Custodia Security

Visit Our Website

Noya Review

Contents

Contents **2**
1. Disclaimer 3
2. Introduction 3
3. About Noya 3
4. Risk Classification 4
   4.1. Impact 4
   4.2. Likelihood 4
   4.3. Action required for severity levels 5
5. Security Assessment Summary 5
6. Executive Summary 5
7. Findings 8
   7.1. High Findings **8**
      [H-01] stakeWithPermit can be frontrun to force longer lock duration; lockMonths not part of signature 8
   7.2. MediumFindings **8**
      [M-01] Removing a reward token permanently forfeits unclaimed rewards; re-adding causes double distribution 8
   7.3. Low Findings **9**
      [L-01] Reentrancy during sNOYA _safeMint can burn receipt before position is stored, permanently locking stake 9
      [L-02] Fee-on-transfer and rebasing tokens cause epochRewards accounting mismatch and vault insolvency 9
      [L-03] Precision loss from dividing epochRewards by 4 before pro-rata multiplication 10
      [L-04] CommunityEmissionVault.approve unused code path 10
      [L-05] getCurrentWeekIdForEpoch underflow reverts when called before TGE, breaking all pre-TGE staking 11
      [L-06] CommunityEmissionVault.approve() uses non-safe transfers, fails for USDT and non-standard ERC20 tokens 11
      [L-07] StakerNoyaToken phantom approvals not blocked 12
      [L-08] IRewardsReader.getPendingRewards return type mismatch 12
      [L-09] Single failed reward token transfer blocks all claims for all users 13
      [L-10] updateMultiplier doesn't validate ascending order, allowing inverted or equal multipliers 13
      [L-11] Changing vault address orphans rewards 14

1. Disclaimer

A smart contract security review cannot ensure the absolute absence of vulnerabilities. This process is limited by time, resources, and expertise and aims to identify as many vulnerabilities as possible. We cannot guarantee complete security after the review, nor can we assure that the review will detect every issue in your smart contracts. We strongly recommend follow-up security reviews, bug bounty programs, and on-chain monitoring.

2. Introduction

Noya Staker Protocol is a comprehensive smart contract protocol that enables NOYA token holders to stake their tokens with variable lock durations (1, 3, 6, or 12 months) and earn proportional rewards from protocol revenue.

3. About Noya

NOYA represents a paradigm shift in decentralized finance, introducing a protocol that empowers AI agents to control liquidity across multiple chains with unparalleled trustlessness and precision. Engineered with a foundational composable system, NOYA built from the ground up a secure private keeper network, a trustless AI-compatible oracle, and a competitive environment for AI architects alongside strategy managers.

4. Risk Classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4.1. Impact

- High: Results in a substantial loss of assets within the protocol or significantly impacts a group of users.
- Medium: Causes a minor loss of funds (such as value leakage) or affects a core functionality of the protocol.
- Low: Leads to any unexpected behavior in some of the protocol's functionalities, but is not critical.

4.2. Likelihood

- High: The attack path is feasible with reasonable assumptions that replicate on-chain conditions, and the cost of the attack is relatively low compared to the potential funds that can be stolen or lost.
- Medium: The attack vector is conditionally incentivized but still relatively likely.
- Low: The attack requires too many or highly unlikely assumptions, or it demands a significant stake by the attacker with little or no incentive.

4.3. Action required for severity levels

- Critical: Must fix as soon as possible
- High: Must fix
- Medium: Should fix
- Low: Could fix

5. Security Assessment Summary

Duration: 27/02/2026 - 03/03/2026
Repository: https://github.com/noya-protocol/contracts
Commit: 1393cae3e3528bea2308ce62116e70a916aa8c34
Fix Commit: 0abecd27bae5fd33f376379bb4c22076ce166f87

Scope:

- src/common/*
- src/CommunityEmissionVault/*
- src/NoyaRewardEscrow/*
- src/NoyaStaker/*
- src/NoyaStaker/*
- src/NoyaToken/*
- src/RewardsCollector/*
- src/StakerNoyaToken/*
- src/StakerNoyaToken/*
- script/MainDeployment.s.sol

6. Executive Summary

Throughout the security review, our team engaged with Noya to review the new token staking contracts and the deployment script. In this period, a total of 13 issues were uncovered.

