



SECURITY AUDIT

Project Nami

Presale

FINAL REPORT

Aug 2026

BAILSEC.IO



Disclaimer:

BailSec was engaged by the client to perform a time-boxed technical security assessment of the target identified in this report. The assessment applies only to the scope, version, commit, deployment, materials and information identified in the report. Its findings are point-in-time and not exhaustive. Changes made after the assessment—including remediation changes—were not assessed unless expressly stated otherwise and may affect the applicability of the report's conclusions.

No security assessment can identify every vulnerability or guarantee that a system is secure, error-free or protected against attack or loss. Only the components, assumptions, threat models and risk categories expressly identified as in scope were assessed. The report should be read in full, including its scope, exclusions, unresolved findings and remediation status. The client remains responsible for design, testing, remediation, deployment, operation, monitoring and incident response.

This report is a technical assessment, not a certification, attestation, investment recommendation, or legal, financial, regulatory or tax advice. It does not constitute an endorsement of the client, the assessed system or any related project.

This report was prepared for the client under a separate engagement agreement. If published, its public availability is for transparency and informational purposes only. To the fullest extent permitted by applicable law, publication does not create an auditor-client relationship, duty of care or right of reliance in favor of any third party. Other readers should conduct their own diligence and obtain appropriate professional advice.

Only the complete, unmodified report as issued by BailSec is authoritative. The signed engagement agreement exclusively governs the contractual rights and obligations between BailSec and the client, including warranties, remedies, liability limitations, indemnities and dispute resolution. Nothing in this notice excludes liability that cannot lawfully be excluded.

1. Project Details

Important: Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	Project Nami – Presale – Audit Report
Website	nami.fi
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/Project-Nami/Nami-Contracts/tree/bd8aa442bbf55542920763af965756e037d2cc39
Resolution 1	https://github.com/namifi/Private-Nami-Contracts/tree/6c82ec44a87ae9790b563f6fca8913fb18f92358

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution	Open
High	2	2				
Medium						
Low	4	3		1		
Informational	10	8		2		
Governance	3	2		1		
Total	19	15		4		

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior



3. Detection

Presale Module

Resolution:

<https://github.com/Project-Nami/Nami-Contracts/tree/b0912332b06b51ab68f2ebd4e0ebb5188a6fe7c9>

Forked:

github.com/bailsec/Nami-Contracts/tree/b0912332b06b51ab68f2ebd4e0ebb5188a6fe7c9

Initial commit:

[http://github.com/bailsec/Nami-Contracts/tree/bd8aa442bbf55542920763af965756e037d2cc39](https://github.com/bailsec/Nami-Contracts/tree/bd8aa442bbf55542920763af965756e037d2cc39)



PresaleManager

PresaleManager is the coordinator contract for the presale system, managing presale lifecycle, user vesting vaults, and incentive token accounting. The contract works in tandem with the Presale contract to handle post-sale vesting initialization, track all deployed UserVestingVault instances, and coordinate burning of unclaimed incentive tokens. It references the PermissionsRegistry for multisig access control.

The contract operates in two phases: during the presale when it registers new user vaults created by the Presale contract via registerVault, and after finalization when initializeVesting is called to start the vesting schedule. The manager calculates buyer incentive pool size based on total sold versus allocation, transferring unused incentives to the treasury. User vaults call back to burnIncentive when users forfeit incentives.

Privileged Functions

- setPresale
- initializeVesting

Core Invariants:

INV 1: Only the multisig can call **setPresale** and **initializeVesting**

INV 2: Only registered vaults can call **burnIncentive**

INV 3: Presale can only be set once

INV 4: Each user can only have one vault registered

INV 5: Buyer incentive pool scales proportionally if sale doesn't reach allocation

