NAMI Protocol Rujira Index Product - THORChain / Rujira

Prepared by: **HALBORN** Last Updated 08/11/2025

Date of Engagement: May 12th, 2025 - May 16th, 2025

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
17	0	1	4	7	5

1. Introduction

`THORChain` engaged Halborn to conduct a security assessment of the `Nami Index` contracts, beginning on May 13th, 2025 and ending on May 19th, 2025. This security assessment was scoped to the smart contracts in the Nami GitHub repository. Commit hashes and further details can be found in the **Sources** section of this report.

`Nami Index` is a protocol that enables the creation and management of tokenized index vaults on top of Thorchain. It allows users to deposit a single asset and receive diversified exposure across multiple assets according to predefined weights. The protocol handles rebalancing, swapping, and accounting through a set of CosmWasm smart contracts that interact with Thorchain's liquidity network.

2. Caveats

All remediations described in this report were completed prior to the following commit, which serves as a consolidated snapshot of the final codebase. Although remediations may have been implemented across multiple earlier commits, this single commit includes **all changes** and can be used for verification purposes:

833f72787ee26d141008d2bb620cf324436a6d08

Below are the checksums for the deployed contracts:

Contract	Checksum
nami-index-nav	0ff970150655cb53687b6b478c6540f3deaec5ce06063ef64d3e02a519d6b56b
nami-index-fixed	63dd9426926704db38dc25b6c1830d202bbad7d92d8d298056cd7e0de3efd9ce
nami-index-entry-adapter	e9927b93feeef8fd2e8dcdca4695dddd38d0a832d8e62ad2c0e9cf2826a4f61a
nami-affiliate	223ea20a4463696fe32b23f845e9f90ae5c83ef0175894a4b0cec114b7dd4b26

3. 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 have been partially addressed by the `Nami team`. The main ones were the following:

- `Restrict affiliate fee injection by enforcing a whitelist and storing fee parameters on-chain.`
- `Prevent fee evasion by accumulating fractional fees in state or using ceiling rounding methods.`
- `Ensure deposit logic uses up-to-date amount_per_share by reordering rebalances.`
- `Limit Callback execution to known swap contracts by verifying the sender against the allocation map.`
- `Isolate and track the exact quote_denom amount received in the first swap before performing the second during reallocation.`
- `Enforce automatic rebalancing after relevant events or based on elapsed time in NAV index.`
- `Support batched allocation updates to verify that total weights sum to 1 within a single atomic call.`
- `Use separate min_return values for multi-step swaps or validate the final output instead of individual steps.`
- `Validate that the registered denom matches the swap contract's base denom and is not the quote token.`
- `Reject duplicate allocation entries by checking for existing denom values before insertion.`

4. 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.

5. 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.

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):

TABLE OF CONTENTS

1. Introduction
2. Caveats
3. Assessment summary
4. Test approach and methodology
5. Risk methodology
6. Scope
7. Assessment summary & findings overview
8. Findings & Tech Details
    8.1 Deposit value not multiplied by token price
    8.2 Unrestricted affiliate fee injection enables fund redirection attacks
    8.3 Fee system vulnerable to rounding-based evasion attacks
    8.4 Deposit uses outdated amount_per_share after prior withdraw
    8.5 Unit mismatches in withdraw slippage flow
    8.6 Callback can be triggered by unauthorized external contracts
    8.7 Reallocation uses full quote_denom balance without isolating source
    8.8 Missing validation that map key matches base denom in multiple contracts
    8.9 Allocation entries can be silently overwritten
    8.10 Reuse of min_return across independent swaps in reallocation
    8.11 Lack of automatic rebalance enforcement may lead to stale or unbalanced portfolios
    8.12 Weight sum cannot be verified with individual allocation updates
    8.13 Lack of documentation across the codebase
    8.14 Inefficient swap strategy in deposit flow
    8.15 Burn + mint fee shares could be simplified via transfer
    8.16 Receipt token denoms are identical across different index types
    8.17 Inconsistent slippage semantics across withdraw and rebalance functions

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

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: nami-contracts

(b) Assessed Commit ID: 4dac9fd

(c) Items in scope:

- contracts/nami-index-nav/src/events.rs
- contracts/nami-index-nav/src/error.rs
- contracts/nami-index-nav/src/config.rs

↓ Expand ↓

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

REMEDIATION COMMIT ID:

- fc81a64
- 9b2da1b
- 370a7a6
- 34b790d
- 6f90423
- f485834
- 44d71d0
- bec2b78
- 5be715d
- 26c3413
- dd42566
- 39fb050
- e4a872b
- 4caa0b4

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

7. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	1	4	7	5

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
DEPOSIT VALUE NOT MULTIPLIED BY TOKEN PRICE	HIGH	SOLVED - 06/24/2025
UNRESTRICTED AFFILIATE FEE INJECTION ENABLES FUND REDIRECTION ATTACKS	MEDIUM	PARTIALLY SOLVED - 05/28/2025
FEE SYSTEM VULNERABLE TO ROUNDING-BASED EVASION ATTACKS	MEDIUM	SOLVED - 05/27/2025
DEPOSIT USES OUTDATED AMOUNT_PER_SHARE AFTER PRIOR WITHDRAW	MEDIUM	SOLVED - 05/28/2025
UNIT MISMATCHES IN WITHDRAW SLIPPAGE FLOW	MEDIUM	SOLVED - 08/08/2025
CALLBACK CAN BE TRIGGERED BY UNAUTHORIZED EXTERNAL CONTRACTS	LOW	SOLVED - 05/27/2025
REALLOCATION USES FULL QUOTE_DENOM BALANCE WITHOUT ISOLATING SOURCE	LOW	SOLVED - 05/27/2025
MISSING VALIDATION THAT MAP KEY MATCHES BASE DENOM IN MULTIPLE CONTRACTS	LOW	SOLVED - 05/27/2025
ALLOCATION ENTRIES CAN BE SILENTLY OVERWRITTEN	LOW	SOLVED - 05/28/2025
REUSE OF MIN_RETURN ACROSS INDEPENDENT SWAPS IN REALLOCATION	LOW	SOLVED - 05/27/2025
LACK OF AUTOMATIC REBALANCE ENFORCEMENT MAY LEAD TO STALE OR UNBALANCED PORTFOLIOS	LOW	RISK ACCEPTED - 05/28/2025
WEIGHT SUM CANNOT BE VERIFIED WITH INDIVIDUAL ALLOCATION UPDATES	LOW	SOLVED - 05/28/2025
LACK OF DOCUMENTATION ACROSS THE CODEBASE	INFORMATIONAL	ACKNOWLEDGED - 05/29/2025
INEFFICIENT SWAP STRATEGY IN DEPOSIT FLOW	INFORMATIONAL	ACKNOWLEDGED - 05/28/2025
BURN + MINT FEE SHARES COULD BE SIMPLIFIED VIA TRANSFER	INFORMATIONAL	SOLVED - 05/27/2025
RECEIPT TOKEN DENOMS ARE IDENTICAL ACROSS DIFFERENT INDEX TYPES	INFORMATIONAL	SOLVED - 05/27/2025
INCONSISTENT SLIPPAGE SEMANTICS ACROSS WITHDRAW AND REBALANCE FUNCTIONS	INFORMATIONAL	SOLVED - 05/27/2025

