QuillAudits

AUDIT REPORT

May , 2025

For

Noma Protocol

Table of Content

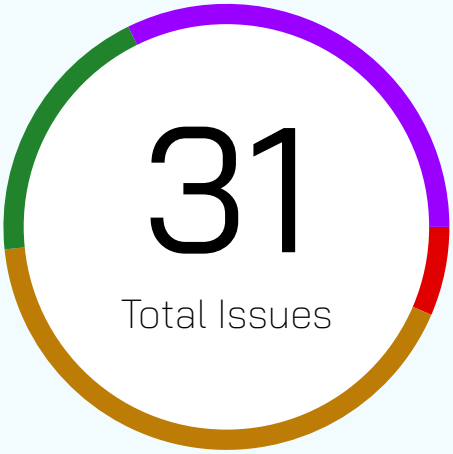
Executive Summary

Project name	Noma
Project URL	https://noma.money/
Overview	The Noma protocol is a novel token vault contract implementing a floor price for their ERC20 token by performing token buybacks using a rebalancing system based on the intrinsic minimum value calculations and circulating supply at any time. Noma is built using Uniswap V3, and is reliant on the Diamond proxy pattern to handle complexities introduced by its size. In addition to the ERC20 token capabilities, it also introduces a lending and borrowing mechanism to allow users take on “risk-free” loans that are protected from liquidations by the floor price mechanism which boosts capital efficiency.
Audit Scope	https://github.com/noma-protocol/core_contracts/tree/dev/src

Contracts in Scope	src/interfaces/IsNomaToken.sol src/interfaces/IDiamond.sol src/interfaces/IDiamondLoupe.sol src/interfaces/IFacet.sol src/interfaces/IVaultUpgrades.sol src/interfaces/IAddressResolver.sol src/interfaces/IDiamondCut.sol src/interfaces/IVault.sol src/interfaces/IModelHelper.sol src/interfaces/IDeployer.sol src/vault/BaseVault.sol | src/vault/LendingVault.sol src/vault/StakingVault.sol | src/vault/ExtVault.sol src/vault/deploy/EtchVault.sol src/vault/init/VaultFinalize.sol src/vault/init/VaultUpgrade.sol src/factory/TokenFactory.sol src/factory/ExtFactory.sol src/factory/DeployerFactory.sol src/factory/NomaFactory.sol src/factory/PresaleFactory.sol src/Deployer.sol | src/Diamond.sol | src/Resolver.sol src/controllers/supply/RewardsCalculator.sol src/controllers/supply/AdaptiveSupply.sol src/model/Helper.sol src/staking/Staking.sol src/staking/Gons.sol | src/libraries/Conversions.sol src/libraries/LibAppStorage.sol src/libraries/LibDiamond.sol src/libraries/LiquidityDeployer.sol src/libraries/LiquidityOps.sol src/libraries/Logarithm.sol | src/libraries/MathInt.sol src/libraries/Underlying.sol | src/libraries/Utils.sol src/bootstrap/Presale.sol | src/bootstrap/Bootstrap.sol src/bootstrap/token/pAsset.sol src/facetsDiamondLoupeFacet.sol src/facets/OwnershipFacet.sol src/types/Types.sol | src/init/DiamondInit.sol src/token/MockNomaTokenRestricted.sol src/token/MockNomaToken.sol src/token/RebaseToken.sol
Commit Hash	358936c3b788e84f387833a5b0b76f193f4fa0cd
Language	Solidity

Blockchain	Ethereum
Method	Manual Review, Functional Testing, Automated Testing
Review 1	March 7 - April 22, 2025
Updated Code Received	https://github.com/noma-protocol/core_contracts/tree/audit_ready
Review 2	April 28- May 6, 2025
Fixed In	54e4dda06bff679560fcb11688b023aa152130ec

Number of Issues per Severity



High	2 (6.45%)
Medium	13 (41.94%)
Low	6 (19.35%)
Informational	10 (32.26%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**2**	**13**	**6**	**2**
Acknowledged	0	0	0	**8**
Partially Resolved	0	0	0	0