Issue_01	<code>burnIncentive</code> OOG prevents withdrawals
Severity	High
Description	The <code>burnIncentive</code> function iterates through the entire <code>allVaults</code> array to verify that <code>msg.sender</code> is a valid vault. When approximately 4,700 vaults are created, all subsequent purchases will no longer be able to withdraw when their <code>toBurn</code> is greater than 0, since iterating through the initial 4.7k vaults will exceed block gas limits. When a user calls <code>withdraw</code> on their <code>UserVestingVault</code>, the function transfers unearned incentive tokens to the manager and calls <code>burnIncentive</code>. As the <code>allVaults</code> array grows beyond several thousand entries, the loop in <code>burnIncentive</code> will consume excessive gas, causing the entire withdrawal transaction to revert. This denial of service prevents users from accessing their vested tokens and earned incentives between the cliff period ending and the <code>INCENTIVE_DURATION</code> time being met. The issue affects all users attempting to withdraw once the vault count reaches the gas limit threshold, effectively locking funds in the vesting vaults.
Recommendations	Add a mapping of <code>address => bool</code> alongside the <code>allVaults</code> array. Set the mapping to <code>true</code> in <code>registerVault</code> and use it in <code>burnIncentive</code> for vault verification instead of iterating through the array.
Comments / Resolution	Fixed by following recommendation.

Issue_02	Vesting cannot be initialized when <code>totalSold < saleAllocation</code>
Severity	High
Description	Function <code>initializeVesting</code> can be called once the presale is finalized. This is done to ensure user withdrawals can start their vesting period. However the function attempts to transfer out unused incentive to the treasury when the contract holds no incentive tokens (presale finalization in <code>Presale.sol</code> sends the tokens to the treasury). <pre> if (unusedIncentive > 0) { IERC20(token).safeTransfer(treasury, unusedIncentive); emit UnusedIncentiveSent(unusedIncentive); } </pre> Due to this, function <code>initializeVesting</code> cannot be called when <code>totalSold < saleAllocation</code> is true as there are no incentive tokens to transfer out. This also prevents users from withdrawing their tokens from vesting vaults.
Recommendations	Consider removing the transfer.
Comments / Resolution	Fixed by following recommendation.

Issue_03	Consider removing redundant variable VESTING_DURATION
Severity	Informational
Description	The variable VESTING_DURATION is never utilized in the PresaleManager and UserVestingVault contracts. <pre>uint256 public constant VESTING_DURATION = 5 * EPOCH_PERIOD;</pre>
Recommendations	Consider removing the variable if it is not intended to be used.
Comments / Resolution	Fixed. All constants have now been moved to a new PresaleConstants library.

Issue_04	Consider improving _startFromPreviousEpoch variable name
Severity	Informational
Description	Function initializeVesting allows the multisig to pass the _startFromPreviousEpoch parameter value. However, when _startFromPreviousEpoch = true, it uses the start of the current epoch. <pre>vestingStart = _startFromPreviousEpoch ? currentEpoch : currentEpoch + EPOCH_PERIOD; vestingInitialized = true;</pre>
Recommendations	We consider the bug to only be a grammatical error in the variable name. It is recommended to rename it to _startFromCurrentEpochStart .
Comments / Resolution	Fixed by following recommendation.

Presale

Presale is the token sale contract managing the purchase phase, user vault creation, and payment collection for the Nami presale. The contract accepts payment token (e.g., USDC) in discrete units from users, deploys individual UserVestingVault instances per user via BeaconProxy, and transfers both purchased tokens and incentive tokens to those vaults. It enforces per-user purchase caps, target raise amounts, and pausability, while integrating with the PresaleManager for vault registration. Access control uses PermissionsRegistry for multisig operations.

The contract operates through the purchase function which validates sale timing, calculates purchase amounts in PURCHASE_UNIT increments, caps individual contributions at MAX_PER_USER, and stops at targetRaiseAmount. First-time purchasers trigger vault deployment, repeat purchasers have additional tokens added to their existing vault. Payment tokens transfer directly to the treasury while sale tokens transfer to vaults. The multisig controls sale timing via startSale and extendSale, finalization via finalizeSale, and emergency functions for pausing and withdrawing.