Findings Count

Severity	Amount
Critical	N/A
High	1
Medium	1
Low	11
Total Finding	**13**

Summary of Findings

ID	Title	Severity	Status
H-01	stakeWithPermit can be frontrun to force longer lock duration	High	Fixed
M-01	Removing reward tokens causes loss and double distribution	Medium	Fixed
L-01	Reentrancy during sNOYA _safeMint can burn receipt	Low	Fixed
L-02	Fee-on-transfer tokens cause vault insolvency	Low	Fixed
L-03	Precision loss from dividing epochRewards by 4	Low	Acknowledged
L-04	CommunityEmissionVault.approve unused code	Low	Fixed
L-05	getCurrentWeekIdForEpoch underflow before TGE	Low	Fixed
L-06	CommunityEmissionVault.approve non-safe transfers	Low	Fixed

L-07	StakerNoyaToken phantom approvals not blocked	Low	Fixed
L-08	IRewardsReader.getPendingRewards return type mismatch	Low	Fixed
L-09	Single failed reward transfer blocks all claims	Low	Fixed
L-10	updateMultiplier doesn't validate ascending order	Low	Fixed
L-11	Changing vault address orphans rewards	Low	Fixed

7. Findings

7.1. High Findings

[H-01] stakeWithPermit can be frontrun to force longer lock duration; lockMonths not part of signature

Severity: High

Description:
`stakeWithPermit()` does not require `msg.sender==owner` and does not bind lockMonths to any user-signed data. A mempool attacker can frontrun the victim's stakeWithPermit transaction by reusing the victim's EIP-2612 permit signature but choosing a larger lockMonths value (e.g., calling with 12 months instead of the victim's intended 1 month). The permit signature is valid regardless of lockMonths, so the victim's NOYA tokens end up locked much longer than intended. Meanwhile, the victim's original transaction reverts due to nonce consumption, forcing them to re-attempt staking and suffer unintended lock duration.

Recommendations:
Either require msg.sender==owner in stakeWithPermit(), or add an EIP-712 typed signature that includes the tuple (owner, amount, lockMonths, deadline) and validate the signature. If relayers are desired for gasless staking, lockMonths must be signed by the user so attackers cannot modify it.

7.2. MediumFindings

[M-01] Removing a reward token permanently forfeits unclaimed rewards; re-adding causes double distribution

Severity: Medium

Description:
RewardsCollector.collectRewards() records epochRewards += amount using the caller-supplied amount, then transfers amount to the vault via CommunityEmissionVault.transfer(). For fee-on-transfer tokens, the vault receives less than the recorded amount; for rebasing tokens, balances may change unexpectedly. Later, claimRewards() computes distributions based on the

inflated epochRewards and attempts to transfer from the vault. When the vault balance is exhausted (due to the fee mismatch), transfers revert and permanently block claims. Since claimRewards() processes multiple tokens in sequence, a single insolvent token can revert the entire claim transaction, locking all pending rewards.

Recommendations:
Record the actual received amount using a balance-before/balance-after pattern in collectRewards(). Alternatively, explicitly disallow fee-on-transfer and rebasing tokens via validation checks when adding reward tokens. Consider refactoring claimRewards() to be tolerant of single-token failures so one failing token doesn't block distribution of all other tokens.

7.3. Low Findings

[L-01] Reentrancy during sNOYA _safeMint can burn receipt before position is stored, permanently locking stake

Severity: Low

Description:
In NoyaStaker._stake(), sNOYA is minted via ERC721 _safeMint, invoking onERC721Received on contract recipients before positions[tokenId] is stored. Without nonReentrant on stake()/stakeForUser(), the recipient can re-enter withdraw(tokenId) during the callback, burning the newly minted NFT while the position is still uninitialized. _stake() then persists a position for a burned tokenId, leaving the transferred NOYA stuck because future withdraws fail on ownerOf(tokenId) check, permanently locking user funds.