Checked Vulnerabilities

- ☑ Access Management
- ☑ Arbitrary write to storage
- ☑ Centralization of control
- ☑ Ether theft
- ☑ Improper or missing events
- ☑ Logical issues and flaws
- ☑ Arithmetic Computations Correctness
- ☑ Race conditions/front running
- ☑ SWC Registry
- ☑ Re-entrancy
- ☑ Timestamp Dependence
- ☑ Gas Limit and Loops
- ☑ Exception Disorder
- ☑ Gasless Send
- ☑ Use of tx.origin
- ☑ Malicious libraries
- ☑ Compiler version not fixed
- ☑ Address hardcoded
- ☑ Divide before multiply
- ☑ Integer overflow/underflow
- ☑ ERC's conformance
- ☑ Dangerous strict equalities
- ☑ Tautology or contradiction
- ☑ Return values of low-level calls
- ☑ Missing Zero Address Validation
- ☑ Private modifier
- ☑ Revert/require functions
- ☑ Multiple Sends
- ☑ Using suicide
- ☑ Using delegatecall
- ☑ Upgradeable safety
- ☑ Using throw

- [x] Using inline assembly
- [x] Unsafe type inference
- [x] Style guide violation
- [x] Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools And Platforms Used For Audit

Remix IDE, Foundry, Solhint, Mythril, Slither, Solidity statistical analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational Issues

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

Security vulnerabilities identified that must be resolved and are currently unresolved.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/ impact of the security issue, but are not completely resolved.

High Severity Issues

Inaccessible defaultLoans() Function Prevents Collateral Seizure

Resolved

Path

src/vault/LendingVault.sol

Function

defaultLoans()

Description

The defaultLoans() function, responsible for seizing collateral on expired loans, is currently not callable due to its placement behind a proxy without proper exposure. It's protected with onlyInternalCalls, and there is no external interface (such as from the ExtVault contract) making it accessible. As a result, expired loans are not being defaulted and collateral cannot be reclaimed, potentially locking significant protocol funds.

Impact

If left unresolved, borrowers with expired loans can indefinitely retain collateral without repayment, leading to bad debt and capital inefficiency.

Recommendation

Expose the defaultLoans() function through the ExtVault or another externally accessible contract that complies with access controls. Ensure that only authorized actors (e.g., managers, keepers) can trigger it.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/ce25314ea253005f38bcd53895369efcf30ef4e2

withdraw() function transfers available balance instead of minAmountOut

Resolved

Path

src/bootstrap/Presale.sol

Function

withdraw()

Description

The withdraw() transfers the availableBalance for the token, instead it should be the amount calculated according to the number of pNoma tokens the user had. In withdraw() first it gets the balance of the user, then burns the p-assets. Then calculates the minAmountOut value. Finally it transfers the availableBalance instead of minAmountOut resulting in transfer of more than intended tokens.

Recommendation

To resolve the issue please make sure to change the availableBalance variable to minAmountOut in withdraw() function.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/07738cfaaf56801bcde10e8bf12e790ca881efc3

Medium Severity Issues

Loan Can Be Repaid After Expiry, Bypassing Collateral Seizure

Resolved

Path

src/vault/LendingVault.sol

Function

paybackLoan

Description

The paybackLoan() function lacks a proper expiry check, allowing borrowers to repay loans even after the loan.expiry timestamp. This violates expected loan mechanics where overdue loans should be considered defaulted and handled via collateral seizure (_seizeCollateral()), not repayment.

Recommendation

1. Enforce block.timestamp <= loan.expiry within paybackLoan() to prevent post-expiry repayments.
2. Ensure defaultLoans() is the only path for handling expired loans.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/60a0fdbe-bc7a8e8dda837df4011e60459caf67e4

Unrestricted Loan Operations on Behalf of Other Users

Resolved

Path

src/vault/ExtVault.sol & src/vault/LendingVault.sol

Function

borrow(), payback(), and roll()

Description

The borrow, payback, and roll functions allow any caller to initiate these actions on behalf of any address (who parameter), assuming the target user has given allowance to the vault contract.
This setup enables malicious users to:
- Take loans for other users (if they've approved tokens),
- Forcefully repay others' loans,
- Extend (roll) loan durations without consent.
There are no ownership, permission, or caller identity checks to ensure the msg.sender is authorized to act on behalf of who.

