# Credit Accounts - THORChain / Rujira

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

Date of Engagement: October 29th, 2025 - November 3rd, 2025

## Summary

**100%** OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

| ALL FINDINGS | CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|---|
| 7 | 0 | 0 | 2 | 2 | 3 |

### 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 Conflicting liquidation checks allow denial of service
   7.2 Liquidation dos when final ltv drops below adjustment_threshold
   7.3 Missing validation for liquidation preferences
   7.4 Missing enforcement of critical configuration invariants
   7.5 Missing validation in setvault and setcollateral
   7.6 Division-by-zero in adjusted ltv
   7.7 Missing repay message allows incomplete settlement flow

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

### 1. Introduction

`THORChain` engaged Halborn to conduct a security assessment of the `Ghost Credit` contracts, beginning on October 29th, 2025 and ending on November 4th, 2025. This security assessment was scoped to the CosmWasm smart contracts in the Rujira Ghost repository. Commit hashes and further details can be found in the **Sources** section of this report.

`Ghost Credit` is a decentralized credit module designed for the `Rujira` ecosystem. It enables users to create self-custodied credit accounts that interact with mapped `Ghost Vault` markets to borrow and repay assets against on-chain collateral. The system enforces per-account Loan-to-Value (LTV) ratios, supports permissionless liquidations with bounded over-liquidation windows, and integrates liquidation preferences to determine the order of collateral consumption. The contracts are implemented in Rust using CosmWasm and operate over THOR chain.

### 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 and logic issues within the contract implementation

In summary, Halborn identified several improvements to reduce the likelihood and impact of operational and financial risks, which were partially addressed by the `Rujira team`. The main ones were the following:

- `Refactor liquidation validation to separate collateral and debt sets, ignore debt-denom increases, and compute slippage in USD value with oracle pricing.`
- `Treat “Account Safe” (LTV below adjustment threshold) as a successful liquidation outcome or remove the unnecessary unsafe check from the liquidation path.`
- `Validate liquidation preferences on save to reject cycles, self-references, and invalid or non-whitelisted denoms, and ensure acyclic dependency enforcement.`
- `Add strict configuration validation enforcing coherent thresholds, fee sums below 1, valid liquidation windows, and consistent parameter ranges on every update.`
- `Enforce cross-validation in SetVault and SetCollateral ensuring ratios are in [0,1], borrow denoms are whitelisted, and mappings remain consistent under governance updates.`
- `Guard LTV calculations against division-by-zero and rounding errors by returning safe sentinel values and performing conservative comparisons in post-checks.`

### 3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to ensure coverage and precision throughout this assessment. Manual review focused on business logic, inter-contract interactions, and edge-case validation, while automated tooling was used to detect known vulnerability classes and dependency risks. The following phases and techniques were used during the audit:

- Research into the architecture, purpose, and integration with Ghost Vault contracts.
- Manual line-by-line code review and walkthrough of all CosmWasm modules in scope.
- Manual assessment of critical functions to detect arithmetic and logical vulnerabilities.
- Evaluation of post-check atomicity and liquidation invariants.
- Verification of authorization boundaries and sudo/admin safety.
- Review of cross-contract calls to external vaults and potential DoS vectors.
- Automated static analysis using `cargo audit` and other Rust-specific scanning tools.
- Review and verification of integration and multi-test scenarios for borrow/repay/liquidation flows.

### 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)<br>Specific (AO:S) | 1<br>0.2 |
| Attack Cost (AC) | Low (AC:L)<br>Medium (AC:M)<br>High (AC:H) | 1<br>0.67<br>0.33 |
| Attack Complexity (AX) | Low (AX:L)<br>Medium (AX:M)<br>High (AX:H) | 1<br>0.67<br>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)<br>Low (C:L)<br>Medium (C:M)<br>High (C:H)<br>Critical (C:C) | 0<br>0.25<br>0.5<br>0.75<br>1 |
| Integrity (I) | None (I:N)<br>Low (I:L)<br>Medium (I:M)<br>High (I:H)<br>Critical (I:C) | 0<br>0.25<br>0.5<br>0.75<br>1 |
| Availability (A) | None (A:N)<br>Low (A:L)<br>Medium (A:M)<br>High (A:H)<br>Critical (A:C) | 0<br>0.25<br>0.5<br>0.75<br>1 |
| Deposit (D) | None (D:N)<br>Low (D:L)<br>Medium (D:M)<br>High (D:H)<br>Critical (D:C) | 0<br>0.25<br>0.5<br>0.75<br>1 |
| Yield (Y) | None (Y:N)<br>Low (Y:L)<br>Medium (Y:M)<br>High (Y:H)<br>Critical (Y:C) | 0<br>0.25<br>0.5<br>0.75<br>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)<br>Partial (R:P)<br>Full (R:F) | 1<br>0.5<br>0.25 |
| Scope ($s$) | Changed (S:C)<br>Unchanged (S:U) | 1.25<br>1 |