Privileged Functions

- startSale
- extendSale
- setPresaleManager
- finalizeSale
- pause
- emergencyWithdraw

Core Invariants:

INV 1: Purchases can only occur at or after saleStart, and before saleEnd timestamps

INV 2: Purchases are processed in PURCHASE_UNIT increments

INV 3: Users cannot exceed MAX_PER_USER total spend

INV 4: Total raised cannot exceed targetRaiseAmount

Issue_05	Auto-Reduced purchases will impact 3rd party contracts
Severity	Low
Description	The <code>purchase</code> function automatically reduces requested units when they exceed user spending cap or target raise, but lacks documentation explaining this behavior. When <code>newUserSpent</code> exceeds <code>MAX_PER_USER</code>, <code>paymentAmount</code> recalculates to <code>maxUserUnits</code> multiplied by <code>PURCHASE_UNIT</code>. Similarly, when <code>newTotalRaised</code> exceeds <code>targetRaiseAmount</code>, <code>paymentAmount</code> adjusts to <code>maxUnits</code> multiplied by <code>PURCHASE_UNIT</code>. Users calling <code>purchase</code> with a specific <code>units</code> parameter may expect that exact amount to process or the transaction to revert. The reduction without warning means users receive fewer tokens than requested without knowledge of this behavior. The lack of documentation or slippage control for the user makes the Presale contract unsafe for 3rd party contracts and integrators that may want to take part in the presale event.
Recommendations	Add documentation stating that the <code>purchase</code> function will automatically reduce the amount requested to fit within user cap and remaining target raise amount. Additionally add slippage or consider returning the actual purchase amount so that users can use this value for any subsequent logic that they will be executing.
Comments / Resolution	Acknowledged.

Issue_06	Missing validation blocks early presale finalization
Severity	Low
Description	The Presale contract allows a configuration where the <code>targetRaiseAmount</code> is not a multiple of the <code>PURCHASE_UNIT</code>. This makes it impossible to reach the target amount required to finalize the sale early. The <code>finalizeSale</code> function reverts with <code>SaleNotEnded</code> if called before the sale end time while <code>totalRaised < targetRaiseAmount</code>. If the target cannot be exactly matched due to unit size (e.g., target 100, unit 3 -> max raise 99), <code>totalRaised</code> will always remain strictly less than <code>targetRaiseAmount</code>. This permanently blocks the early finalization logic, forcing the protocol to wait for the full sale duration to expire before the sale can be finalized. Additionally <code>MAX_PER_USER</code> not being divisible by purchase units will prevent users from reaching the actual max purchase amount.
Recommendations	Add a validation check in the initialize function to ensure <code>targetRaiseAmount</code> and <code>MAX_PER_USER</code> are divisible by <code>PURCHASE_UNIT</code> .
Comments / Resolution	Fixed by following recommendation.



UserVestingVault

UserVestingVault is a per-user vesting contract deployed by the Presale to manage purchased token vesting and incentive distribution. The contract stakes purchased tokens immediately during initialization, releases them linearly over 5 epochs after a 1-epoch cliff, and distributes incentive tokens over 12 epochs with forfeited incentives burned. Each vault is uniquely tied to one user address and manages their entire presale allocation.

The contract operates through the withdraw function which users call to claim vested principal and earned incentives, with an autoStake option to immediately restake into ogNamiStaking.

Privileged Functions

- none

Core Invariants:

INV 1: Only the user can call withdraw and claimStakingRewards

INV 2: Withdrawals can only occur after the cliff duration has passed

INV 3: Forfeited incentives are sent to manager for burning on first withdrawal

INV 4: Purchased tokens are automatically staked during initialization

INV 5: addPurchase reverts if vesting has already started

Issue_07	<code>WrappedRewardTokenAdapter</code> reverts on dust withdrawals, causing DoS for <code>UserVestingVault</code> reward claiming
Severity	Low
Description	The <code>UserVestingVault</code> forces users to claim all rewards simultaneously via <code>claimStakingRewards()</code>, which iterates through every reward token. However, the <code>WrappedRewardTokenAdapter</code> strictly reverts if <code>convertToAssets(shares)</code> rounds down to zero. This creates a Denial of Service (DoS) where accumulating a dust amount of a wrapped reward token blocks the user from claiming any rewards. When a user fully withdraws their principal, their staking balance becomes zero. The rewards can only be claimed by the owner() or when anyone stakes more tokens to <code>OGNamiStaking</code> with the vault as the receiver. However, these newly staked tokens cannot be withdrawn.
Recommendations	Expose a function in <code>UserVestingVault</code> that allows users to claim rewards for specific tokens individually. This enables users to bypass the reverting dust token and rescue their other rewards.
Comments / Resolution	Fixed by following recommendation.