Recommendation

Implement access controls such that only:
- The borrower (who) themselves, or
- A whitelisted delegate/manager contract
can perform borrow, payback, or roll actions for that address.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/ee9a33113a7cc4f7f582259fb5bdc3575d50ada4

And

https://github.com/noma-protocol/core_contracts/commit/6b669859353052c53547b66d981d495995eaf459

Anyone Can Trigger Finalization Prematurely After Soft Cap

Resolved

Path

src/bootstrap/Presale.sol

Function

finalize()

Description

The finalize() function can be called by any external address once the soft cap is met. However, if the protocol intends to continue raising funds until the hard cap is reached, this behavior prematurely ends the presale and deploys liquidity.
There is no condition enforcing that the hard cap has been reached or the presale has expired, allowing a malicious actor or a front-runner to finalize early, disrupting fundraising goals

Recommendation

Restrict finalize() to either:
- Be callable only by the owner or authorized role
- Uncomment the expired check

Fixed in

https://github.com/noma-protocol/core_contracts/commit/45a92f68cbeb70309c0a813a5dd240a04064876f

Call to payReferrals() function will always fail

Resolved

Path

src/bootstap/Presale.sol

Function

payReferrals()

Description

The call to payReferrals() will always fail, because during the finalize() function call the excess eth has been already withdrawn and payReferrals makes it mandatory that it can be only called after the presale is finalized.

Recommendation

To resolve the issue it is recommended that the owner should make sure that referrals have been made first before withdrawing the excess eth. To fulfill the conditions owner can call the withdrawExcessEth() in from payReferrals() function or to call it separately like payReferrals(). Another option is to call payReferrals inside of the finalize() function before the _withdrawExcessEth() call is made.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/474ed4407d415f83f8a7cad569fedd989a2529a2

Contracts can be reinitialized without reentrancy protection

Resolved

Path

src/Deployer.sol, src/staking/Gons.sol

Function

initialize()

Description

The initialize() function in the Deployer contract lacks proper reentrancy protection despite the contract implementing a lock modifier elsewhere. This creates a vulnerability where the contract owner could reinitialize the contract, potentially overwriting critical contract state and parameters. If the owner is compromised at any time, the contract will be open to reinitialization and critical state variables that affect user interactions would be affected.

The same happens in the Gons contract, where the stakingContract can be reset by any user.

```
contract GonsToken is ERC20Permit {
    function initialize(address _stakingContract) external  { // @audit anyone can initialize the contract
        if (stakingContract != address(0)) {
            revert AlreadyInitialized();
        }
        stakingContract = _stakingContract;
        transfer(stakingContract, _totalSupply);
    }
```

Recommendation

Add a check to prevent reinitialization and apply the lock/initializer modifier for consistency.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/f28407a2451c1cbddee1874b27bf156de108b76

And

https://github.com/noma-protocol/core_contracts/commit/e07a5aa772eddf3c2f11f9fbd30ba513d2f037ef

AdaptiveSupply incorrectly uses totalSupply instead of deltaSupply leading to excessive token minting

Resolved

Path

src/controllers/supply/AdaptiveSupply.sol, src/libraries/LiquidityOps.sol

Function

reDeploy(), computeMintAmount()

Description

The LiquidityOps library incorrectly passes the total token supply instead of the change in supply (delta) to the AdaptiveSupply.computeMintAmount() function, leading to significantly inflated mint amounts during liquidity repositioning operations.

In the reDeploy() function of LiquidityOps.sol, when handling LiquidityType.Discovery positions and minting tokens, the contract passes the token's total supply as the first parameter to computeMintAmount()

However, examining the AdaptiveSupply.sol contract confirms that the first parameter is intended to be deltaSupply (the change in supply), not the total supply.

```
library LiquidityOps {
    function reDeploy(
        uint256 circulatingSupply = IModelHelper(addresses⬆.modelHelper)
        .getCirculatingSupply(
            addresses⬆.pool,
            addresses⬆.vault
        );

        uint256 totalSupply = IERC20Metadata(IUniswapV3Pool(addresses⬆.pool).token0()).totalSupply();

        (uint160 sqrtRatioX96,,,,,,) = IUniswapV3Pool(addresses⬆.pool).slot0();

        if (balanceToken0 < circulatingSupply / IVault(address(this)).getProtocolParameters().lowBalanceThresholdFactor) {
            if (isShift⬆) {
                // Mint unbacked supply
                (uint256 mintAmount) = IAdaptiveSupply(
                    addresses⬆.adaptiveSupplyController
                ).computeMintAmount(
                    totalSupply, // @audit AdaptiveSupply expects to use the deltaSupply in computeMintAmount but gets passed in totalSupply of token0
                    IVault(address(this)).getTimeSinceLastMint() > 0 ?
                    IVault(address(this)).getTimeSinceLastMint() :
                    1,
```

```
contract AdaptiveSupply { // audit NOTE - expected to be called by vaults
    function computeMintAmount(
        uint256 deltaSupply⬆,
        uint256 timeElapsed⬆,
        uint256 spotPrice⬆, // @audit-ok fixed-point math
        uint256 imv⬆
    ) public view returns (uint256 mintAmount) {
        if (timeElapsed⬆ == 0) revert TimeElapsedZero();
        if (deltaSupply⬆ == 0) revert DeltaSupplyZero();
        if (imv⬆ == 0) revert IMVZero();
```