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

$$C = rs$$

The Vulnerability Severity Score $S$ is obtained by:

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

The score is rounded up to 1 decimal places.

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

## 5. SCOPE

REPOSITORY

(a) Repository: rujira

(b) Assessed Commit ID: 9989bb5

(c) Items in scope:

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

↓ Expand ↓

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

REMEDIATION COMMIT ID:

- 1f6cce7
- beaf062
- ed05d6f
- 07bf175

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

## 6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

| CRITICAL | HIGH | MEDIUM | LOW | INFORMATIONAL |
|---|---|---|---|---|
| 0 | 0 | 2 | 2 | 3 |

| SECURITY ANALYSIS | RISK LEVEL | REMEDIATION DATE |
|---|---|---|
| CONFLICTING LIQUIDATION CHECKS ALLOW DENIAL OF SERVICE | MEDIUM | SOLVED - 11/24/2025 |
| LIQUIDATION DOS WHEN FINAL LTV DROPS BELOW ADJUSTMENT_THRESHOLD | MEDIUM | NOT APPLICABLE - 11/28/2025 |
| MISSING VALIDATION FOR LIQUIDATION PREFERENCES | LOW | SOLVED - 11/24/2025 |
| MISSING ENFORCEMENT OF CRITICAL CONFIGURATION INVARIANTS | LOW | SOLVED - 11/24/2025 |
| MISSING VALIDATION IN SETVAULT AND SETCOLLATERAL | INFORMATIONAL | SOLVED - 11/24/2025 |
| DIVISION-BY-ZERO IN ADJUSTED LTV | INFORMATIONAL | ACKNOWLEDGED - 11/28/2025 |
| MISSING REPAY MESSAGE ALLOWS INCOMPLETE SETTLEMENT FLOW | INFORMATIONAL | ACKNOWLEDGED |

## 7. FINDINGS & TECH DETAILS

### 7.1 CONFLICTING LIQUIDATION CHECKS ALLOW DENIAL OF SERVICE

// MEDIUM

#### Description

A logic conflict between two liquidation validators (`ensure_collateral_decrease` and `validate_liquidation_execute_step`) can either **abort valid liquidation routes (DoS)** or **allow undetected movement of non-collateral tokens**, depending on whether the debt token is present in the collateral whitelist.

Both functions rely on overlapping sets of whitelisted denoms derived from `AccountCredit.collaterals` (built from `Config.collateral_ratios`), but they serve opposite purposes:

- `ensure_collateral_decrease` rejects any ***increase*** or ***new*** collateral denom in `AccountCredit.collaterals`.
- `validate_liquidation_execute_step` expects that only the debt token was received. It enforces ***"only-debt-received"*** and computes slippage based on received debt.

Because both use `AccountCredit.collaterals` as the observed set, two problematic cases arise:

1. **Debt denom also appears in `AccountCredit.collaterals`:**
   A liquidation step (Execute) that converts collateral into the debt token legitimately increases the balance of that denom in the account.
   The ***"Collateral increase after liquidation"*** revert only manifests when the execution queue has **no further steps** for that iteration ( i.e., when `Execute` is the **only** step or the **last** step before checks are re-evaluated). If the `Execute` step is immediately followed by a `Repay` within the same liquidation call, the engine tolerates the intermediate increase error while the queue is non-empty, performs the `Repay`, and only then applies final checks, which can result in a ***"Account Safe"*** outcome or a success depending on parameter calibration.
   - Result: **Denial-of-Service;** valid liquidation routes that should succeed are reverted.
2. **Debt denom not included in `AccountCredit.collaterals`:**
   The received debt is invisible to the validator (`received = ∅`), so ***only-debt-received*** passes vacuously while slip calculation uses zero received (causing ~100% slip or divide-by-zero style reverts). Also, movements of non-whitelisted tokens can go undetected, enabling **value movement without detection**.
   - Result: **Denial-of-Service;** valid liquidation routes that should succeed are reverted and other non-collateral tokens may move during liquidation without being noticed.