8. FINDINGS & TECH DETAILS

8.1 DEPOSIT VALUE NOT MULTIPLIED BY TOKEN PRICE

// HIGH

Description

The `Deposit` branch mints shares by dividing the raw token amount by the NAV, but it never multiplies that amount by the current oracle price of the `quote_denom`.
When the `quote_denom` is not a 1-to-1 stable token, this mis-prices the deposit and over (or under) mints shares, distorting the vault's accounting and enabling unfair share allocation.

Code Location

Code of `execute` function from contracts/nami-index-nav/src/contract.rs file.

Copy Code

```
match msg {
    ExecuteMsg::Deposit {} => {
        let amount = must_pay(&info, &config.quote_denom)?;
        let nav = vault.nav(deps.storage, Some(amount), rcpt.supply(deps.querier)?)?;
        let minted = Decimal::from_ratio(amount, Uint128::one())
            .checked_div(nav)?
            .to_uint_floor();
        if minted.is_zero() {
            return Err(ContractError::Std(StdError::generic_err(
                "Minted amount is zero",
            )));
        }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:H (7.5)

Recommendation

It is recommended to multiply the deposited amount by the latest oracle price of the `quote_denom` before dividing by the NAV, ensuring the calculation is performed in consistent units.

Remediation Comment

SOLVED: The `Nami team` has resolved this issue by including the token price in the deposit amount calculation.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/fc81a643fecc766cb8c05ab38e7fc83e312ee0c7

8.2 UNRESTRICTED AFFILIATE FEE INJECTION ENABLES FUND REDIRECTION ATTACKS

// MEDIUM

Description

The `nami-affiliate` contract introduces a trust-based design flaw by allowing arbitrary callers to specify both the affiliate address and the fee basis points (`affiliate` input parameter) in the `ExecuteMsg::Execute` message. This design enables a range of front-end hijack or phishing scenarios, in which users unknowingly interact with malicious UIs that inject attacker-controlled affiliate addresses and inflated fees.

As a result, unsuspecting users may unintentionally redirect a portion (or in extreme cases, the entirety) of their funds to unauthorized third parties. Since the contract does not enforce any kind of whitelist, approval mechanism, or validation logic for the affiliate address, there is no on-chain safeguard preventing this redirection.

Code Location

Code of `execute` function from contracts/nami-affiliate/src/contract.rs file.

```
match msg {
    ExecuteMsg::Execute {
        contract_addr,
        msg,
        affiliate,
    } => {
        ensure!(!info.funds.is_empty(), ContractError::InsufficientFunds {});
        ensure!(
            WHITELIST.has(
                deps.storage,
                deps.api.addr_validate(contract_addr.as_str())?
            ),
            ContractError::Unauthorized {}
        );
        let mut net_funds = info.funds.clone();
        let mut response =
            Response::new().add_event(execute_event(contract_addr.clone(), affiliate.clone()));

        if let Some((addr, bps)) = affiliate {
            ensure!(bps <= 10_000u16, ContractError::InvalidAffiliateFee {});
            let pairs: Vec<(Coin, Coin)> = info
                .funds
                .iter()
                .map(|c| -> Result<(Coin, Coin), ContractError> {
                    let fee = c
                        .amount
                        .checked_mul(bps.into())?
                        .checked_div(10_000u16.into())?;
                    let net = c.amount.checked_sub(fee)?;
                    Ok((
                        coin(net.u128(), c.denom.clone()),
                        coin(fee.u128(), c.denom.clone()),
                    ))
                })
                .collect::<Result<_, _>>()?;

            let (nets, mut fees): (Vec<Coin>, Vec<Coin>) = pairs.into_iter().unzip();
            fees.retain(|c| !c.amount.is_zero());

            if !fees.is_empty() {
                response = response.add_message(BankMsg::Send {
                    to_address: addr.clone(),
                    amount: fees,
                });
            }

            net_funds = nets;
        }
```

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:M/A:N/I:N/D:M/Y:N (6.3)

Recommendation

It is recommended to introduce a whitelist or registry of approved affiliate addresses that can receive fees. The fee percentage (bps) should be stored on-chain and retrieved from state based on the caller or affiliate identity, rather than being supplied directly in the transaction input. This change would eliminate the risk of fund redirection via untrusted interfaces and reinforce user protection against affiliate fee abuse.

Remediation Comment

PARTIALLY SOLVED: The **Nami team** has increased protection against this vulnerability by adding a `maximum fee cap`, hardcoded in the Config. However, this does not eliminate the risk of a hijacked UI replacing the affiliate address, potentially redirecting the fees to an attacker.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/9b2da1bef20f33f19657c308ba085400706d6b77

8.3 FEE SYSTEM VULNERABLE TO ROUNDING-BASED EVASION ATTACKS

// MEDIUM

Description

The current fee mechanism in the Nami protocol used for both AUM (Assets Under Management) and transaction fees can be bypassed through rounding behavior during high-frequency or micro-amount operations.

Specifically, both `aum_fee` and `tx_fee` functions use `.to_uint_floor()` to convert Decimal values to integers. When the calculated fee is less than 1 token unit, it is silently rounded down to zero and discarded. Despite this, `last_accrual_time` is still updated as if the fee had been collected. This allows attackers to repeatedly perform very small interactions (e.g., automated bots calling many micro-withdrawals), causing both AUM and transaction fees to round to zero on every call, effectively avoiding any fee payment.

Over time, this can significantly reduce protocol revenue and break the expected economic model, especially if abused at scale.

Code Location

Code of `aum_fee` function from packages/nami-rs/src/fee_manager.rs file.

```
pub fn aum_fee(
    &mut self,
    now: Timestamp,
    total_supply: Uint128,
) -> Result<Uint128, FeeManagerError> {
    let rate = self.rates.management.unwrap_or_default();

    let elapsed = now
        .seconds()
        .saturating_sub(self.last_accrual_time.seconds());

    if elapsed == 0 {
        return Ok(Uint128::zero());
    }

    const SECS_PER_YEAR: u64 = 31_557_600;
    let year_fraction = Decimal::from_ratio(elapsed as u128, SECS_PER_YEAR as u128);
    let effective = rate.checked_mul(year_fraction)?;

    let fee_amount = Decimal::from_ratio(total_supply, Uint128::one())
        .checked_mul(effective)?
        .to_uint_floor();

    self.last_accrual_time = now;
    Ok(fee_amount)
}
```

Code of `tx_fee` function from packages/nami-rs/src/fee_manager.rs file.