Recommendation

Replace the use of totalSupply with an appropriate delta calculation in LiquidityOps.sol

Noma Team comment

Initially the algorithm used to compute mint amounts accepted a generic parameter, which I called deltaSupply. While I changed the algorithm several times, I settled on the current implementation. During real life testing I found out that total supply works better than the generic parameter delta supply.

See

https://github.com/noma-protocol/core_contracts/blob/5dfa744e9e25b14f01c755bb3ef09291ba39dfb0/test/supply/AdaptiveMint.t.sol#L151

Corroborated by test with

https://github.com/noma-protocol/core_contracts/commit/d4ade8d30c51cb6b2a19c9f425826fa1cabd8aff

Loan rollover function restricts extending loan duration, contrary expected behavior

Resolved

Path

src/vault/LendingVault.sol

Function

rollLoan()

Description

The rollLoan function in the LendingVault contract contains a logic error that prevents borrowers from extending their loan duration, which contradicts the typical expectation of a loan rollover functionality expected to allow new loans

The issue is in the validation check for newDuration. The function reverts if the new duration is longer than the original loan duration, which prevents users from extending their loans. This contradicts the expected behavior of a loan rollover, which typically allows users to extend their loan period in exchange for additional fees or interest.

```
contract LendingVault is BaseVault {
    function rollLoan(address who↑, uint256 newDuration↑) public onlyInternalCalls {
        // Fetch the loan position
        LoanPosition storage loan = _v.loanPositions[who↑];

        // Check if the loan exists
        if (loan.borrowAmount == 0) revert NoActiveLoan();

        // Check if the loan has expired
        if (block.timestamp > loan.expiry) revert LoanExpired();

        // Ensure the new duration is valid
        if (newDuration↑ == 0) revert InvalidDuration();
        if (newDuration↑ > loan.duration) revert InvalidDuration(); // @audit
```

Recommendation

Remove or modify the duration check to allow loan extensions.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/3ff96dbe3b6a371f09a0fb3af3dbcf25551158e

Incorrect Epoch Initialization and Duplicate Epoch Numbers

Resolved

Path

src/staking/Staking.sol

Function

notifyRewardAmount()

Description

The constructor() initializes the first epoch with number = 1 and distribute = 0. Later in notifyRewardAmount(), when totalEpochs == 1, it forcibly sets the reward to epoch.distribute (i.e., 0), leading to zero rewards for the first real epoch.
Additionally, the newly created epoch inside notifyRewardAmount() is assigned number = totalEpochs, which is already used by the previous epoch. This results in two epochs with the same number, which breaks the uniqueness assumption of epoch identifiers.

Recommendation

- Initialize the first epoch with number = 0 and only store it if needed as a placeholder
- Use epoch.number = totalEpochs before storing in epochs[totalEpochs] and incrementing
- Ensure epoch.number is always unique and in sync with totalEpochs

Fixed in

https://github.com/noma-protocol/core_contracts/commit/b2dbb0aad8b8208ca73da69d6d8ed69676a99d7d

Noma Token Proxy Not Upgradeable

Resolved

Path

src/factory/NomaFactory.sol

Function

N/A

Description

The upgradeToAndCall() function in the proxy contract triggers _authorizeUpgrade(), which is restricted by the onlyOwner modifier. However, the NomaFactory contract lacks any function or mechanism to invoke upgradeToAndCall() on the Noma token proxy.
As a result, the Noma token becomes effectively non-upgradeable, preventing any future upgrades, fixes, or enhancements.

Recommendation

Introduce an upgrade mechanism that allows a trusted admin (e.g., NomaFactory, a multisig, or timelock) to call upgradeToAndCall() securely, preserving upgradability as intended

Fixed in

https://github.com/noma-protocol/core_contracts/commit/a250d1d6938b440a4694bc7be524adf84144c4d2

Inaccurate Slippage Calculation

Resolved

Path

src/libraries/Uniswap.sol

Function

swap()

Description

The application of slippage especially on sqrtPriceX96 - a non-linear representation of price - results in a vastly exaggerated price buffer, which:
- Defeats the purpose of a slippage limit.
- Opens up the protocol to frontrunning or sandwich attacks.
- Risks executing swaps at unintended price points.