**Net effect:** The liquidation pipeline is fragile. With Execute-only (or Execute as terminal step), the route can be blocked by "Collateral increase." With Execute+Repay, the increase change may bypass intermediate checks and may later abort as "Account Safe" (over-correction), or succeed depending on calibration.

#### Code Location

Code of `ensure_collateral_decrease` and `validate_liquidation_execute_step` functions from `src/account.rs`:

Copy Code

```
pub fn validate_liquidation_execute_step(
    &self,
    deps: Deps,
    config: &Config,
    old: &Self,
) -> Result<(), ContractError> {
    let old_balance = old.balance();
    let new_balance = self.balance();
    let received = old_balance.received(&new_balance);
    let spent = old_balance.sent(&new_balance);

    for coin in received.clone().into_vec() {
        if !self.debts.iter().any(|x| x.item.can_accept(&coin)) {
            return Err(ContractError::InvalidReceipt { coin: coin.clone() });
        }
    }

    // Check against liquidation preference order
    for coin in spent.clone().into_vec() {
        self.liquidation_preferences
            .order
            .validate(&coin, &new_balance)?;
    }

    // Check against config liquidation slip
    let spent_value = spent.value_usd(deps.querier)?;
    let received_value = received.value_usd(deps.querier)?;
    if !spent_value.is_zero() {
        let slip = spent_value
            .checked_sub(received_value)
            .unwrap_or_default()
            .div(spent_value);

        ensure!(
            slip.le(&config.liquidation_max_slip),
            ContractError::LiquidationMaxSlipExceeded { slip }
        );
    }

    Ok(())
}

/// Ensure that all the collaterals that we now have were already present, and are now less than or equal to
pub fn ensure_collateral_decrease(&self, old: &Self) -> Result<(), ContractError> {
    let old_balance = old.balance();
    let new_balance = self.balance();
    // this will error if there are:
    //   a) new tokens not originally present
    //   b) new balance is greater than old balance
    old_balance
        .clone()
        .sub(new_balance.clone().into_vec())
        .map_err(|_| ContractError::LiquidationCollateralIncrease {
            old: old_balance,
            new: new_balance,
        })?;

    Ok(())
}
```

#### Proof of Concept

#### Scenario

A liquidation route with only an Execute step (no Repay) swaps collateral into the debt token while that debt denom is whitelisted as collateral. This increases the account's collateral-only balance versus the original snapshot. On the next DoLiquidate iteration, the contract compares "original vs current" collateral balances and aborts with "Collateral increase after liqudiation," preventing the liquidator from proceeding (operational DoS).

#### Test Code

Copy Code

```
#[test]
fn liquidation_execute_triggers_collateral_increase_error() {
    let mut app = mock_rujira_app();
    let owner = app.api().addr_make("owner");
    let ctx = setup(&mut app, &owner);

    // Fund account with 1 BTC as collateral
    app.send_tokens(
        owner.clone(),
        ctx.account.account.contract(),
        &[coin(100_000_000, BTC)],
    )
    .unwrap();

    // Borrow 100,000 USDC and send it out so it doesn't inflate collateral
    let account = ctx.ghost_credit.query_account(&app, &ctx.account.idx);
    ctx.ghost_credit
        .account_borrow(&mut app, &account, 100_000_000_000u128, USDC)
        .unwrap();
    ctx.ghost_credit
        .account_send(
            &mut app,
            &account,
            100_000_000_000u128,
            USDC,
            &owner,
        )
        .unwrap();

    // Crash collateral price so LTV > 1
    app.init_modules(|router, _api, _storage| {
        router
            .stargate
            .with_prices(vec![
                ("BTC", Decimal::from_str("1000").unwrap()),
                ("USDC", Decimal::from_str("1.0").unwrap()),
            ]);
    });
```

```rust
    let account = ctx.ghost_credit.query_account(&app, &ctx.account.idx);
    assert!(account.ltv > Decimal::one());

    // Liquidate with ONLY an Execute swap BTC->USDC.
    // USDC is whitelisted as collateral in setup, so this increases
    // collateral versus the original snapshot, and the next DoLiquidate
    // iteration should fail on ensure_collateral_decrease.
    let err = ctx
        .ghost_credit
        .liquidate_execute(
            &mut app,
            &account,
            ctx.fin_btc_usdc.addr(),
            fin::ExecuteMsg::Swap(fin::SwapRequest {
                min_return: None,
                to: None,
                callback: None,
            }),
            coins(500_000, BTC),
        )
        .unwrap_err();

    let msg = format!("{err:?}");
    assert!(
        msg.contains("Collateral increase after liqudiation"),
        "unexpected error: {msg}"
    );
}
```