```
pub fn tx_fee(&self, amount: Uint128) -> Result<(Uint128, Uint128), FeeManagerError> {
    let rate = self.rates.transaction.unwrap_or_default();

    let fee = Decimal::from_ratio(amount, Uint128::one())
        .checked_mul(rate)?
        .to_uint_floor();
    let net = amount.checked_sub(fee)?;
    Ok((net, fee))
```

Proof of Concept

Scenario

This test simulates a scenario where a user makes a deposit into a vault that has both management (AUM) and transaction fees enabled at 1%. The deposit value is intentionally low enough that the resulting fee calculations produce amounts below 1 unit per asset, which are then rounded down to zero.

One year of blockchain time is simulated to pass, allowing AUM fees to accrue. The user then performs a minimal withdrawal of 1 share. The goal is to confirm that the total_fees vector is empty, meaning no fees were collected.

Test

```
fn test_fee_rounding_to_zero_skips_fee_accrual() {

    // Set up a vault with AUM and TX fees
    let balances = vec![
        (
            "user",
            vec![
                coin(10_000_000, "nami"),
                coin(10_000_000, "auto"),
                coin(10_000_000, "lqdy"),
            ],
        ),
        (
            "owner",
            vec![
                coin(100_000_000, "nami"),
                coin(100_000_000, "auto"),
                coin(100_000_000, "lqdy"),
                coin(100_000_000, "eth-usdc"),
            ],
        ),
    ];

    let mut test_env = index::setup(
        balances,
        "eth-usdc".to_string(),
        vec![
            ("nami".to_string(), Uint128::from(100_000u128)),
            ("auto".to_string(), Uint128::from(100_000u128)),
            ("lqdy".to_string(), Uint128::from(100_000u128)),
        ],
        Some(Decimal::percent(1)), // AUM fee
        Some(Decimal::percent(1)), // TX fee
    );

    let rcpt_denom = format!("x/nami-index-{}-rcpt", test_env.index.address);

    // First deposit - small enough so that TX and AUM fee round to 0
    test_env
        .index
        .execute_deposit(
            &mut test_env.app,
            "user",
            vec![
                coin(100_000_0, "nami"),
                coin(100_000_0, "auto"),
                coin(100_000_0, "lqdy"),
            ],
        )
        .unwrap();

    // Move time forward 1 year
    const SECS_PER_YEAR: u64 = 31_557_600;
    test_env.app.update_block(|block| {
        block.height += 1;
        block.time = block.time.plus_seconds(SECS_PER_YEAR);
    });

    // Withdraw 1 share - fee should be 0 due to rounding
    let _ = test_env
        .index
        .execute_withdraw(
            &mut test_env.app,
            "user",
            vec![coin(1, rcpt_denom.clone())],
        )
        .unwrap();

    // Now query vault status
    let status = test_env.index.query_status(&mut test_env.app).unwrap();
    let shares = status.total_shares;
    println!("----> Status:");
    println!("Total shares: {}", shares);
    println!("Allocations:");
    for (denom, weight) in status.allocation.iter() {
        println!("- {}: {}", denom, weight);
    }

    let config = test_env.index.query_config(&mut test_env.app).unwrap();

    let total_fees = test_env
    .app
    .query_all_balances( config.fee_collector.as_str(), false);


    println!("Total fees collected: {:?}", total_fees);
    assert_eq!(total_fees.len(), 0);

}
```

Result

The test passes and confirms that both AUM and transaction fees were effectively skipped due to rounding.

BVSS

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

Recommendation

It is recommended to accumulate fractional fees in contract state using a Decimal field (e.g., `pending_aum` or `pending_tx`). Only once the accumulated amount surpasses the 1-token threshold should shares be minted. Alternatively, using `.to_uint_ceil()` would ensure at least 1 share is collected whenever a non-zero fee is due. For further mitigation, the protocol could enforce a `minimum fee` or implement a `cooldown period and minimum transaction size` to prevent abuse through micro-interactions.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by rounding fractional fees up (using ceiling rounding) and setting a minimum fee of 1 whenever the calculated fee is greater than zero.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/370a7a69205c9e019b424c0cb7ff17d06bdd0f41

8.4 DEPOSIT USES OUTDATED AMOUNT_PER_SHARE AFTER PRIOR WITHDRAW

// MEDIUM

Description

In the `nami-index-fixed` contract, the `Withdraw` operation triggers a rebalance **before** calculating and distributing output funds. However, if a `Deposit` is executed immediately after, the contract still uses the pre-rebalance `amount_per_share` value to calculate how many shares to mint for the depositor, because the Deposit logic only performs the rebalance **after** completing the share calculation.

This creates a subtle but important inconsistency: the deposit is evaluated using outdated economic proportions that do not reflect the current vault state. As a result, the user may receive more shares than warranted, especially if the rebalance significantly changed the value of the underlying assets.

One possible solution would be to move the rebalance to the beginning of the Deposit execution. However, this approach may conflict with external systems—such as nami-index-entry-adapter—which rely on pre-rebalance `amount_per_share` values to proportionally prepare token inputs.

This behaviour introduces opportunities for front-running, where users could time deposits right after large withdrawals to exploit stale accounting data and gain an unfair advantage.

Code Location

Code of `execute` function from contracts/nami-index-fixed/src/contract.rs file.

```rust
match msg {
    ExecuteMsg::Deposit {} => {
        let amount = Vault::deposit(deps.storage, info.funds.clone())?;
        response = response.add_event(event_deposit(
            info.sender.clone(),
            info.funds.clone(),
            amount,
        ));
        vault.rebalance(deps.storage, rcpt.supply(deps.querier)? + aum_fee + amount)?;
        response = response.add_message(rcpt.mint_msg(amount, info.sender));
    }
    ExecuteMsg::Withdraw {} => {
        let amount = must_pay(&info, rcpt.denom().as_str())?;
        let (net, burn_fee) = fee_manager.tx_fee(amount)?;
        vault.rebalance(deps.storage, rcpt.supply(deps.querier)? + aum_fee)?;
        let withdraw_funds = Vault::withdraw(deps.storage, net)?;
        response = response.add_event(event_withdraw(
            info.sender.clone(),
            withdraw_funds.clone(),
            amount,
        ));
        let send_msg = BankMsg::Send {
            to_address: info.sender.to_string(),
            amount: withdraw_funds,
        };
        response = response.add_message(rcpt.burn_msg(amount));
        if burn_fee.gt(&Uint128::zero()) {
            response =
                response.add_message(rcpt.mint_msg(burn_fee, config.fee_collector.clone()));
        }
        response = response.add_message(send_msg);
    }
```

Proof of Concept

Scenario

This test demonstrates how a deposit can be incorrectly evaluated using a stale amount_per_share if a rebalance is not executed between a previous withdrawal and the new deposit. This inconsistency leads to an over-minting of shares. To enable manual rebalancing and simulate a correct accounting update, an additional helper function named `execute_run` was added to the mock_nami_index_fixed.rs file.

A vault is initialized with 3 assets (nami, auto, lqdy), each with equal weights. The user performs an initial deposit, which mints some receipt tokens, establishing the initial amount_per_share for each asset. A full year of simulated block time is advanced to accrue the AUM fee.

Test