Recommendation

Apply slippage only once, ideally in human-readable price form (price = token1/token0), and then convert the result to sqrtPriceX96.

Alternatively, pass a slippage-adjusted price directly to the swap() function and remove any internal recalculation.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/9a03dd7ed91d51ba91a70c0b8cd8a24ab88391c2

Self-Referral Abuse Allows Users to Earn Discounts

Resolved

Path

src/bootstrap/Presale.sol

Function

deposit() & payReferrals()

Description

The deposit() function in the presale contract allows users to pass an arbitrary referralCode. However, this referralCode is later interpreted as the address to receive referral rewards in payReferrals()
- A user can generate their own referral code from their own address (e.g., bytes32(uint256(uint160(msg.sender)))) and pass it to the deposit() function.
- As a result, they receive a percentage of their own deposit back via the referral mechanism.
- This effectively gives them a discount on token purchases at the cost of the protocol.

Additionally:
- msg.value == 0 is allowed, which could result in unintended free mints or bloated contributor lists

Recommendation

•Restrict Self-Referrals:
o Check that msg.sender != address(uint160(uint256(referralCode))) during deposit()
•Use a Mapping for Referral Attribution
•Best to use nonreentrant modifier and access control for the payReferrals function

Fixed in

8b04984163c4894185e1e1ca96bd3328d42f8186

Inconsistent use of transfer methods in LendingVault could lead to failed token transfers

Resolved

Path

src/vault/LendingVault.sol

Function

rollLoan(), borrowFromFloor()

Description

The LendingVault contract inconsistently uses transfer and safeTransfer methods for token transfers, which could lead to failed transfers if tokens like USDC or USDT are blacklisted or if they return false instead of reverting. In the borrowFromFloor function, the contract uses transfer without checking the return value

LendingVault.sol
IERC20(_v.pool.token1()).transfer(who, borrowAmount - loanFees);

Recommendation

Use safeTransfer from the SafeERC20 library for all token transfers to ensure that failures are detected and handled appropriately

Fixed in

https://github.com/noma-protocol/core_contracts/commit/761ab042fdcd72948ba5a9fb715c83b1ea2f3a29

Front-Running the notifyReward Call to Exploit Rebase Gains

Resolved

Path

src/staking/Staking.sol

Function

stake() and unstake()

Description

The staking contract allows any user to call the stake() function at any time, including moments immediately before the notifyReward() function is called by the vault (which is assumed to be responsible for funding staking rewards and triggering a rebase event).
immediately before the notifyReward() function is called by the vault (which is assumed to be responsible for funding staking rewards and triggering a rebase event).
Since staking rewards are distributed via rebasing and sNOMA is minted 1:1 to the user's input (before any rebase logic runs), a malicious actor can:
1.Monitor for an impending notifyReward() call, possibly through off-chain bots or mempool monitoring.
2.Front-run the rebase by calling stake() with a large amount of NOMA just before the reward is distributed.
3.Rebase increases the value of sNOMA across all holders, including the attacker.
4.Immediately unstake after the rebase, realizing an instant, risk-free profit without long-term staking commitment.
This behavior disincentivizes long-term stakers and creates a competitive, unfair environment favoring sophisticated actors with the ability to monitor or front-run transactions.

Recommendation

1.Introduce a warm-up or lock-in period for newly staked tokens (e.g., 1 epoch or X blocks), during which rewards are not earned or withdrawal is restricted.
2.Snapshot-based reward accounting: Only users who were staked before the notifyReward() snapshot should receive rewards for that epoch.
3.Add a cooldown or minimum stake duration before unstaking is allowed.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/9a03dd7ed91d51ba91a70c0b8cd8a24ab88391c2

Fixed by

67e14bab6ffc804aae645aae4cc61f5b7ebd2883

Low Severity Issues

Presale can exceed hardcap

Resolved

Path

src/bootstrap/Presale.sol

Function

finalize()

Description

The deposit function in the Presale contract performs an incorrect validation check that could allow contributions to exceed the intended hardcap limit. The check is performed before the new contribution is added to the total, and it only reverts if the current balance is already greater than the hardcap.