### Result

Running the PoC test causes Liquidate to fail with the specific error “Collateral increase after liqudiation” immediately after the Execute step, confirming that a swap into a debt-as-collateral denom is treated as collateral growth and halts liquidation before any Repay can occur. This demonstrates an actionable DoS path controlled by route selection and configuration (debt denom present in collateral whitelist).

### BVSS

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

### Recommendation

It is recommended to refactor the liquidation validation flow to separate concerns and remove set overlap:

1. **Use explicit denom categories:**

- Keep `self.collaterals` for collateral-only tokens, derived from `Config`.`collateral_ratios`.
- Introduce a dedicated **debt token set** derived from `self.debts[*].item.denom()`.

2. **Modify** `ensure_collateral_decrease`**:**

- Apply only to tokens in the collateral set, **excluding** debt denoms.
- Increases in debt denoms during liquidation should not trigger failure.

3. **Revise** `validate_liquidation_execute_step`**:**

- Convert `spent_collateral` and `received_debt` to a common price unit (using the same oracle data used in LTV).
- Compute `slip = value_spent / value_received` (with guards for zero received; use explicit error rather than divide-by-zero). If `value_received == 0` → explicit error (`NoDebtReceived`) rather than relying on vacuous checks.
- Apply `max_slip` on value terms, not token counts.

### Remediation Comment

**SOLVED**: This issue is resolved by the **Rujira team** through a refactor of the `validate_liquidation_execute_step` function. That function was replaced by the `validate_liquidation()` and `debt()` functions to enable tracking of all debt denominations (avoiding unknown denoms) and to prevent the issues described in the finding.

### Remediation Hash

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

## 7.2 LIQUIDATION DOS WHEN FINAL LTV DROPS BELOW ADJUSTMENT_THRESHOLD

// MEDIUM

### Description

A logical flaw in the liquidation loop (`DoLiquidate`) of the `rujira-ghost-credit` contract causes **valid liquidation routes to revert** when the resulting Loan-to-Value (LTV) ratio falls below the `adjustment_threshold`.

During liquidation, the protocol enforces an LTV ***safety band*** `[adjustment_threshold, liquidation_threshold)`. After executing an `Execute + Repay` sequence, if the resulting account state becomes “too safe” (i.e., **LTV < adjustment_threshold**), the engine returns an `AccountSafe` error instead of treating this as a successful liquidation.

This behavior blocks legitimate liquidations where a solver or swap slightly **over-repays** or **over-swaps** the debt, leaving the account under-leveraged. In such cases, the transaction completes correctly from an economic standpoint (debt reduced, LTV well below the liquidation threshold), but it reverts due to the misplaced safety condition.

The final `check_unsafe(adjustment_threshold)` is treated as a mandatory condition even during liquidation. While `check_unsafe` is appropriate for owner-initiated actions (`Account` flow), applying it in the liquidation context is incorrect; it makes “too-safe” accounts revert, effectively converting a successful liquidation into a false failure.

