),
        contract.clone(),
        &ExecuteMsg::Market(MarketMsg::Repay { delegate: None }),
        &[coin(1u128, "btc")],
    );
    assert!(res.is_err());
}
```

Result

After advancing time by multiple years with a small, fully utilized pool, both Status queries and execute paths (e.g., Deposit, Repay) consistently revert due to the anti-dilution check during interest distribution. The state remains stuck because `last_updated` is never updated, confirming the DoS mechanism under these extreme parameters.

BVSS

AO:A/AC:L/AX:H/R:N/S:U/C:N/A:M/I:N/D:N/Y:N (1.6)

Recommendation

It is recommended to prevent this persistent DoS by **limiting the time step**. Use `step = dt.min(MAX_DT_SECONDS)` (for example, `MAX_DT = 1 day`).
Apply the interest for that step (`I_step`), then advance `last_updated` by `step`. Repeat in a loop until the full `dt` is consumed. This way, no single `I_step` becomes so large that the anti-dilution probe fails every time.

Remediation Comment

SOLVED: The **Rujira team** solved this issue by changing the `SharePool.shares` type from Uint128 to Decimal. Decimal shares prevent truncation and rounding loss; consequently, the anti-dilution probe no longer triggers on large interest credits, eliminating the denial-of-service condition.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/0fb88acfed58d82f17b2de6ad0c13e3c7a9d50c1

7.5 LACK OF HARD CAP AND UNSAFE DEFAULT INTEREST PARAMETERS

// INFORMATIONAL

Description

The core issue is that the interest calculation lacks a **hard cap** on the final APR. The piecewise utilization curve can generate very high rates from the `Interest` parameters (`target_utilization`, `base_rate`, `step1`, `step2`), but there is **no hard cap** to stop the final APR at a safe maximum.

As a secondary note, the `Interest::default()` parameters are economically aggressive (`target=0.8`, `base=0`, `step1=1`, `step2=2` → up to ~300% APR at U=1). However, these defaults are **not critical** because deployers are expected to override them during instantiation. The absence of a **hard cap** remains the principal risk, regardless of the chosen parameters.

Code Location

Code of `default` and `validate` functions from `contracts/rujira-ghost-vault/src/interest.rs` file:

```rust
impl Default for Interest {
    fn default() -> Self {
        Self {
            target_utilization: Decimal::from_ratio(4u128, 5u128),
            base_rate: Decimal::zero(),
            step1: Decimal::one(),
            step2: Decimal::from_ratio(2u128, 1u128),
        }
    }
}

impl Interest {
    pub fn validate(&self) -> StdResult<()> {
        ensure!(
            self.step2.gt(&self.step1),
            StdError::generic_err("step2 must be > step1".to_string())
        );

        ensure!(
            !self.target_utilization.is_zero(),
            StdError::generic_err("target_utilization must be > 0".to_string())
        );

        ensure!(
            self.target_utilization.lt(&Decimal::one()),
            StdError::generic_err("target_utilization must be < 1".to_string())
        );
        Ok(())
    }
```

BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:N/I:N/D:N/Y:C (1.3)

Recommendation

It is recommended to:

- Implement a **hard cap** on the computed APR (e.g., `min(calculated_rate, MAX_APR)` to limit interest growth even under faulty parameters).
- Add a **rate-of-change clamp** to restrict how much the APR can change between accrual updates (e.g., per-hour or per-block limits).
- Replace the current defaults with **economically safe parameters** — for example, a non-zero `base_rate` (2–5%), moderate step values (≤0.2–0.5), and a capped maximum APR (50–100%).
- Require explicit parameter initialization at instantiation (avoid using the `default()` constructor in production).

Remediation Comment

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

7.6 MISSING VALIDATION ON SUDO SETINTEREST ALLOWS INVALID INTEREST CURVES

// INFORMATIONAL

Description

The `sudo` handler that updates interest parameters does not call a validation step before saving. Because of this, governance can set **invalid curves** (for example: `target_utilization == 0` or `>= 1`, or `step2 <= step1`).

These wrong values can create **division-by-zero** in the piecewise formula (divide by `ut` or by `1 - ut`), or non-monotonic/flat behavior that breaks expected economics. The result is **failed executes/queries** (panic/revert) and a **configuration-based DoS** risk until parameters are corrected.

Code Location

Code of `sudo` handler (SetInterest branch) from `contracts/rujira-ghost-vault/src/contract.rs` file:

Copy Code

```rust
pub fn sudo(deps: DepsMut, _env: Env, msg: SudoMsg) -> Result<Response, ContractError> {
    let mut config = Config::load(deps.storage)?;

    match msg {
        SudoMsg::SetBorrower { contract, limit } => {
            Borrower::set(deps.storage, deps.api.addr_validate(&contract)?, limit)?;
            Ok(Response::default())
        }
        SudoMsg::SetInterest(interest) => {
            config.interest = interest;
            config.save(deps.storage)?;
            Ok(Response::default())
        }
    }
}
```

BVSS

AO:S/AC:L/AX:M/R:F/S:U/C:N/A:N/I:N/D:N/Y:C (0.3)

Recommendation

It is recommended to validate the new interest parameters before persisting them (e.g., `interest.validate()`) ensuring invalid curves are rejected.

Remediation Comment

SOLVED: The **Rujira team** has solved this issue by adding interest validation into `SudoMsg`.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/008f46c534b118b65721a9cc00425321d09f15ee

7.7 DIVISION-BY-ZERO ON REPAY WHEN THE DEBT POOL IS EMPTY

// INFORMATIONAL

Description

When there is no outstanding debt (`debt_pool.size() == 0`), the `repay` path converts the incoming base-token `amount` to **debt shares** using a division by the pool size. This leads to a **division-by-zero revert/panic** before any borrower-specific checks run.

The issue does not endanger funds but degrades usability and can disrupt integrations that expect graceful handling of zero-debt scenarios.

Code Location

Code of `repay` function from `contracts/rujira-ghost-vault/src/state.rs` file:

Copy Code

```rust
pub fn repay(&mut self, amount: Uint128) -> Result<Uint128, ContractError> {
    // Calculate the amount of shares that this repay will burn
    let shares = amount.multiply_ratio(self.debt_pool.shares(), self.debt_pool.size());
    self.debt_pool.leave(shares)?;
    Ok(shares)
}
```

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 guard the repay path against an empty debt pool and return a clear, non-panicking result.

Remediation Comment

SOLVED: The **Rujira team** solved this issue.

Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/1f6cce711b0245c13222fbc773000d650cc9536c#d27050d340235f9a9ffc9bf1e86536cfcd9b5d91

7.8 NO SUDO-CONTROLLED EMERGENCY STOP

// INFORMATIONAL

Description

The vault does not have a global **pause** (circuit-breaker) to stop critical actions like `deposit`, `withdraw`, `borrow`, and `repay`.

If there is an incident (wrong interest params, big accrual after long idle, accounting bug), operators cannot pause the contract on-chain to contain the impact and fix the problem. This increases operational risk (runaway debt, economic DoS, bad user experience).

Having an **emergency pause** is a **good practice** in DeFi: it lets the team react fast during incidents and protect users while a fix or parameter change is prepared.

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 add a `sudo` message to control a global pause flag and enforce it in all state-changing entry points.

Remediation Comment

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

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.