```rust
#[test]
fn test_deposit_uses_stale_amount_per_share() {
    // Initialize user balances
    let balances = vec![
        (
            "user",
            vec![
                coin(20_000_000_000, "nami"),
                coin(20_000_000_000, "auto"),
                coin(20_000_000_000, "lqdy"),
            ],
        ),
        (
            "owner",
            vec![
                coin(100_000_000_000, "nami"),
                coin(100_000_000_000, "auto"),
                coin(100_000_000_000, "lqdy"),
                coin(100_000_000_000, "eth-usdc"),
            ],
        ),
    ];

    let allocations = vec![
        ("nami".to_string(), Uint128::from(1000u128)),
        ("auto".to_string(), Uint128::from(1000u128)),
        ("lqdy".to_string(), Uint128::from(1000u128)),
    ];

    // ---- Scenario A: without forced Run ----
    let mut test_env_a = index::setup(
        balances.clone(),
        "eth-usdc".to_string(),
        allocations.clone(),
        Some(Decimal::percent(50)),
        Some(Decimal::percent(0)),
    );

    let rcpt_denom_a = format!("x/nami-index-{}-rcpt", test_env_a.index.address);

    // First deposit
    test_env_a
        .index
        .execute_deposit(
            &mut test_env_a.app,
            "user",
            vec![
                coin(10_000u128, "nami"),
                coin(10_000u128, "auto"),
                coin(10_000u128, "lqdy"),
            ],
        )
        .unwrap();


    // move block 1 year
    const SECS_PER_YEAR: u64 = 31_557_600; // ≈365.25 days
    test_env_a.app.update_block(|block| {
        block.height += 1;
        block.time = block.time.plus_seconds(SECS_PER_YEAR);
    });

    let status_a = test_env_a.index.query_status(&mut test_env_a.app).unwrap();
    let shares_a = status_a.total_shares;

    // Check status before withdraw
    println!("");
    println!("----- SCENARIO A -----");
    println!("");
    println!("Check Status: first deposit, before withdraw, without manual Run");
    println!("Total shares: {}", shares_a);
    println!("Allocations:");
    for (denom, weight) in status_a.allocation.iter() {
        println!("- {}: {}", denom, weight);
    }

    // Withdraw
    test_env_a
        .index
        .execute_withdraw(
            &mut test_env_a.app,
            "user",
            vec![coin(10u128, rcpt_denom_a.clone())],
        )
        .unwrap();


    // Check status after withdraw
    let status_a = test_env_a.index.query_status(&mut test_env_a.app).unwrap();
    let shares_a = status_a.total_shares;
    println!("Check Status: first deposit, after withdraw, without manual Run");
    println!("Total shares: {}", shares_a);
    println!("Allocations:");
    for (denom, weight) in status_a.allocation.iter() {
        println!("- {}: {}", denom, weight);
    }

    // Second deposit
    test_env_a
        .index
        .execute_deposit(
            &mut test_env_a.app,
            "user",
            vec![
                coin(1_000_000u128, "nami"),
                coin(1_000_000u128, "auto"),
                coin(1_000_000u128, "lqdy"),
            ],
        )
        .unwrap();

    // Check status after second deposit
    let status_a = test_env_a.index.query_status(&mut test_env_a.app).unwrap();
    let shares_a = status_a.total_shares;
    println!("Check Status: second deposit, after withdraw, without manual Run");
    println!("Total shares: {}", shares_a);
    println!("Allocations:");
    for (denom, weight) in status_a.allocation.iter() {
        println!("- {}: {}", denom, weight);
    }

    // ---- Scenario B: with forced Run after withdraw ----
    let mut test_env_b = index::setup(
        balances,
        "eth-usdc".to_string(),
        allocations,
        Some(Decimal::percent(50)),
        Some(Decimal::percent(0)),
    );

    let rcpt_denom_b = format!("x/nami-index-{}-rcpt", test_env_b.index.address);

    // First deposit
    test_env_b
        .index
        .execute_deposit(
            &mut test_env_b.app,
            "user",
            vec![
                coin(10_000u128, "nami"),
                coin(10_000u128, "auto"),
                coin(10_000u128, "lqdy"),
            ],
        )
        .unwrap();

    // move block 1 year
    test_env_b.app.update_block(|block| {
        block.height += 1;
        block.time = block.time.plus_seconds(SECS_PER_YEAR);
    });

    // Check status before withdraw
    let status_b = test_env_b.index.query_status(&mut test_env_b.app).unwrap();
    let shares_b = status_b.total_shares;
    println!("");
    println!("----- SCENARIO B -----");
    println!("");
    println!("Check Status: first deposit, before withdraw, before manual Run");
    println!("Total shares: {}", shares_b);
    println!("Allocations:");
    for (denom, weight) in status_b.allocation.iter() {
        println!("- {}: {}", denom, weight);
    }

    // Withdraw
    test_env_b
        .index
        .execute_withdraw(
            &mut test_env_b.app,
            "user",
            vec![coin(10u128, rcpt_denom_b.clone())],
        )
        .unwrap();


    // Check status after withdraw
    let status_b = test_env_b.index.query_status(&mut test_env_b.app).unwrap();
    let shares_b = status_b.total_shares;
    println!("Check Status: first deposit, after withdraw, BEFORE manual Run");
    println!("Total shares: {}", shares_b);
    println!("Allocations:");
    for (denom, weight) in status_b.allocation.iter() {
        println!("- {}: {}", denom, weight);
    }

    // Call Run manually to update internal state before next deposit
    test_env_b
        .index
        .execute_run(&mut test_env_b.app, "user")
        .unwrap();


    // Check status after Run
    let status_b = test_env_b.index.query_status(&mut test_env_b.app).unwrap();
    let shares_b = status_b.total_shares;
    println!("Check Status: first deposit, after withdraw, AFTER manual Run");
    println!("Total shares: {}", shares_b);
    println!("Allocations:");
    for (denom, weight) in status_b.allocation.iter() {
        println!("- {}: {}", denom, weight);
    }