Issue_08	Misleading Vesting Documentation Can Lead Users to Incur Penalties for Zero Principal Gain
Severity	Informational
Description	The presale documentation states "Week 2: 25%" for principal vesting, implying users can withdraw 25% of their principal starting on Day 8. However, the code implementation in <code>UserVestingVault.withdraw</code> enforces that no principal vests until two full weeks (14 days) have elapsed. Users relying on this documentation may choose to withdraw on Day 8, accepting the documented "early withdrawal penalty" (burning unvested incentives) in exchange for accessing 25% of their principal. However, due to the discrepancy, these users will: - Incur the full penalty (permanently burning 90% of their incentives). - Receive 0% of their principal, instead of the expected 25%. This creates a scenario where users are misled into taking a significant financial penalty for no principal gain, an outcome they likely would not have chosen if the schedule were accurately documented.
Recommendations	Update the documentation to clarify that principal vesting begins only after the completion of the second week (Day 14+). Also consider adding a <code>getClaimable()</code> view function that returns the exact amounts of principal and incentives available for withdrawal, allowing users to verify their expected return before committing to the transaction.
Comments / Resolution	Acknowledged.



MultiRewardsDistributor

MultiRewardsDistributor is an abstract base contract implementing multi-reward distribution mechanics for staking systems. The contract enables protocols to distribute multiple different reward tokens to stakers over time-based periods, with automatic decimal normalization through wrapper tokens when needed. It's used by OGNamiStaking and IncentivizedVault contracts to provide reward distribution capabilities on top of their core staking functionality.

The contract operates on an epoch-based distribution model where reward tokens are streamed linearly over the PERIOD duration. Users interact through staking position tracking, with rewards accruing continuously based on their share of the total supply. The contract supports reward redirection and automated wrapping of non-18-decimal tokens.

Privileged Functions

- addReward
- notifyRewardAmount
- setDistributor
- setBatcherFactory
- recoverERC20

Core Invariants:

INV 1: Reward tokens with decimals other than 18 must be wrapped before distribution

INV 2: Reward rate cannot exceed available token balance divided by PERIOD

INV 3: Rewards are updated before any balance changes through the updateReward modifier

INV 4: Each reward token can only be added once

Issue_09	<code>recoverERC20</code> withdraws share-backing assets
Severity	Governance
Description	The <code>recoverERC20</code> function calculates recoverable asset tokens as the difference between <code>balance</code> and <code>totalSupply</code>. This logic works for <code>OGNamiStaking</code> where shares and assets maintain a 1:1 ratio. However, <code>IncentivizedVault</code> inherits <code>MultiRewardsDistributor</code> and implements ERC4626 vault mechanics where <code>totalSupply</code> represents shares, not assets. In vault implementations, ALL assets back the shares through the <code>convertToAssets</code> mechanism. The incorrect calculation allows the owner to withdraw assets that actually back user shares, rendering those shares partially or fully unredeemable. For example, if a vault holds 1000 assets backing 800 shares, the function calculates 200 tokens as recoverable when those 200 assets are still necessary to maintain the share price and enable withdrawals.
Recommendations	For <code>IncentivizedVault</code> , block the asset recovery case from <code>recoverERC20</code> , preventing withdrawal of any asset tokens.
Comments / Resolution	Fixed by following recommendation.