### Code Location

Code of `ExecuteMsg::DoLiquidate` message from `src/contract.rs` :

```rust
ExecuteMsg::DoLiquidate {
    idx,
    mut queue,
    payload,
} => {
    ensure_eq!(info.sender, ca, ContractError::Unauthorized {});
    let account = CreditAccount::load(deps.as_ref(), &config, &ca, idx)?;
    let original_account: CreditAccount = from_json(&payload)?;

    let check = account
        .ensure_collateral_decrease(&original_account)
        // Check safe against the liquidation threshold
        .and_then(|_| account.check_safe(&config.liquidation_threshold))
        // Check we've not gone below the adjustment threshold
        .and_then(|_| account.check_unsafe(&config.adjustment_threshold));
    match (queue.pop(), check) {
        (_, Ok(())) => {
            // We're done and we're safe. Return regardless of queue state
            Ok(Response::default())
        }
        (None, Err(err)) => {
            // We're done and the Account hasn't passed checks. Fail
            Err(err)
        }
        (Some((msg, is_preference)), Err(_)) => {
            // Not safe, more messages to go. Continue
            Ok(execute_liquidate(
                deps.as_ref(),
                env.clone(),
                info,
                &config,
                msg,
                &account,
                if is_preference {
                    REPLY_ID_PREFERENCE
                } else {
                    REPLY_ID_LIQUIDATOR
                },
            )?
            .add_message(
                ExecuteMsg::DoLiquidate {
                    idx: account.idx,
                    queue,
                    payload,
                }
                .call(&ca)?,
            ))
        }
    }
}
```

### Proof of Concept

### Scenario:

A liquidation route composed of Execute (swap collateral → debt token) followed by Repay leaves the account “too safe.” Because the engine enforces the band `adjustment_threshold ≤ LTV < liquidation_threshold` during liquidation, if the final LTV drops below `adjustment_threshold`, DoLiquidate returns an “Account Safe” error instead of succeeding. This causes an operational DoS for valid routes when the solver slightly over-swaps or over-repays.

### Test Code:

```rust
#[test]
fn liquidation_aborts_account_safe_after_repay() {
    // PoC: When the debt denom is also whitelisted as collateral, a liquidation
    // Execute that swaps collateral into the debt token increases the
    // collateral-only balance and causes ensure_collateral_decrease to fail,
    // aborting before Repay.
    let mut app = mock_rujira_app();
    let owner = app.api().addr_make("owner");
    let ctx = setup(&mut app, &owner);

    // Fund account with BTC collateral (1 BTC)
    app.send_tokens(
        owner.clone(),
        ctx.account.account.contract(),
        &[coin(100_000_000, BTC)],
    )
    .unwrap();

    // Borrow USDC (USDC is whitelisted as collateral in setup with ratio 0.9)
    let account = ctx.ghost_credit.query_account(&app, &ctx.account.idx);
    // Borrow enough to exceed collateral after price drop, but small vs initial collateral
    ctx.ghost_credit
        .account_borrow(&mut app, &account, 100_000_000_000u128, USDC) // 100,000 USDC
        .unwrap();

    // Remove borrowed USDC from the account so it doesn't count as collateral
    // and worsen LTV; send to owner wallet
    ctx.ghost_credit
        .account_send(
            &mut app,
            &account,
            100_000_000_000u128,
            USDC,
            &owner,
        )
        .unwrap();

    // Re-query after send
    let _account_after_send = ctx.ghost_credit.query_account(&app, &ctx.account.idx);

    // Push prices so the account becomes liquidatable
    app.init_modules(|router, _api, _storage| {
        router
            .stargate
            .with_prices(vec![
                ("BTC", Decimal::from_str("1000").unwrap()),
                ("USDC", Decimal::from_str("1.0").unwrap()),
            ]);
    });

    let account = ctx.ghost_credit.query_account(&app, &ctx.account.idx);
    assert!(account.ltv > Decimal::one());

    // Liquidation route: Execute swap BTC->USDC then Repay USDC
    // Because USDC is treated as collateral, the Execute increases
    // collateral-only balance and the next iteration fails on
    // ensure_collateral_decrease before reaching Repay.
    let err = ctx
        .ghost_credit
        .liquidate_execute_repay(
            &mut app,
            &account,
            ctx.fin_btc_usdc.addr(),
            fin::ExecuteMsg::Swap(fin::SwapRequest {
                min_return: None,
                to: None,
                callback: None,
            }),
            coins(500_000, BTC),
            USDC,
        )
        .unwrap_err();

    // Assert we aborted due to Account Safe
    let msg = format!("{err:?}");
    assert!(msg.contains("Account Safe"), "unexpected error: {msg}");
}
```

### Result:

Running the PoC produces an error response with “Account Safe” after the Execute+Repay sequence. The final LTV is observed below `adjustment_threshold` while below `liquidation_threshold`, and the liquidation aborts despite having reduced debt, confirming a DoS path triggered by over-correction rather than insolvency.

