CredShields

Smart Contract Audit

Jan 30th, 2024 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Numa Money between Jan 9th, 2024, and Jan 22nd, 2024. And a retest was performed on Jan 28th, 2024.

Author

Shashank (Co-founder, CredShields)

shashank@CredShields.com

Reviewers

Aditya Dixit (Research Team Lead)

aditya@CredShields.com

Prepared for

Numa Money

Table of Contents

CRED SHiELDS

1. Executive Summary

Numa Money engaged CredShields to perform a smart contract audit from Jan 9th, 2024, to Jan 22nd, 2024. During this timeframe, Ten (10) vulnerabilities were identified. **A retest was performed on Jan 28th, 2024, and all the bugs have been addressed.**

During the audit, Zero (0) vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Numa Money" and should be prioritized for remediation, and fortunately, none were found.

The table below shows the in-scope assets and a breakdown of findings by severity per asset. Section 2.3 contains more information on how severity is calculated.

Assets in Scope	Critical	High	Medium	Low	info	Gas	Σ
Smart Contract	0	0	1	1	2	1	**5**
	0	**0**	**1**	**1**	**2**	**1**	**5**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in Smart Contract's scope during the testing window while abiding by the policies set forth by Smart Contract's team.

State of Security

To maintain a robust security posture, it is essential to continuously review and improve upon current security processes. Utilizing CredShields' continuous audit feature allows both Numa Money's internal security and development teams to not only identify specific vulnerabilities, but also gain a deeper understanding of the current security threat landscape.

To ensure that vulnerabilities are not introduced when new features are added, or code is refactored, we recommend conducting regular security assessments. Additionally, by analyzing the root cause of resolved vulnerabilities, the internal teams at Numa Money can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Numa Money can future-proof its security posture and protect its assets.

2. Methodology

Numa Money engaged CredShields to perform a Numa Money Smart Contract audit. The following sections cover how the engagement was put together and executed.

2.1 Preparation phase

The CredShields team meticulously reviewed all provided documents and comments in the smart-contract code to gain a thorough understanding of the contract's features and functionalities. They meticulously examined all functions and created a mind map to systematically identify potential security vulnerabilities, prioritizing those that were more critical and business-sensitive for the refactored code. To confirm their findings, the team deployed a self-hosted version of the smart contract and performed verifications and validations during the audit phase.

A testing window from Jan 9th, 2024, to Jan 22nd, 2024, was agreed upon during the preparation phase.

2.1.1 Scope

During the preparation phase, the following scope for the engagement was agreed-upon:

IN SCOPE ASSETS
https://github.com/NumaMoney/Numa/tree/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts

Table: List of Files in Scope

2.1.2 Documentation

Documentation was not required as the code was self-sufficient for understanding the project.

2.1.3 Audit Goals

CredShields uses both in-house tools and manual methods for comprehensive smart contract security auditing. The majority of the audit is done by manually reviewing the contract source code, following SWC registry standards, and an extended industry standard self-developed checklist. The team places emphasis on understanding core concepts, preparing test cases, and evaluating business logic for potential vulnerabilities.

2.2 Retesting phase

Numa Money is actively partnering with CredShields to validate the remediations implemented towards the discovered vulnerabilities.

2.3 Vulnerability classification and severity

CredShields follows OWASP's Risk Rating Methodology to determine the risk associated with discovered vulnerabilities. This approach considers two factors - Likelihood and Impact - which are evaluated with three possible values - **Low**, **Medium**, and **High**, based on factors such as Threat agents, Vulnerability factors, Technical and Business Impacts. The overall severity of the risk is calculated by combining the likelihood and impact estimates.

Overall Risk Severity				
Impact	HIGH	Medium	High	Critical
	MEDIUM	Low	Medium	High
	LOW	Note	Low	Medium
		LOW	MEDIUM	HIGH
	Likelihood			

Overall, the categories can be defined as described below -

1. **Informational**

   We prioritize technical excellence and pay attention to detail in our coding practices. Our guidelines, standards, and best practices help ensure software stability and reliability. Informational vulnerabilities are opportunities for improvement and do

not pose a direct risk to the contract. Code maintainers should use their own judgment on whether to address them.

2. Low

Low-risk vulnerabilities are those that either have a small impact or can't be exploited repeatedly or those the client considers insignificant based on their specific business circumstances.

3. Medium

Medium-severity vulnerabilities are those caused by weak or flawed logic in the code and can lead to exfiltration or modification of private user information. These vulnerabilities can harm the client's reputation under certain conditions and should be fixed within a specified timeframe.

4. High

High-severity vulnerabilities pose a significant risk to the Smart Contract and the organization. They can result in the loss of funds for some users, may or may not require specific conditions, and are more complex to exploit. These vulnerabilities can harm the client's reputation and should be fixed immediately.