Issue_10	New stakers can steal past rewards if rewards are notified before token is added
Severity	Governance
Description	The <code>MultiRewardsDistributor.sol</code> contract allows for a "Ghost Token" state where rewards can be notified and accumulate for a token that has not yet been formally added to the contract's tracking system. This vulnerability arises from a logical disconnect between <code>notifyRewardAmount</code> and <code>addReward</code>. The <code>notifyRewardAmount</code> function permits any whitelisted distributor to initialize reward parameters (rate, periodFinish, lastUpdateTime) for any token, even if that token is not currently in the <code>rewardTokens</code> array. Because the token is missing from the <code>rewardTokens</code> array, the <code>updateReward</code> modifier (which relies on iterating this array) fails to take the necessary "debt snapshot" (<code>userRewardPerTokenPaid</code>) for users entering the pool during this period. The contract effectively "forgets" to mark new stakers as having zero claim on the pre existing rewards. When the token is eventually added via <code>addReward</code>, these users who have a default 0 debt snapshot are treated as if they had been staking since the beginning of the "Ghost Period," allowing them to claim rewards they did not earn Proof of Concept Scenario: Day 0 (The Mistake): A Distributor calls <code>notifyRewardAmount</code> for GhostToken before the Admin calls <code>addReward</code>. The "Ghost Period" begins. Day 0-2 (Accumulation): Correct behavior would dictate that only users staking during this time earn rewards. Day 2 (The Entry): User1 stakes 1,000 tokens. Crucially, because GhostToken is unlisted, the contract fails to set <code>userRewardPerTokenPaid[User1][GhostToken]</code>. It remains 0. Day 2 (The Fix): Admin notices the issue and calls <code>addReward[GhostToken]</code>. Day 2 (The Theft): User1 calls <code>getReward</code>. The contract calculates rewards based on the time delta from Day 0 (Last Update) to Day 2. Since User1's paid snapshot is 0, they receive full credit for Day 0-2. Result: User1 steals rewards for the 2 days they were not in the pool.

Recommendations	Enforce a strict check in <code>notifyRewardAmount()</code> to ensure the token has been formally added to the system before accepting any rewards. This ensures that <code>updateReward</code> will always effectively track the token before any meaningful reward data can be initialized, guaranteeing that all user debts are properly snapshotted.
Comments / Resolution	Fixed by following recommendation.

Issue_11	Centralization Risk in Reward Redirection
Severity	Governance
Description	In <code>MultiRewardsDistributor.sol</code>, <code>setRewardRedirect()</code> is used by <code>RewardOwner</code> to configure the reward for other accounts. This function allows the admin to set rewards for anyone. It poses a risk for heavy centralization which should be avoided in the contract as it can also be used maliciously by the admin/owner. <pre> function setRewardRedirect(address _from, address _to) external nonReentrant onlyRewardsOwner { // @audit Centralization risk as the Owner can redirect rewards to any address rewardRedirect[_from] = _to; emit RewardRedirected(_from, _to); } </pre>
Recommendations	Consider acknowledging this behaviour if intended.
Comments / Resolution	Acknowledged.

Issue_12	Zero-supply periods permanently lock reward tokens due to restrictive recovery logic
Severity	Low
Description	When the contract holds no staked tokens (<code>totalSupply() == 0</code>), the reward rate logic essentially pauses the accumulation of <code>rewardPerToken</code>, meaning no rewards are assigned to users during these empty windows. However, the contract logic creates a black hole for rewards released during these zero-supply periods. While the math correctly stops assigning rewards to users, the tokens allocated for that time window remain in the contract's balance. The current implementation of <code>recoverERC20</code> explicitly prevents the owner from sweeping any token listed in the <code>rewardTokens</code> array, regardless of the active state of the reward period. This restriction results in a permanent loss of funds. If all users withdraw their stake while a reward period is active, the rewards for the remaining duration are accrued by the contract but owned by no one. The owner cannot recover them due to the <code>recoverERC20</code> check, nor can they effectively roll them over into a new period, as <code>notifyRewardAmount</code> requires a non-zero supply to function and expects a fresh transfer of funds for the new rate. Consequently, these orphaned reward tokens remain permanently locked in the contract.
Recommendations	Consider acknowledging this issue as changes will likely result in side effects. Or modify <code>recoverERC20</code> to allow the recovery of reward tokens, but strictly enforce that this cannot occur for the unclaimed rewards.
Comments / Resolution	Fixed by following recommendation.