Recommendation

Adjust the function this way to protect from overflowing the hardcap

```
function deposit(bytes32 referralCode) external payable {
    if (hasExpired()) revert PresaleEnded();
    if (finalized) revert AlreadyFinalized();

    // if (msg.value < MIN_CONTRIBUTION || msg.value > MAX_CONTRIBUTION) revert InvalidParameters();

    uint256 balance = address(this).balance;

-   if (balance > hardCap) revert HardCapExceeded();
+   if (balance + msg.value > hardCap) revert HardCapExceeded();

    // Track contributions
    contributions[msg.sender] += msg.value;
    totalRaised += msg.value;
```

Fixed in

06762639187916b5a9d171856b62f5d8fc723aef

Comments mismatch with slippage value

Resolved

Path

src/bootstrap/Presale.sol

Function

finalize()

Description

In Presale contract, slippage check is calculated in the finalize() function to ensure the stability of the price. But the issue is that the comment has mentioned slippage to be 0.1% and the actual slippage calculated in contract is 0.5% which is higher and is mismatched from the sentence.

Recommendation

It is recommended to change slippage value to match the comment or comment value to match the calculated value

Fixed in

5dfa744e9e25b14f01c755bb3ef09291ba39dfb0

stake() function takes _to value can cause revert

Resolved

Path

src/staking/Staking.sol

Function

stake()

Description

In staking contract, stake() function takes _to parameter where sNOMA tokens are minted but stakedBalances is increased for msg.sender. Then while in unstake() function _from parameter is used. So if user puts different address while staking then unstake() function might revert.

Recommendation

To resolve the issue it is recommended to remove the _to parameter and mint the tokens to the msg.sender

Fixed in

https://github.com/noma-protocol/core_contracts/commit/0724cfda3e9e03257f93b4afd4a5705862425a39

Noma Team comment

Acknowledged, it's better to use msg.sender across stake/unstake and remove the address parameter altogether.

GonsToken implementation only supports upward re-basing

Resolved

Path

src/staking/Gons.sol

Description

The GonsToken contract has a token rebasing model that allows for upward rebasing (expanding supply), but there is no mechanism for downward rebasing (contracting supply) as the interface documentation suggests. In the rebase() function, the implementation only adds to the total supply and never subtracts.

The function only accepts a non-negative supplyDelta value and adds it to the total supply. There is no parameter or functionality to indicate a negative rebase (contraction of supply).

```
contract GonsToken is ERC20Permit {
    function rebase(uint256 supplyDelta⬆)
        public
    {
        if (supplyDelta⬆ == 0) {
            emit LogRebase(_totalSupply);
        }

        _totalSupply = _totalSupply.add(supplyDelta⬆);

        if (_totalSupply > MAX_SUPPLY) {
```

```
interface IsNomaToken is IERC20 {
    /**
     * @notice Adjusts the total supply of the token by a specified amount.
     * @param supplyDelta The amount by which to adjust the total supply. Positive values increase the supply, negative values decrease it.
     * @dev This function is intended to be called to perform a rebase operation, modifying the total token supply.
     */
    function rebase(uint256 supplyDelta) external;
```

Recommendation

Adjust the contract logic / interface documentation appropriately.

Noma Team's comment

Rebases only inflate the supply, so they are unidirectional. Will update the commentary as it is not in line with the code.

https://github.com/noma-protocol/core_contracts/commit/fd3ca697ded8faa9e1e6d48d9b867a036788c78f which was later merged to “audit_ready”

Missing Emergency Switch, Contribution Constraints, and State Sync in Emergency Withdrawals

Resolved

Path

src/bootstrap/Presale.sol

Function

emergencyWithdrawal() & deposit()

Description

1. No Emergency Mode Toggle:
o The emergencyWithdrawal() function does not depend on any emergency toggle/switch, meaning it's passively time-gated (deadline + 30 days), rather than actively controlled by the protocol or owner.
o This limits response flexibility in critical failures (e.g., token price manipulation, stuck funds, or an attack on the protocol).

2.Improper Error Message:
o The revert message in emergencyWithdrawal() uses PresaleOngoing() when finalized is true, which is semantically incorrect. If the presale is finalized, emergency withdrawal shouldn't be allowed at all, or a different error like PresaleFinalized() should be used.

3.State Inconsistency in Emergency Withdrawal:
o While a user's individual contributions[msg.sender] is zeroed, the global totalRaised variable is not decremented.
o This leads to inflated accounting, which can break invariant assumptions during finalization or analytics.