Recommendations:
Add nonReentrant guard to stake() and stakeForUser(), and/or reorder to store position state before any external callback (avoid _safeMint window). Additionally, reinstate the pos.amount==0 existence guard in withdraw().

[L-02] Fee-on-transfer and rebasing tokens cause epochRewards accounting mismatch and vault insolvency

Severity: Low

Description:
RewardsCollector.collectRewards() records epochRewards += amount using the caller-supplied amount, then transfers amount to the vault via CommunityEmissionVault.transfer(). For

fee-on-transfer tokens, the vault receives less than the recorded amount; for rebasing tokens, balances may change unexpectedly. Later, claimRewards() computes distributions based on the inflated epochRewards and attempts to transfer from the vault. When the vault balance is exhausted (due to the fee mismatch), transfers revert and permanently block claims. Since claimRewards() processes multiple tokens in sequence, a single insolvent token can revert the entire claim transaction, locking all pending rewards.

Recommendations:
Record the actual received amount using a balance-before/balance-after pattern in collectRewards(). Alternatively, explicitly disallow fee-on-transfer and rebasing tokens via validation checks when adding reward tokens. Consider refactoring claimRewards() to be tolerant of single-token failures so one failing token doesn't block distribution of all other tokens.

[L-03] Precision loss from dividing epochRewards by 4 before pro-rata multiplication

Severity: Low

Description:
In claimRewards() and getPendingRewards(), reward distributions are computed as mulDiv(position.weight, epochRewards / 4, denominator). By dividing epochRewards by 4 before multiplication, the function truncates the remainder (epochRewards % 4), creating unrecoverable dust that accumulates in the vault. Additionally, each position's pro-rata share is rounded independently, creating further per-position rounding dust. Over time, these dust amounts compound, slowly draining the vault of undistributable rewards. This is exacerbated by the absence of an emergency withdrawal or vault sweep mechanism.

Recommendations:
Refactor to divide last: mulDiv(position.weight, epochRewards, 4 * denominator) (ensuring no overflow) to eliminate the first layer of precision loss. Consider implementing a vault sweep or dust recovery mechanism for accumulated rounding artifacts.

[L-04] CommunityEmissionVault.approve unused code path

Severity: Low

Description:
CommunityEmissionVault.approve() is an external function that allows setting arbitrary approvals on external tokens. This function is not called by any contract in the system and serves no operational purpose within the protocol. Unused code increases attack surface and

maintenance burden. Additionally, improper use of this function by external callers or governance could inadvertently approve tokens to unintended spenders.

Recommendations:
Remove the approve() function and its associated state, or clearly document its purpose if it is intended for external use. If governance needs to manage approvals, consider implementing a more restricted version that enforces explicit token whitelist checks.

[L-05] getCurrentWeekIdForEpoch underflow reverts when called before TGE, breaking all pre-TGE staking

Severity: Low

Description:
EpochCalculator.getCurrentWeekIdForEpoch() does not check if the current block timestamp is before the Token Generation Event (TGE) timestamp, unlike getCurrentEpoch(). When called before TGE, the computation (now - TGE) / weekDuration underflows since now < TGE, causing an integer underflow exception. This prevents any staking before TGE and blocks reward calculations that depend on this function, rendering the protocol non-functional during the pre-TGE phase and affecting all downstream logic.

Recommendations:
Add an explicit pre-TGE guard matching getCurrentEpoch() behavior:
if (block.timestamp < TGE) { return 0; }
Ensure both functions have consistent handling of pre-TGE timestamps and add tests covering staking attempts before TGE.

[L-06] CommunityEmissionVault.approve() uses non-safe transfers, fails for USDT and non-standard ERC20 tokens

Severity: Low

Description:
CommunityEmissionVault.approve() calls IERC20(_token).approve(_spender, _value) directly without using SafeERC20, which fails for non-standard tokens like USDT that do not return a boolean from approve(). Additionally, USDT requires allowance to be reset to zero before changing to a non-zero value — this function does not follow that pattern. If approve() returns false (instead of reverting), the false value is silently propagated without revert, potentially

causing undetected failed approvals. This blocks the vault from working with popular non-standard tokens.