5. Critical

Critical issues are directly exploitable bugs or security vulnerabilities that do not require specific conditions. They often result in the loss of funds and Ether from Smart Contracts or users and put sensitive user information at risk of compromise

or modification. The client's reputation and financial stability will be severely impacted if these issues are not addressed immediately.

6. **Gas**

   To address the risk and volatility of smart contracts and the use of gas as a method of payment, CredShields has introduced a "Gas" severity category. This category deals with optimizing code and refactoring to conserve gas.

2.4 CredShields staff

The following individual at CredShields managed this engagement and produced this report:

- **Shashank, Co-founder CredShields**
    - shashank@CredShields.com

Please feel free to contact this individual with any questions or concerns you have around the engagement or this document.

3. Findings

This chapter contains the results of the security assessment. Findings are sorted by their severity and grouped by the asset and SWC classification. Each asset section will include a summary. The table in the executive summary contains the total number of identified security vulnerabilities per asset per risk indication.

3.1 Findings Overview

3.1.1 Vulnerability Summary

During the security assessment, Ten (10) security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Potential Underflow in rewardsValue Function	Medium	Arithmetic Underflow
Floating and Outdated Pragma	Low	Floating Pragma (SWC-103)
Use Ownable2Step	Low	Missing Best Practices
Missing Best Practices	Low	Missing Best Practices
Centralized Token Withdrawal NumaVault::withdrawToken	Informational	Centralization Issue
Require with Empty Message	Informational	Code optimization

Gas Optimization in Require Statements	Gas	Gas Optimization
Public Constants can be Private	Gas	Public Constants can be Private
Unused Imports	Gas	Gas Optimization
Variables should be Immutable	Gas	Gas Optimization

Table: Findings in Smart Contracts

3.1.2 Findings Summary

SWC ID	SWC Checklist	Test Result	Notes
SWC-100	Function Default Visibility	Not Vulnerable	Not applicable after v0.5.X (Currently using solidity v >= 0.8.6)
SWC-101	Integer Overflow and Underflow	Not Vulnerable	The issue persists in versions before v0.8.X.
SWC-102	Outdated Compiler Version	Not Vulnerable	Version 0^.8.0 and above is used
SWC-103	Floating Pragma	Not Vulnerable	Contract uses floating pragma
SWC-104	Unchecked Call Return Value	Not Vulnerable	call() is not used
SWC-105	Unprotected Ether Withdrawal	Not Vulnerable	Appropriate function modifiers and require validations are used on sensitive functions that allow token or ether withdrawal.
SWC-106	Unprotected SELFDESTRUCT Instruction	Not Vulnerable	selfdestruct() is not used anywhere
SWC-107	Reentrancy	Not Vulnerable	No notable functions were vulnerable to it.
SWC-108	State Variable Default Visibility	Not Vulnerable	Not Vulnerable
SWC-109	Uninitialized Storage Pointer	Not Vulnerable	Not vulnerable after compiler version, v0.5.0

SWC-110	Assert Violation	Not Vulnerable	Asserts are not in use.
SWC-111	Use of Deprecated Solidity Functions	Not Vulnerable	None of the deprecated functions like block.blockhash(), msg.gas, throw, sha3(), callcode(), suicide() are in use
SWC-112	Delegatecall to Untrusted Callee	Not Vulnerable	Not Vulnerable.
SWC-113	DoS with Failed Call	Not Vulnerable	No such function was found.
SWC-114	Transaction Order Dependence	Not Vulnerable	Not Vulnerable.
SWC-115	Authorization through tx.origin	Not Vulnerable	tx.origin is not used anywhere in the code
SWC-116	Block values as a proxy for time	Not Vulnerable	Block.timestamp is not used
SWC-117	Signature Malleability	Not Vulnerable	Not used anywhere
SWC-118	Incorrect Constructor Name	Not Vulnerable	All the constructors are created using the constructor keyword rather than functions.
SWC-119	Shadowing State Variables	Not Vulnerable	Not applicable as this won't work during compile time after version 0.6.0
SWC-120	Weak Sources of Randomness from Chain Attributes	Not Vulnerable	Random generators are not used.
SWC-121	Missing Protection against Signature Replay Attacks	Not Vulnerable	No such scenario was found