4.Commented-Out Contribution Limits:
o In deposit(), the MIN_CONTRIBUTION and MAX_CONTRIBUTION checks are commented out

Recommendation

1.Introduce an Emergency Mode Switch
2.Replace PresaleOngoing() in emergencyWithdrawal() with a clearer message
3.Decrement totalRaised in the emergencyWithdrawal function
4.Remove comment and enforce min/max contribution limits
5.Add msg.value != 0 Check in deposit()

Fixed in

e0e1a1a11efc8ab0d5d1bfa696cf91eefca2a098

And

b40b773f28dc2c4b2d74ebcfddcb38100ad72a6a

And

5ce77cc86ff185c13a01878260d46e3200cd63ae

_authority Parameter Unused in Constructor

Resolved

Path

src/token/Gons.sol

Function

constructor()

Description

The _authority parameter is never used within the constructor or stored in a state variable.

Recommendation

Remove the _authority parameter from the constructor to clean up the code and avoid misleading usage

Fixed in

https://github.com/noma-protocol/core_contracts/commit/022fa828c6fc2f94a356a03c65b48af6f31983eb

Informational Severity Issues

No self transfer check in GonsToken contract

Resolved

Path

src/token/GonsToken.sol

Function

transfer()

Description

In GonsToken contract even if you self transfer the tokens it won't give error. In normal token transfer it is prohibited. Here in GonsToken it doesn't create issues or hinder the ability of protocol working. But to ensure ERC20 compatibility the check is needed.

Recommendation

To resolve the issue please add a require check for self transfer.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/cdfeec2f6fe63e0b18ad908b05a47f7a478906b8

Ambiguity in onlyVault modifier allows both vault and staking contract to call protected functions

Acknowledged

Path

src/staking/Staking.sol

Description

The onlyVault modifier in the staking contract is currently ambiguous, allowing both the vault and staking contract to call functions protected by this modifier. This could lead to confusion about which contract is intended to have exclusive access to these functions.

Recommendation

Clarify the onlyVault modifier to ensure it aligns with the intended access control policy. If the intention is to allow only the vault contract to call these functions, update the modifier.

Noma Team's Comment

This is a design choice

Unused modifiers

Acknowledged

Path

src/vault/LendingVault.sol

Description

There are few modifiers which are not used in contracts.
•onlyVault in LendingVault
The codebase contains several unused functions, variables, and libraries that contribute to unnecessary complexity and potential confusion for developers and auditors. These unused elements can obscure the code's true functionality and make maintenance more challenging.
•MockNomaTokenRestricted.sol and Bootstrap.sol are unused.
•In StakingVault.sol, the LiquidityOps library and functions _collectLiquidity and _transferExcessBalance are defined but not used.
•In NomaFactory.sol, the teamMultiSig address is declared but never set, rendering it ineffective.
•In Presale.sol, the variables MIN_CONTRIBUTION and MAX_CONTRIBUTION are calculated in the constructor but never used.

Recommendation

To resolve the issue please remove the unused modifier, functions, variables, libraries or use them accordingly in the contract.

Redundant condition in TokenFactory's _deployNomaToken function

Resolved

Path

src/factory/TokenFactory.sol

Description

The _deployNomaToken function in TokenFactory.sol contains a redundant condition that checks if the proxy address is greater than or equal to _token1. This check is unnecessary because the logic of the do...while loop already ensures that this condition will never be true when the loop exits.

Recommendation

The check below can be removed because it is redundant.

```
if (address(proxy) >= _token1) revert InvalidTokenAddressError(); // Redundant check
```

Noma Team's Comment

This is necessary to force the order of tokens in the Uniswap V3 pair.

Mismatched NatSpec comments and code structures can cause confusion

Acknowledged

Path

src/factory/TokenFactory.sol

Description

The codebase contains NatSpec comments that do not accurately reflect the corresponding code structures. This mismatch can lead to confusion for devs trying to understand the intended functionality and usage of these structures.

Recommendation

Update the codebase NatSpec comments.

Commented out solvency variant

Resolved

Path

src/vault/LendingVault.sol

Function

borrowFromFloor

Description

The solvency enforceSolvencyInvariant() function is being commented out in the borrrowFromFloor function in the lending vault.

Recommendation

Uncomment the function call.