### BVSS

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

### Recommendation

It is recommended to **treat “Account Safe” as a success** during liquidation and adjust the LTV checks accordingly.

### Remediation Comment

**NOT APPLICABLE**: It is considered by the **Rujira team** that the `adjustment_threshold` validation constitutes a design flaw; therefore, this issue is deemed not applicable.

## 7.3 MISSING VALIDATION FOR LIQUIDATION PREFERENCES

// LOW

### Description

The `rujira-ghost-credit` contract allows users to configure **liquidation preferences** that control the order in which collaterals can be spent or liquidated during a liquidation event.

These preferences are set via `AccountMsg::SetPreferenceOrder` and `AccountMsg::SetPreferenceMsgs`, and internally stored by calling `account::set_preference_order` and `account::set_preference_msgs` into `BTreeMap<String, String>` and `Vec<String>` structures, respectively.

However, the current implementation **does not perform any validation** on the preference data at the time of storage. This means the system accepts arbitrary and potentially inconsistent configurations.

The lack of validation creates multiple failure modes that can degrade both functionality and security:

| CASE | DESCRIPTION | IMPACT |
|---|---|---|
| **Cyclic preferences** | A cycle such as `A → B` and `B → A` produces an unresolvable dependency, making it impossible to liquidate any collateral until the user manually clears their preferences. | Liquidation stalls indefinitely → **Denial of Service (DoS)** |
| **Self-references** | A self-dependency like `A → A` makes that denom unspendable under all conditions, as the validator will always see the dependency as unmet. | The affected collateral becomes permanently locked, preventing any liquidation or repayment that depends on it → **asset-level DoS** |
| **Invalid or non-whitelisted denoms** | If a user sets a required denom that is not part of the collateral whitelist (or has ratio `= 0`), that dependency is ignored during liquidation. | A supposed "protected" asset can be spent earlier than intended, **bypassing the user's intended protection policy** |
| **Missing transitive enforcement** | If preferences form a longer chain (`X → Y → Z`), the system enforces only the immediate relation (`X` depends on `Y`). Once `Y` is spent, `X` becomes spendable even if `Z` remains. | Breaks intended liquidation order, **weakening user control and predictability** |

At liquidation time, `validate_liquidation_execute_step` calls `order.validate(...)`, which checks only the *immediate* dependency. The net result is that liquidation preferences, instead of providing predictable execution order, can lead to deadlocks, unexpected liquidation sequences, or ignored protection rules; all without any indication to the user.

### BVSS

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

### Recommendation

It is recommended to implement **comprehensive validation** for preferences both at save time and during liquidation:

1. **At save time (`set_preference_order` / `set_preference_msgs`):**

- Verify that all referenced denoms exist in `Config.collateral_ratios` and have non-zero ratios.
- Reject any self-references (`A → A`).
- Detect and reject cycles by verifying the map forms a **directed acyclic graph (DAG)** before committing.
- Emit explicit error messages for invalid configurations.

2. **At liquidation time (`validate_liquidation_execute_step`):**

- Extend validation to traverse transitive dependencies (full chain) instead of only the immediate one.
- Reject liquidation steps involving denoms not in the current collateral whitelist.
- Provide clear failure reasons (e.g., "cyclic preference", "invalid denom", or "non-whitelisted dependency").

3. **Optional Enhancements:**

- Limit the total number of preferences per account to bound gas and storage usage.
- Add a `clear_preferences()` entrypoint for users to safely reset misconfigured preferences.

### Remediation Comment

**SOLVED**: The **Rujira team** has solved this finding along the following commits:

- beaf062: check the cycles and self-references.
- 99a951b: check the transitive references.
- 5ade419: add a maximum number of references.
- dc3f0f5: validate the whitelisted denoms.

### Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/beaf062d0c964fe59d69af23ffeb9c2109a63971

## 7.4 MISSING ENFORCEMENT OF CRITICAL CONFIGURATION INVARIANTS

// LOW

### Description

The `rujira-ghost-credit` contract accepts unsafe configuration values during `instantiate` and `sudo::UpdateConfig` because the `Config::validate()` function performs no checks on economic and security-critical parameters.