    // Second deposit
    test_env_b
        .index
        .execute_deposit(
            &mut test_env_b.app,
            "user",
            vec![
                coin(1_000_000u128, "nami"),
                coin(1_000_000u128, "auto"),
                coin(1_000_000u128, "lqdy"),
            ],
        )
        .unwrap();

    // Check status after second deposit
    let status_b = test_env_b.index.query_status(&mut test_env_b.app).unwrap();
    let shares_b = status_b.total_shares;
    println!("Check Status: second deposit, after withdraw, after manual Run");
    println!("Total shares: {}", shares_b);
    println!("Allocations:");
    for (denom, weight) in status_b.allocation.iter() {
        println!("- {}: {}", denom, weight);
    }

    // Compare results
    assert!(
        shares_a != shares_b,
        "Expected different share totals with and without Run"
    );
}
```

Result

The test succeeds in showing a measurable difference in total shares. Without the manual Run, the vault over-mints shares, proving the impact of stale share calculations and demonstrating the vulnerability.

BVSS

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

Recommendation

It is recommended to restructure the `Withdraw` logic so that the rebalance is performed only after the withdrawal has completed. This ensures that the internal `amount_per_share` values remain stable during the deposit calculation phase.

Remediation Comment

SOLVED: The **Nami team** has solved this issue by introducing an extra rebalance of the vault at the end of the withdraw transaction.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/34b790d59452e22e05f6e685c2c2af73c9e66991

8.5 UNIT MISMATCHES IN WITHDRAW SLIPPAGE FLOW

// MEDIUM

Description

The `withdraw` path presents two independent issues:

- End-of-flow slippage check uses mismatched units: `min_amount` is computed in USD while `send_quote` is in the quote denom in

```
ensure!(send_quote >= min_amount, ContractError::SlippageExceeded {})
```

allowing false passes or spurious `SlippageExceeded` depending on price drift;

- Per-asset slippage guard mixes units: in the `try_fold` that builds `swap_msg`, `min_return` is computed as

```
let min_return = (amt > alloc_max).then(|| amt.checked_sub(alloc_max).unwrap());
```

while `amt` represents the tranche in USD (from `remaining`) and `alloc_max` is derived from `max_cost` in quote; when quote ≠ 1 USD, the comparison/subtraction are inconsistent, so the guard can be too lax (no bound) or overly strict (reverts), and aggregate loss can exceed the user's intended USD slippage.

Code Location

Code of `withdraw` function from contracts/nami-index-nav/src/vault.rs file.

```
pub fn withdraw(
    &self,
    storage: &dyn Storage,
    value: Uint128,
    sender: Addr,
    slippage: Option<Decimal>,
) -> Result<Vec<CosmosMsg>, ContractError> {
    let (quote, others) = self.load_allocations(storage)?;
    let (quote_bal, quote_price, _) = quote.snapshot(&self.address, self.querier)?;
    let slip = slippage.unwrap_or_default();
    let min_amount = Decimal::from_ratio(value, Uint128::one())
        .checked_mul(slip)
        .unwrap_or_default()
        .to_uint_floor();

    let amount = Decimal::from_ratio(value, Uint128::one())
        .checked_div(quote_price)?
        .to_uint_floor();

    let send_quote = amount.min(quote_bal);
    let mut msgs = vec![BankMsg::Send {
        amount: coins(send_quote.u128(), &quote.denom),
        to_address: sender.to_string(),
    }
    .into()];
    let remaining = amount.checked_sub(send_quote)?;
    if remaining.is_zero() {
        ensure!(send_quote >= min_amount, ContractError::SlippageExceeded {});
        return Ok(msgs);
    }

    let total_weight = others.iter().map(|a| a.weight).sum::<Decimal>();
    let max_cost = value.checked_sub(min_amount)?;

    let (swap_msgs, _) = others.into_iter().try_fold(
        (Vec::new(), remaining),
        |(mut msgs, rem), alloc| -> Result<(Vec<CosmosMsg>, Uint128), ContractError> {
            let share = alloc.weight.checked_div(total_weight)?;
            let amt = Decimal::from_ratio(rem, Uint128::one())
                .checked_mul(share)?
                .to_uint_floor();
            if amt.is_zero() {
                return Ok((msgs, rem));
            }

            let alloc_max = Decimal::from_ratio(max_cost, Uint128::one())
                .checked_mul(share)?
                .to_uint_floor();
            let min_return = (amt > alloc_max).then(|| amt.checked_sub(alloc_max).unwrap());

            msgs.push(alloc.swap_msg(&self.address, amt, &sender, self.querier, min_return)?);

            let rem = rem.checked_sub(amt)?;
            Ok((msgs, rem))
        },
    )?;
    msgs.extend(swap_msgs);

    Ok(msgs)
}
```

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:
1. Align the end-of-flow check by expressing `min_amount` in the quote denom (same unit as `send_quote`) and add a unit test to detect USD/quote mismatches;
2. Make the per-asset slippage guard unit-consistent by using **quote** throughout: convert each asset's USD share to quote with the same price snapshot used elsewhere, then compute `min_return` in quote.

Remediation Comment

SOLVED: The `Nami team` has solved the project in commits:

- 3034c9440ed907715fa8151ac1a07e6def3e1941
- 13bcde4f64aa22a50abd9a16d662276154162b9c
- 6f9042395d88b0edd2b04370190d5a27ab0e2771

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/6f9042395d88b0edd2b04370190d5a27ab0e2771

8.6 CALLBACK CAN BE TRIGGERED BY UNAUTHORIZED EXTERNAL CONTRACTS

// LOW

Description

In the `nami-index-fixed` contract, the `ExecuteMsg::Callback` message can be executed by **any sender**, without verifying whether the caller is an authorized contract registered in the vault's allocation map. This lack of sender validation creates a security risk: untrusted or malicious contracts may send Callback messages to trigger sensitive internal operations such as `AfterReallocate`.

These internal operations are intended to be used exclusively by trusted swap contracts as part of the reallocation flow. Allowing arbitrary contracts to invoke them introduces the risk of unexpected state changes, misuse of vault funds, or denial of service conditions.

Code Location

Code of `execute` function from contracts/nami-index-fixed/src/contract.rs file.