Issue_13	RewardPaid event emits wrong token and amount
Severity	Informational
Description	The transferReward function emits misleading information when rewarding tokens with less than 18 decimals. When a reward token has fewer than 18 decimals, the contract wraps it for internal accounting. During reward claims, the transferReward function withdraws the wrapped token by calling the adapter's withdraw function, which transfers the underlying unwrapped token to the recipient. However, the RewardPaid event emits the wrapped token address in the rewardsToken parameter and the wrapped amount in the reward parameter, despite the recipient actually receiving the underlying token. This creates incorrect event data for off-chain systems tracking reward distributions, as the token address and amount logged in the event do not match what was actually transferred to the user.
Recommendations	Get the actual underlying token address and unwrapped amount from the adapter before emitting the event, then emit the underlying token address and actual transferred amount in RewardPaid.
Comments / Resolution	Fixed by following recommendation.

Issue_14	Reward rate precision loss results in permanently locked dust
Severity	Informational
Description	The <code>notifyRewardAmount</code> function calculates the <code>rewardRate</code> by dividing the provided reward amount by the distribution duration (<code>PERIOD</code>). Since Solidity performs integer division, this operation rounds down, leaving a remainder of up to <code>PERIOD - 1</code> wei that is not included in the <code>rewardRate</code>. Because this remainder is excluded from the rate, it is never distributed to users. Furthermore, <code>notifyRewardAmount</code> does not include the contract's existing dust balance when calculating the new rate. It only considers the newly notified amount and the remaining valid rewards from the current period. Combined with the strict <code>recoverERC20</code> restriction that prevents withdrawing any reward tokens, this remainder accumulates in the contract with no mechanism to retrieve it.
Recommendations	Consider acknowledging this issue as changes will likely result in side effects. Or modify <code>recoverERC20</code> to allow the recovery of reward tokens, but strictly enforce that this cannot occur for the unclaimed rewards.
Comments / Resolution	Fixed by following recommendation.

Issue_15	Unbounded reward token list creates permanent DoS risk for all contract operations
Severity	Informational
Description	The <code>MultiRewardsDistributor</code> contract relies on the <code>_updateReward</code> modifier to calculate and checkpoint rewards for every token in the <code>rewardTokens</code> array whenever a user interacts with the contract (via <code>stake</code>, <code>withdraw</code>, or <code>getReward</code>). The <code>addReward</code> function allows the owner to append new tokens to this array indefinitely, but there is no corresponding functionality to remove or disable a reward token. This creates a critical Denial of Service (DoS) vector. If the owner adds a sufficient number of reward tokens, the gas cost to execute the <code>_updateReward</code> loop will exceed the block gas limit. Because this modifier is applied to all state-changing functions, including <code>withdraw</code> and <code>exit</code>, users would be unable to retrieve their staked assets. Since tokens cannot be removed, the contract would remain permanently bricked with user funds locked inside.
Recommendations	Impose a reasonable hard cap on the number of reward tokens that can be added (e.g., <code>MAX_REWARD_TOKENS = 20</code>) within the <code>addReward</code> function. Alternatively, add functionality to remove reward tokens. However, this would need to be implemented carefully.
Comments / Resolution	Fixed by following recommendation.

Issue_16	Missing Validation for <code>address[0]</code> in <code>getRewardTo()</code> can lead to loss of funds for User
Severity	Informational
Description	The <code>getRewardTo()</code> function in <code>MultiRewardsDistributor.sol</code> does not validate that the <code>_receiver</code> parameter is not <code>address[0]</code>. When called with <code>address[0]</code> as the receiver, this causes the user's rewards to be permanently lost or stuck in the staking contract when the wrapped token is being used. <code>getRewardTo()</code> accepts <code>address[0]</code> without validation. When <code>WrappedRewardTokenAdapter.withdraw()</code> receives <code>address[0]</code> as the receiver, it falls back to <code>msg.sender</code> (the staking contract). The user's reward balance is cleared and the staking contract gets the reward which is then not accessible to the user and can be recovered by admin via <code>recoverERC20()</code> function. <pre> function getRewardTo(address _account, address _receiver) external nonReentrant updateReward(_account) { if { msg.sender != _account && msg.sender != owner() && !_isAuthorizedBatcher(_account) } revert UnauthorizedClaimer(); _getRewardTo(_account, _receiver); // No validation for address[0] } function _getRewardTo(address _account, address _receiver) internal { _claimRewards(_account, _receiver, address[0], true); // Directly uses _receiver } function withdraw(uint256 _shares, address _receiver) external returns (uint256 assets) { if [_shares == 0] revert ZeroAmount(); if [_receiver == address[0]] _receiver = msg.sender; // Falls back to staking contract // ... </pre>