Recommendations:
Replace raw IERC20(_token).approve() with IERC20(_token).forceApprove() from OpenZeppelin's SafeERC20 library, which handles non-standard return values and USDT's zero-reset requirement. Update the function signature appropriately.

[L-07] StakerNoyaToken phantom approvals not blocked

Severity: Low

Description:
StakerNoyaToken does not override the ERC721 approve() and setApprovalForAll() functions. This allows users to approve other accounts to transfer their receipt tokens (sNOYA), even though transferFrom() is disabled. The approval state is maintained but has no functional effect, creating phantom approvals that confuse users and external tooling about actual capabilities. Additionally, the missing override creates a false sense of control over who can access positions.

Recommendations:
Override approve() and setApprovalForAll() to either: (1) Revert unconditionally to clarify that approvals are not supported; (2) Emit a custom event for transparency; or (3) If future delegation is planned, implement a whitelist of allowed delegates with clear governance controls. Make the non-transferable nature explicit in the contract.

[L-08] IRewardsReader.getPendingRewards return type mismatch

Severity: Low

Description:
IRewardsReader.getPendingRewards() declares a return type of PendingRewards[] (array of structs), but the interface method and actual implementation details suggest per-position return values should be keyed or organized differently. There is a potential mismatch between the declared interface and the actual data structure returned, making it difficult for external consumers to integrate with the reward reader and potentially causing off-chain tooling failures.

Recommendations:
Review and clarify the intended return format for getPendingRewards(). Document the structure clearly in comments and ensure both the interface declaration and implementation align.

Consider returning a map or named struct with clear labeling of which rewards correspond to which positions/epochs.

[L-09] Single failed reward token transfer blocks all claims for all users

Severity: Low

Description:
claimRewards() iterates through multiple reward tokens and transfers each one's claimable amount from CommunityEmissionVault. If a single transfer fails (due to token blacklist, allowance issues, external revert, or the vault being insolvent for that specific token), the entire transaction reverts. This prevents users from claiming ANY of their pending rewards, not just the failed token. A single problematic reward token blocks all reward claims, creating a denial-of-service scenario.

Recommendations:
Implement a try-catch pattern for each token transfer, allowing the function to continue claiming other tokens even if one fails. Alternatively, provide a separate per-token or paginated claim mechanism so users can claim subsets of rewards. Log failed transfers for monitoring and allow users to retry or bypass the failed token.

[L-10] updateMultiplier doesn't validate ascending order, allowing inverted or equal multipliers

Severity: Low

Description:
The updateMultiplier() function allows updating lock duration multipliers (1m, 3m, 6m, 12m) without validating that they remain strictly in ascending order. If multipliers are updated such that shorter locks receive equal or higher multipliers than longer locks, the reward distribution logic is compromised. For example, a 3-month position could receive more rewards than a 12-month position despite locking tokens for 1/4 the duration. This violates the protocol's incentive design and can be exploited to maximize rewards with minimal lock commitment.

Recommendations:
Add validation after each multiplier update to ensure strict ascending order: multiplier[1m] < multiplier[3m] < multiplier[6m] < multiplier[12m]. Optionally add bounds checks relative to previous values to catch accidental misconfigurations.

[L-11] Changing vault address orphans rewards

Severity: Low

Description:
RewardsCollector.setCommunityEmissionVault() allows changing the vault address at any time without restrictions. When collection happens, rewards are transferred to the then-current vault. However, claimRewards() uses the vault address at claim time, not the address at deposit time. If the vault is changed after deposits but before claims, pending claims will target the new vault which does not have the old balances, causing transfer reverts and permanently bricking all claims for those epochs. Users lose access to their earned rewards.

Recommendations:
Implement one of the following: (1) Migrate balances atomically when changing the vault address; (2) Store the vault address per-epoch at deposit time so claims use the correct historical vault; (3) Require the old vault to have zero relevant balances before allowing the vault address to be changed; or (4) Only allow setting the vault once.