```
ExecuteMsg::Callback(cb) => {
    let callback_type: CallbackType = cb.deserialize_callback()?;
    match callback_type {
        CallbackType::AfterReallocate {
            swap_to,
            min_return,
        } => {
            let coin = deps
                .querier
                .query_balance(&env.contract.address, &config.quote_denom)?;
            let msg = WasmMsg::Execute {
                contract_addr: swap_to.to_string(),
                msg: to_json_binary(&fin::ExecuteMsg::Swap(SwapRequest {
                    min_return,
                    to: None,
                    callback: None,
                }))?,
                funds: vec![coin],
            };
            let run_msg = WasmMsg::Execute {
                contract_addr: env.contract.address.to_string(),
                msg: to_json_binary(&ExecuteMsg::Run {})?,
                funds: vec![],
            };
            response = response.add_message(msg).add_message(run_msg);
        }
    }
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (3.4)

Recommendation

It is recommended to validate that `info.sender` matches a known swap contract listed in the `ALLOCATIONS` map before executing any logic under the Callback message. This ensures that only authorized and registered contracts involved in the reallocation process are allowed to invoke internal callbacks.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by verifying whether the `info.sender` address belongs to the set of allowed allocations.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/f485834af93f3789665630102ef44536bdacfa2a

8.7 REALLOCATION USES FULL QUOTE_DENOM BALANCE WITHOUT ISOLATING SOURCE

// LOW

Description

In the `nami-index-fixed` contract, during a reallocation operation, the vault performs two sequential swaps: the first converts the source asset (from) into `quote_denom`, and the second swaps `quote_denom` into the target asset (to). However, the second swap uses the entire `quote_denom` balance held by the vault, without distinguishing whether those funds originated from the first swap or were already present in the vault.

This design assumes that all `quote_denom` available at that moment comes from the initial swap. But if the vault holds pre-existing `quote_denom` due to dust, prior rounding, user donations, or unrelated transfers, then the second swap may consume more than intended. This can lead to:

- Inaccurate `amount_per_share` calculations after rebalance.
- Systemic misalignment affecting all users and share value accuracy.

The lack of isolation between **intended swap input** and **residual balances** introduces uncertainty and undermines the deterministic behavior of the reallocation logic.

Code Location

Code of `Callback::AfterReallocate` logic from contracts/nami-index-fixed/src/contract.rs file.

```
ExecuteMsg::Callback(cb) => {
    let callback_type: CallbackType = cb.deserialize_callback()?;
    match callback_type {
        CallbackType::AfterReallocate {
            swap_to,
            min_return,
        } => {
            let coin = deps
                .querier
                .query_balance(&env.contract.address, &config.quote_denom)?;
            let msg = WasmMsg::Execute {
                contract_addr: swap_to.to_string(),
                msg: to_json_binary(&fin::ExecuteMsg::Swap(SwapRequest {
                    min_return,
                    to: None,
                    callback: None,
                }))?,
                funds: vec![coin],
            };
            let run_msg = WasmMsg::Execute {
                contract_addr: env.contract.address.to_string(),
                msg: to_json_binary(&ExecuteMsg::Run {})?,
                funds: vec![],
            };
            response = response.add_message(msg).add_message(run_msg);
        }
    }
}
```

BVSS

AO:A/AC:L/AX:M/R:N/S:U/C:N/A:N/I:M/D:N/Y:N (3.4)

Recommendation

It is recommended to isolate and track the exact amount of `quote_denom` received from the first swap, and use only that amount in the second swap step. Alternatively, before initiating the reallocation process, the contract could check for any existing `quote_denom` balance and abort the operation if such funds are present.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by querying the amount swapped during the first operation and passing that exact value to the second swap, instead of using the vault's entire balance.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/44d71d08314c5cb0963756ccce80e2d65dd971d7

8.8 MISSING VALIDATION THAT MAP KEY MATCHES BASE DENOM IN MULTIPLE CONTRACTS

// LOW

Description

In the Nami protocol, several functions across different contracts allow the registration of asset-related configurations using a `denom` string as the `map` `key`. These include:

- `add_contract` in nami-index-entry-adapter
- `add_allocation` in nami-index-fixed
- `save_allocation` in nami-index-nav

None of these functions verify whether the key being used (the `denom`) actually matches the `base_denom` declared in the configuration of the associated swap contract.

An important nuance is that swap contracts may define the `quote_denom` in either the `base()` or `quote()` field of their internal configuration, depending on the direction of the swap. Therefore, the validation logic must ensure that the registered denom matches the **opposite field** (i.e., the one that is not equal to the current `quote_denom`).

This is not just a consistency problem, it also has direct functional consequences. For example, if a mismatched `denom` is used, subsequent operations that rely on `query_balance` (such as calculating deposit proportions or rebalancing asset weights) will query for a token that is not actually associated with the contract, likely resulting in zero balances or execution errors.

Code Location

Code of `add_contract` function from contracts/nami-index-entry-adapter/src/state.rs file.

Copy Code

```rust
pub fn add_contract(
    &self,
    storage: &mut dyn Storage,
    querier: &QuerierWrapper,
    key: &'a str,
    value: String,
    quote_denom: String,
) -> Result<(), ContractError> {
    let config: fin::ConfigResponse =
        querier.query_wasm_smart(&value, &fin::QueryMsg::Config {})?;
    ensure_eq!(
        config.denoms.quote(),
        quote_denom,
        ContractError::InvalidQuoteDenom
    );
    self.swap_contracts.save(storage, key, &value)?;
    Ok(())
}
```

Code of `add_allocation` function from contracts/nami-index-fixed/src/vault.rs file.

Copy Code

```rust
pub fn add_allocation(
    &self,
    storage: &mut dyn Storage,
    quote_denom: &str,
    denom: &str,
    weight: Uint128,
    contract: &str,
) -> Result<(), ContractError> {
    let contract = self.api.addr_validate(contract)?;
    let config: fin::ConfigResponse = self
        .querier
        .query_wasm_smart(&contract, &fin::QueryMsg::Config {})?;
    ensure!(
        config.denoms.quote() == quote_denom || config.denoms.base() == quote_denom,
        ContractError::InvalidQuoteDenom
    );
    ALLOCATIONS.save(storage, denom, &(weight, contract))?;
    Ok(())
}
```

Code of `save_allocation` function from contracts/nami-index-nav/src/vault.rs file.