	<pre>IERC20(underlying).safeTransfer(_receiver, assets); }</pre> By calling <code>getRewardTo(user,address[0])</code>, the user can lose his reward tokens and they will be stuck in the staking contract and the user's reward balance would be cleared i.e <code>rewards[user][token] = 0</code>. User cannot reclaim rewards as their balance is already cleared. For wrapped token the tokens are not permanently lost rather the owner can recover them via <code>recoverERC20()</code> but tokens will go to the owner not the user.
Recommendations	It is recommended to put validation for <code>address[0]</code> so the user receives its reward in the proper manner.
Comments / Resolution	Fixed by following recommendation.

Issue_17	Missing function to remove reward tokens
Severity	Informational
Description	The contract <code>MultiRewardsDistributor.sol</code> has <code>addReward()</code> but no <code>removeReward()</code> . Once added, reward tokens remain permanently in the <code>rewardTokens</code> array. Functions like <code>_updateReward()</code> iterate over all reward tokens and gas cost increases linearly with the number of tokens. There seems no cleanup mechanism for deprecated tokens or tokens which might need removal.
Recommendations	It is recommended that <code>removeRewards()</code> function be added with proper safety checks.
Comments / Resolution	Fixed by following recommendation.

Issue_18	Reward dilution via frequent small updates
Severity	Informational
Description	The <code>OGNamiFeeConverter</code> and <code>OGNamiLPAutomation</code> contracts are designed to frequently harvest and distribute rewards to the staking contract. However, the <code>notifyRewardAmount</code> logic in <code>MultiRewardsDistributor</code> works by spreading the sum of the new amount plus all remaining rewards over a fresh full duration (<code>PERIOD</code>). When these distributor contracts are called frequently with small amounts (e.g., by a Keeper), they unintentionally trigger this re-calculation. Since the new rewards are small, the operation effectively takes the existing large pool of rewards (meant to be paid out over the next few days) and stretches it out over a new full period. This significantly reduces the <code>rewardRate</code> (APY) for current stakers and perpetually delays the payout of the main reward pool.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.



OGNamiStaking

OGNamiStaking is an ERC20 staking contract that accepts ogNami tokens and issues receipt tokens representing staked positions. The contract integrates **MultiRewardsDistributor** to enable multiple reward token distributions to stakers over time, with rewards accruing based on staking duration and share of total staked supply.

The contract operates through standard stake and unstake flows with optional receiver parameters to direct staked receipts or withdrawn tokens to different addresses. Users can claim rewards individually or exit entirely by withdrawing all staked tokens and claiming rewards in one transaction. Reward accounting updates automatically before any balance transfer.

Privileged Functions

- none

Core Invariants:

INV 1: Only the owner (multisig) can add reward tokens

INV 2: Only whitelisted distributors can notify reward amounts

INV 3: Reward accounting updates before any token transfer

INV 4: Unstaking burns receipt tokens and returns the staked asset

Issue_19	Misleading event emission
Severity	Informational
Description	The Staked event emits msg.sender as the account parameter and the actual share recipient as receiver. This parameter ordering is misleading because the account parameter suggests it represents the staking account holder, but msg.sender only provides the tokens while the receiver parameter holds the actual staking position, earns rewards, and can withdraw. Off-chain systems indexing this event will incorrectly attribute the staking position to msg.sender instead of the actual position holder.
Recommendations	The event should emit the share recipient as the account parameter since that address holds the balance and staking position.
Comments / Resolution	Fixed by following recommendation.