A privileged governance/operator or a misconfiguration at deployment time can set:

- Inverted thresholds: `adjustment_threshold ≥ liquidation_threshold`
- Excessive fees: `fee_liquidation + fee_liquidator ≥ 1` (repay becomes zero)
- Invalid slip bounds: `liquidation_max_slip outside [0,1]`
- Collateral config: empty `collateral_ratios` or ratios set to 0. Collateral ratios must be strictly within `[0,1]`.
- Untrusted deployment: injecting an unverified `code_id` for account instantiation

These misconfigurations can lead to severe consequences:

- **DoS of borrow/repay flows** (fees ≥ 100% or invalid repay amounts causing panics or no-op repays).
- **Incorrect LTV semantics / insolvency** (inverted thresholds, underflowing liquidation windows).
- **Over-liquidation or blocked liquidation** (invalid `liquidation_max` or misordered thresholds).
- **Asset exfiltration or arbitrary code execution** (malicious `code_id` used for `instantiate2` → rogue account implementation).
- **Operational instability and unexpected gas/behaviour** (empty collateral sets, many denoms, or inconsistent Vault mappings).

These problems are systemic and affect both safety for individual users (funds and fairness) and systemic health of the protocol (liquidation correctness, ability to recover positions).

### Code Location

Code of `validate` function from `src/config.rs` file:

Copy Code

```
pub fn validate(&self) -> Result<(), ContractError> {
    Ok(())
}
```

### BVSS

AO:S/AC:L/AX:M/R:N/S:U/C:N/A:C/I:C/D:C/Y:N (2.0)

### Recommendation

It is recommended to implement strict validation for all configuration fields and call this validation on both `instantiate` and `sudo::UpdateConfig`. Specifically:

- Enforce `0 < adjustment_threshold < liquidation_threshold` and `liquidation_threshold ≤ Decimal::one()`.
- Enforce `0 ≤ liquidation_max && liquidation_max ≤ liquidation_threshold`.
- Enforce `fee_liquidation >= 0`, `fee_liquidator >= 0` and `(fee_liquidation + fee_liquidator) < Decimal::one()`.
- Require `fee_address` be non-empty and a valid address.
- Require `collateral_ratios` be non-empty and `0 < ratio < 1`; and limit maximum number of collateral denoms to a reasonable bound.
- Validate `code_id` against a trusted allowlist.
- Fail fast: return a descriptive `ContractError::InvalidConfig(...)` with helpful message for operators and block the update/instantiate.

### Remediation Comment

**SOLVED**: The **Rujira team** has solved this issue by implementing the recommended validations.

### Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/ed05d6fa8830949e1d171383af700857a2540fb0

## 7.5 MISSING VALIDATION IN SETVAULT AND SETCOLLATERAL

// INFORMATIONAL

### Description

The `rujira-ghost-credit` contract's registry logic does not enforce key invariants when updating configuration via the `SudoMsg::SetVault` and `SudoMsg::SetCollateral` messages. These two entry points directly affect the mapping between borrowable assets and eligible collateral, but currently lack consistency checks that ensure all borrow denoms are properly represented and safely configured in the collateral whitelist.

`SetVault`

- Allows creating or updating a borrow mapping (`borrow_denom → vault_addr`) **without verifying** that the denom belongs to the designated Vault.
- There is **no explicit whitelist** of allowed borrowable tokens; the contract currently reuses `collateral_ratios` as an implicit source of truth.
- Mapping changes can occur while existing delegated debt still exists, which can strand or reassign borrower positions to an inconsistent vault.

`SetCollateral`

- Allows governance to insert arbitrary `collateral_ratios` entries, including **zero or >1 ratios**.
- Permits governance to clear or zero out all ratios, leaving the map empty, which breaks liquidation and LTV computation.

Together, these missing checks compromise the internal consistency between the borrow and collateral domains, enabling configuration states where liquidations misbehave or accounts cannot be safely resolved.