Copy Code

```rust
pub fn save_allocation(
    &self,
    storage: &mut dyn Storage,
    allocation: AssetAllocation<OracleConfig>,
) -> Result<(), ContractError> {
    ensure!(
        allocation.swap_contract.is_some() || allocation.denom == self.quote_denom,
        ContractError::InvalidSwapContract
    );
    if let Some(ref swap_addr) = allocation.swap_contract {
        self.api.addr_validate(swap_addr)?;
        let cfg: fin::ConfigResponse = self
            .querier
            .query_wasm_smart(swap_addr, &fin::QueryMsg::Config {})?;
        ensure!(
            cfg.denoms.quote() == self.quote_denom || cfg.denoms.base() == self.quote_denom,
            ContractError::InvalidQuoteDenom
        );
        allocation.oracle.price(*self.querier)?;
    }
    ensure!(
        allocation.slippage.gt(&Decimal::zero()) && allocation.slippage.lt(&Decimal::one()),
        ContractError::SlippageOne
    );
    ALLOCATIONS.save(storage, &allocation.denom, &allocation)?;
    Ok(())
}
```

BVSS

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

Recommendation

It is recommended to retrieve the `base_denom` value directly from the swap contract's configuration at the time of registration and validate that it matches the key used in the map. If they differ, the contract should reject the operation to ensure consistent and predictable routing logic.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by validating both the `denom` and the `quote` before storing the swapping contract.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/bec2b7822b41b70ddcb725cb96eb2e5a8852ed00

8.9 ALLOCATION ENTRIES CAN BE SILENTLY OVERWRITTEN

// LOW

Description

In the `nami-index-fixed` contract, the `add_allocation` function uses ALLOCATIONS.save(...) to insert a new asset allocation entry. However, it does not check if an entry with the same denom already exists in the map. As a result, calling `add_allocation` with a duplicate denom silently overwrites the previous configuration.

This behavior can lead to accidental or unauthorized modification of index compositions without any indication or rejection. Overwriting existing entries may also interfere with asset tracking, weight management, or rebalancing logic, potentially resulting in misconfigured indices or inconsistent user experience.

Code Location

Code of `add_allocation` function from contracts/nami-index-fixed/src/vault.rs file.

Copy Code

```rust
pub fn add_allocation(
    &self,
    storage: &mut dyn Storage,
    quote_denom: &str,
    denom: &str,
    weight: Uint128,
    contract: &str,
) -> Result<(), ContractError> {
    let contract = self.api.addr_validate(contract)?;
    let config: fin::ConfigResponse = self
        .querier
        .query_wasm_smart(&contract, &fin::QueryMsg::Config {})?;
    ensure!(
        config.denoms.quote() == quote_denom || config.denoms.base() == quote_denom,
        ContractError::InvalidQuoteDenom
    );
    ALLOCATIONS.save(storage, denom, &(weight, contract))?;
    Ok(())
}
```

BVSS

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

Recommendation

It is recommended to call `ALLOCATIONS.has(denom)` before saving any new allocation. If the denom already exists in the map, the contract should reject the operation to prevent unintended overwrites and preserve index integrity.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by adding an `ALLOCATIONS.has()` check before inserting a new allocation.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/5be715ddb8c3ae7aae6466bb29d70e788029bc4f

8.10 REUSE OF MIN_RETURN ACROSS INDEPENDENT SWAPS IN REALLOCATION

// LOW

Description

In the `nami-index-fixed` contract, the `Vault::reallocate` function performs a two-step swap process: first, it converts the source asset (from) into quote_denom, and then swaps quote_denom into the target asset (to). However, the same `min_return` value is applied across both steps.

This creates a problem because each swap has its own pricing, liquidity, and slippage characteristics. If the first swap fails to meet the threshold, the operation reverts, even if the full two-step swap would have produced an acceptable result overall.

Code Location

Code of `reallocate` function from contracts/nami-index-fixed/src/vault.rs file.

Copy Code

```rust
pub fn reallocate(
    storage: &mut dyn Storage,
    from: &str,
    to: &str,
    weight: Uint128,
    total_shares: Uint128,
    min_return: Option<Uint128>,
) -> Result<CosmosMsg, ContractError> {
    let (curr_weight, swap_from) = ALLOCATIONS.load(storage, from)?;
    let (_, swap_to) = ALLOCATIONS.load(storage, to)?;

    ensure!(curr_weight.gt(&weight), ContractError::InvalidWeight);

    let amount_to_swap = curr_weight.checked_sub(weight)?.checked_mul(total_shares)?;

    let callback = to_json_binary(&CallbackType::AfterReallocate {
        swap_to,
        min_return,
    })?;

    Ok(WasmMsg::Execute {
        contract_addr: swap_from.to_string(),
        msg: to_json_binary(&fin::ExecuteMsg::Swap(SwapRequest {
            min_return,
            to: None,
            callback: Some(callback.into()),
        }))?,
        funds: vec![Coin::new(amount_to_swap, from)],
    }
    .into())
}
```

BVSS

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

Recommendation

It is recommended to use separate `min_return` values for each swap step—one for `from` → `quote_denom` and another for `quote_denom` → `to`. Alternatively, compute the final output after both swaps and compare it against a single `min_return` threshold that reflects the total expected result.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by removing the `min_amount` constraint from the first swap.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/26c34139267e5e188dd0dede3a4ad70697e4f537

8.11 LACK OF AUTOMATIC REBALANCE ENFORCEMENT MAY LEAD TO STALE OR UNBALANCED PORTFOLIOS

// LOW

Description

In the `nami-index-nav` contract, the rebalancing logic is only triggered manually via the `ExecuteMsg::Run{}` message. There is no mechanism to automatically enforce rebalancing after relevant events, such as deposits, withdrawals, or allocation updates.

As a result, the portfolio may remain unbalanced for extended periods, especially if Run is not called regularly. This can cause the vault composition to drift away from the target weights, leading to inaccurate NAV calculations and possible arbitrage opportunities for advanced users.

Since the core purpose of the index is to maintain proportional exposure across assets, the absence of periodic or event-based rebalancing undermines the reliability and predictability of the system.

BVSS

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

Recommendation

It is recommended to enforce rebalancing under key conditions —such as after deposits or withdrawals— or to implement a time-based trigger (e.g., after a threshold duration has passed). This ensures that the index remains aligned with its target allocation and prevents long-term divergence in asset distribution.

Remediation Comment

RISK ACCEPTED: The **Nami team** accepted the risk associated with this finding and provided the following statement:

> We acknowledge the risk of long-term drift without automatic rebalancing. However, triggering `Run{}` on every deposit or withdrawal could block users during low-liquidity scenarios where `Run{}` might fail.
> To minimize friction, we've decided to keep `Run{}` as a permissioned but publicly callable entry point and rely on off-chain keepers to execute it on a regular schedule. This approach allows the vault to continue accepting deposits and withdrawals without interruption while still gradually converging toward the target allocations.

8.12 WEIGHT SUM CANNOT BE VERIFIED WITH INDIVIDUAL ALLOCATION UPDATES

// LOW

Description

In the `nami-index-nav` contract, the sum of allocation weights is enforced to equal exactly 1.0, but **only during the initial contract setup**. Subsequent updates—such as adding or removing allocations—are performed one at a time via functions like `save_allocation` and `remove_allocation`, **without revalidating the total weight.**

This makes it impossible to enforce the global invariant that allocation weights must always sum to 1. For example, removing one allocation and adding another in separate transactions may temporarily—or permanently—leave the total below or above 1.0. This breaks the core assumptions behind NAV calculations and rebalancing logic, leading to inaccurate share pricing, misaligned portfolio proportions, and potential economic inconsistencies.