Fixed in

https://github.com/noma-protocol/core_contracts/commit/6270164c745486c873ea3d21fc8a38c90fd76b5c

numVaults() is a gas-hungry function that can be optimized by reading the totalVaults variable

Acknowledged

Path

src/factory/NomaFactory.sol

Function

numVaults()

Description

The function is redundant since the contract already tracks totalVaults as a state variable that is incremented whenever a new vault is created.

```
contract NomaFactory {
    function numVaults() public view returns (uint256 result) { // @audit is redundant since totalVaults exists
        address[] memory _deployers = getDeployers();
        for (uint256 i = 0; i < _deployers.length; i++) {
            result += numVaults(_deployers[i]);
        }
    }

    /**
    * @notice Retrieves the number of vaults deployed by a specific deployer.
    * @param _deployer The address of the deployer.
    * @return The number of vaults deployed by the specified deployer.
    */
    ftrace | funcSig
    function numVaults(address _deployer⬆) public view returns (uint256) {
        return _vaults[_deployer⬆].length();
    }
```

Recommendation

numVaults should return the value for totalVaults instead, or create a getter function for the private totalVaults variable

Identical Function Signatures: mintTokens(address,uint256) in Multiple Contracts

Acknowledged

Path

src/factory/NomaFactory.sol & src/vault/LendingVault.sol

Function

mintTokens()

Description

Both LendingVault::mintTokens() and NomaFactory::mintTokens() share the same function selector f0dda65c. This could lead to confusion in logs, tooling, or proxy setups, especially in cases where introspection or low-level calls are involved.

Recommendation

Consider renaming one of the functions to avoid selector collision

Noma Team's Comment

Actually, since NomaFactory isn't behind the Diamond proxy, there's no selector collision possible

Missing Lender Controls Over Loan Rollovers

Acknowledged

Path

src/vault/LendingVault.sol

Function

rollLoan()

Description

The rollLoan function allows borrowers to extend (roll) their loan duration and increase borrow amounts, without any lender-side constraints or approval. There is no restriction on the number of rollovers, total duration, or borrower eligibility.

As a result:
•Borrowers may infinitely roll their loans
•Lenders may be locked into illiquid positions
•There is no mechanism for lenders to enforce repayment or liquidation, despite borrower expiry or undercollateralization risks

Recommendation

Introduce lender-side constraints and controls, such as:
•Maximum number of rollovers
•Maximum cumulative duration per loan

Noma Team's Comment

There is no undercollateralization risk as the loan can’t go under water.

Missing or Extra Struct Variables Compared to Doc-strings

Acknowledged

Path

src/types/Types.sol

Function

N/A

Description

Multiple struct definitions in the codebase contain discrepancies between their declared fields and the comments/documentation provided. This creates confusion and increases the likelihood of bugs, misconfigurations, or developer misunderstanding.
1.Struct: RewardParams
a. Missing: imv, spotPrice, totalSupply, kr
b. Unexpected: totalStaked (not mentioned in comment)
2.Struct: ProtocolParameters
a. Docstring omits: shiftAnchorUpperBips, slideAnchorUpperBips, and all the fee-related fields
b. These may be valid additions, but comments should be updated accordingly
3.Struct: LiquidityInternalPars
a. Docstring doesn't match the struct at all — likely refers to a different struct entirely
b. The struct itself is clear, but needs correct documentation

Recommendation

Synchronize docstrings with actual struct fields — ensure the comments reflect reality

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of Noma. We performed our audit according to the procedure described above.

Some issues of High (2), Medium (13), Low (6) and Informational (10) severities are found. Most of the issues were resolved by Noma Team and few of them were acknowledged .

Disclaimer

At QuillAudits, we have spent years helping projects strengthen their smart contract security. However, security is not a one-time event—threats evolve, and so do attack vectors. Our audit provides a security assessment based on the best industry practices at the time of review, identifying known vulnerabilities in the received smart contract source code.

This report does not serve as a security guarantee, investment advice, or an endorsement of any platform. It reflects our findings based on the provided code at the time of analysis and may no longer be relevant after any modifications. The presence of an audit does not imply that the contract is free of vulnerabilities or fully secure.

While we have conducted a thorough review, security is an ongoing process. We strongly recommend multiple independent audits, continuous monitoring, and a public bug bounty program to enhance resilience against emerging threats.

Stay proactive. Stay secure.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem

7+ Years of Expertise	**1M+** Lines of Code Audited
$30B+ Secured in Digital Assets	**1400+** Projects Secured

Follow Our Journey

AUDIT REPORT

May , 2025

For

Noma Protocol

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com