### BVSS

AO:S/AC:L/AX:L/R:N/S:U/C:N/A:N/I:H/D:N/Y:N (1.5)

### Recommendation

It is recommended to implement strong invariants and two-way validation between `SetVault`, `SetCollateral`, and `Config::validate` to prevent inconsistent or unsafe configurations:

`SetVault`

- **Require explicit whitelisting of the target borrow denom:**
  - Introduce a dedicated `debt_whitelist` separate from `collateral_ratios`.
  - Reject any mapping for denoms not in this whitelist.
- **Use a two-phase mapping update (propose → confirm):** Block confirmation if there is existing delegated debt in the old vault, or implement a safe migration/repay procedure.

`SetCollateral`

- **Enforce ratios strictly in [0, 1]:** Reject any entry with ratio > 1.
- **Prevent governance from zeroing all ratios or leaving the map empty:** The protocol must always have at least one valid collateral denom.
- **Disallow modifying ratios to active borrow denoms** (or, if allowed, handle them specially).

### Remediation Comment

**SOLVED**: The **Rujira team** has solved this issue by implementing the recommended fixes. Part of the fix has been implemented in the commit dc3f0f5.

### Remediation Hash

https://gitlab.com/thorchain/rujira/-/commit/07bf1750944c10c238e0ffe5e2ab3310d6525b7e

## 7.6 DIVISION-BY-ZERO IN ADJUSTED LTV

// INFORMATIONAL

### Description

The `rujira-ghost-credit` contract's Loan-to-Value (LTV) calculation divides total debt by adjusted collateral without protecting against cases where the total collateral equals zero. When `debt > 0` and `collateral == 0`, the call to `Decimal::div()` in the LTV calculation panics at runtime.

This situation can occur under some low-likelihood conditions:

- A governance `sudo::SetCollateral` or misconfiguration clears `collateral_ratios`, effectively setting the adjusted collateral to zero.
- Oracle outage/zero pricing returns 0 USD for all collateral denoms, causing them to be skipped and resulting in adjusted collateral 0.

When this happens, any function that queries or computes LTV (including `CheckAccount` or `CheckLiquidate`) **panics and reverts**, causing:

- **Denial-of-Service**: the account (and sometimes the entire liquidation flow) becomes unprocessable until parameters are restored.
- **Stuck state**: critical operations (borrow, repay, liquidation) fail, effectively freezing positions.
- **Inconsistent logic**: parts of the contract assume a valid LTV value and cannot handle a panic from arithmetic division.

### Code Location

Code of `adjusted_ltv` function from `src/account.rs` file:

```rust
pub fn adjusted_ltv(&self) -> Decimal {
    let collateral = self
        .collaterals
        .iter()
        .map(|x| x.value_adjusted)
        .collect::<Vec<Decimal>>()
        .into_iter()
        .reduce(|a, b| a + b)
        .unwrap_or_default();

    let debt = self
        .debts
        .iter()
        .map(|x| x.value)
        .collect::<Vec<Decimal>>()
        .into_iter()
        .reduce(|a, b| a + b)
        .unwrap_or_default();

    if debt.is_zero() {
        return Decimal::zero();
    }

    debt.div(collateral)
}
```

#### BVSS

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

#### Recommendation

It is recommended to **gracefully handle zero-collateral states** to avoid runtime panics and ensure deterministic contract behavior.

#### Remediation Comment

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

## 7.7 MISSING REPAY MESSAGE ALLOWS INCOMPLETE SETTLEMENT FLOW

// INFORMATIONAL

#### Description

The Repay flow relies on receiving a sequence of messages that must **include a final `Repay` operation** to complete the borrower's debt lifecycle. If an external integration omits the `Repay` message when constructing the execution vector, the vault will not settle the outstanding debt and may remain in an incomplete accounting state until a later interaction occurs.

This issue does not impact protocol security directly, as the core logic remains consistent and no loss of funds occurs. However, it represents an **integration risk**: external callers must construct correct message sequences, and the contract does not enforce or validate that a final `Repay` is present. This can lead to unexpected UX inconsistencies, delayed debt settlement, or confusion during liquidation/settlement flows.

#### Score

(0.0)

#### Recommendation

It is recommended that safeguards be implemented in the vault, such as validation of expected settlement invariants and a helper entrypoint that automatically appends the final `Repay` step.

#### Remediation Comment

**ACKNOWLEDGED**: The Rujira team has 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.