BVSS

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

Recommendation

It is recommended to refactor the allocation update process to support batched operations, such as submitting a full set of new allocations and removals in a single transaction. This would allow verifying that the total weight remains exactly 1.0 at the end of the update.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by refactoring the allocation management logic, enabling the entire batch of allocations to be added or removed through a single function, and verifying that the sum of all weights is equal to 1.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/dd425669ef02633c67742b114964e741d8d9a7e0

8.13 LACK OF DOCUMENTATION ACROSS THE CODEBASE

// INFORMATIONAL

Description

The Nami protocol codebase lacks sufficient inline comments and Rust documentation for public functions, modules, and critical logic. Although the contracts implement complex mechanisms correctly, such as fee computation, NAV calculation, rebalancing, and cross-contract swap orchestration, there is little to no documentation explaining how these parts work or why they are designed this way.

This absence of documentation significantly affects readability, auditability, and long-term maintainability. It makes it harder for external contributors, auditors, and even future maintainers to understand the intended behavior, invariants, and design rationale behind the implementation. It also increases the risk of misinterpretation, regressions, or incorrect modifications during future upgrades.

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 Rust documentation comments (///) to all public functions, data structures, and modules. Additionally, inline comments should be used to clarify complex or non-obvious logic, particularly in sections involving fee logic, allocation updates, swap flows, and state transitions. A documented codebase will improve comprehension, reduce onboarding time, and support safer upgrades and audits.

Remediation Comment

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

8.14 INEFFICIENT SWAP STRATEGY IN DEPOSIT FLOW

// INFORMATIONAL

Description

In the `nami-index-entry-adapter` contract, the deposit logic performs a series of swaps from quote_denom to the target allocation tokens without first simulating how much of each is actually needed to meet the index weights. As a result, it may swap **more tokens than necessary**, leaving excess amounts unallocated.

These unused tokens are then typically swapped **back** to `quote_denom`, adding unnecessary complexity and gas costs to the transaction. This inefficiency becomes more pronounced when the number of target assets is high or when the deposit is unbalanced.

The lack of pre-swap simulation undermines the precision of the deposit flow, wastes resources, and may introduce slippage or rounding errors in the reverse swaps.

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 simulate each swap beforehand using the `Simulate` `query` interface provided by the target swap (FIN) contracts. This allows determining the exact amount needed per token to match the index proportions. Only the required swaps should then be executed, reducing overhead and avoiding the need to reverse unused tokens.

Remediation Comment

ACKNOWLEDGED: The **Nami team** has acknowledged this finding and provided the following statement:

> We acknowledge the inefficiency risk. However, performing on-chain pre-swap simulations would require the contract to fetch and interpret prices for every asset and execute multiple dry-run loops, introducing higher gas costs than simply reversing excess swaps.
> Instead, we'll rely on the front end to calculate the exact deposit proportions off-chain and pass those values directly to the contract.

8.15 BURN + MINT FEE SHARES COULD BE SIMPLIFIED VIA TRANSFER

// INFORMATIONAL

Description

In the current withdrawal logic of both `nami-index-fixed` and `nami-index-nav` contracts, the contract burns the full number of shares sent by the user (e.g., 100) and then **mints** the fee portion (e.g., 2 shares) to the `fee_collector`. Although functionally correct, this implementation introduces unnecessary complexity.

The minting of fee shares gives the impression that new value is being created, when in reality it is simply a redirection of part of the user's existing shares. This may confuse external observers, complicate accounting, and increase the risk of errors during audits or integrations.

Moreover, using mint unnecessarily increases the cognitive overhead of the fee system. The same result can be achieved more clearly and efficiently by burning only the `net` shares (e.g., 98) and transferring the remaining `fee` portion (e.g., 2) directly to the `fee_collector`.

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 simplify the fee application logic by using a transfer to move the `fee` `portion` of shares to the `fee_collector`, and then burning only the remaining user shares. This approach better reflects the fact that the fee is not newly created value but a reassignment of ownership, and improves the clarity of the system.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by transferring the fees instead of minting them.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/39fb050441c2cb8ed6fcff011200525dbef8dc10

8.16 RECEIPT TOKEN DENOMS ARE IDENTICAL ACROSS DIFFERENT INDEX TYPES

// INFORMATIONAL

Description

Both the `nami-index-fixed` and `nami-index-nav` contracts generate receipt tokens with denoms following the pattern `nami-index-{contract-address}-rcpt`. While the contract address ensures uniqueness on-chain, the format **does not indicate the index type** (fixed vs. nav), making it difficult for users, integrators, and block explorers to distinguish between them.

This ambiguity can lead to confusion in interfaces that rely on denom patterns, errors in data aggregation or reporting tools, and difficulties in tracing the origin or behavior of receipt tokens. It also complicates off-chain integrations where knowledge of the index logic (e.g., fixed-weighted vs. nav-based) is relevant for display or accounting purposes.

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 include the index type in the receipt token denom, for example, using `nami-fixed-{address}-rcpt` and `nami-nav-{address}-rcpt`. This would improve clarity, reduce integration risks, and make receipt tokens more self-descriptive in logs, UIs, and analysis tools.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by including the index type in the `denom` of the receipt token.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/e4a872ba5f8e30f84a7d5829fc947d409a9e35f9

8.17 INCONSISTENT SLIPPAGE SEMANTICS ACROSS WITHDRAW AND REBALANCE FUNCTIONS

// INFORMATIONAL

Description

In the `nami-index-nav` contract, the `slippage` parameter is interpreted inconsistently across different functions. In the `withdraw` flow, `slippage` is used **as a percentage of the expected return**, meaning that a value of 0.97 implies that the user expects to receive at least 97% of the calculated amount.

In contrast, the `rebalance_msg` function interprets `slippage` as a **tolerated loss**, subtracting it from 1 to compute the minimum return —e.g., a value of 0.03 means that 3% slippage is allowed, and the user should receive at least 97%.

This discrepancy in how slippage is handled across the codebase can confuse developers and integrators, lead to misconfigured transactions.

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 standardize the `slippage` semantics across all functions by adopting a unified interpretation. A consistent model, such as treating slippage as the maximum tolerated loss (i.e., `min_return = value * (1 - slippage)`), should be applied to both withdraw and rebalance flows to ensure predictable and reliable behaviour.

Remediation Comment

SOLVED: The **Nami team** resolved this issue by modifying the slippage calculation in the withdraw function to use `1 - slippage`.

Remediation Hash

https://github.com/NAMIProtocol/nami-contracts/commit/4caa0b4bc26fcb3350bff74a64ffd826edfb341b

---

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.