SWC-122	Lack of Proper Signature Verification	Not Vulnerable	Not used anywhere
SWC-123	Requirement Violation	Not Vulnerable	Not vulnerable
SWC-124	Write to Arbitrary Storage Location	Not Vulnerable	No such scenario was found
SWC-125	Incorrect Inheritance Order	Not Vulnerable	No such scenario was found
SWC-126	Insufficient Gas Griefing	Not Vulnerable	No such scenario was found
SWC-127	Arbitrary Jump with Function Type Variable	Not Vulnerable	Jump is not used.
SWC-128	DoS With Block Gas Limit	Not Vulnerable	Not Vulnerable.
SWC-129	Typographical Error	Not Vulnerable	No such scenario was found
SWC-130	Right-To-Left-Override control character (U+202E)	Not Vulnerable	No such scenario was found
SWC-131	Presence of unused variables	Not Vulnerable	No such scenario was found
SWC-132	Unexpected Ether balance	Not Vulnerable	No such scenario was found
SWC-133	Hash Collisions With Multiple Variable Length Arguments	Not Vulnerable	abi.encodePacked() or other functions are not used.
SWC-134	Message call with hardcoded gas amount	Not Vulnerable	Not used anywhere in the code
SWC-135	Code With No Effects	Not Vulnerable	No such scenario was found
SWC-136	Unencrypted Private Data On-Chain	Not Vulnerable	No such scenario was found

Extended Test Cases

1. Checked all the imports and inherited contracts used by the code and made sure they were updated, and valid.
2. Made sure that all the imported files were used or referenced in the code and highlighted the ones that were not used/dead code.
3. Checked the pragma to make sure it was updated and tested the solidity changelog for any potential vulnerabilities affecting the already-used pragma version.
4. Made sure that business-critical functions have properly configured pausable modifiers.
5. Checked access control of all the functions to make sure onlyOwner modifier is enforced and that they can't be called by external users.
6. The function rewardsValue() was interacting with the Chainlink oracle. We checked for all the possible exploitation scenarios related to Chainlink price fetch features such as circuit breaker scenarios, stale values, deviation and time updates, timestamp validations, and availability of price feed on different chains.
7. We also checked the path used for Chainlink price feed is the best optimal path.
8. The function also subtracted the snapshot token value for 24 hours but the difference was vulnerable to an underflow due to missing validations.
9. We tested multiple scenarios related to price oracle manipulation including manipulating the pool, inflating/deflating values, flash loan scenarios, issues related to swapping tokens to manipulate price, scenarios to take advantage of the 24-hour delay period in price update,
10. We tested cases for tokens interacting with the contract such as if they take excess fees, and if the fee was properly calculated and accounted for, as well as other scenarios in the weird-ERC20 token checklist.
11. We tested all the possible scenarios related to input validation in all the functions, thresholds, and external calls.
12. We tested the functions for cross-function reentrancy since nonReentrant prevents reentrancy issues only to some extent.
13. The functions TokenToNuma and NumaToToken calculated the amount values using the total ETH values of all the synthetics stored across all the vaults. We tried to find precision loss and underflow/overflow potential scenarios in the arithmetic calculations.
14. Checked the arithmetic calculations to make sure multiplication was being done before division to minimize precision losses in functions TokenToNuma and NumaToToken.
15. Tested the implications of using 0 values as user-controlled parameters across the contract.
16. We also tried scenarios when using fresh price oracle values instead of the snapshot price.
17. We tested the contracts for centralization risks and issues, noticed a function called withdrawTokens, and notified the client.

18. Tested the decay decrement logic and formula to make sure it is working as intended.
19. Validated the fee logic to make sure calculations were precise and accurate when buying and selling tokens.
20. The reward extraction mechanism is working as expected and is rebasing the value of the vault from time to time.
21. Validated the test cases and checked for any improper results or reverts, and also tried to change the input values.
22. Verified all the config addresses mentioned in the JSON files.
23. Validated the require statements to check if partial and strict threshold value checks are properly implemented.
24. Checked the vaults add and remove flow, and noticed that EnumerableSet was being used and validations were already implemented.
25. Tested scenarios using buying and selling tokens with multiple vaults trying to drain tokens such as using LP tokens of vault A is used in vault B to sell the tokens.
26. Validated the ordering of tokens and their respective formulas for calculating the ETH value of the token inside OracleUtils.
27. Tested the nuAssetManager function to remove assets from the list. It was working properly and shifting the values as well.

4. Remediation Status

Numa Money is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on Jan 28th, 2024, and all the issues have been addressed.**

Also, the table shows the remediation status of each finding.

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Potential Underflow in rewardsValue Function	Medium	**Fixed [28/01/2024]**
Floating and Outdated Pragma	Low	**Fixed [28/01/2024]**
Use Ownable2Step	Low	**Fixed [28/01/2024]**
Missing Best Practices	Low	**Fixed [28/01/2024]**
Centralized Token Withdrawal NumaVault::withdrawToken	Informational	**Fixed [01/02/2024]**
Require with Empty Message	Informational	**Fixed [28/01/2024]**
Gas Optimization in Require Statements	Gas	**Fixed [28/01/2024]**
Public Constants can be Private	Gas	**Fixed [28/01/2024]**

Unused Imports	Gas	**Fixed [28/01/2024]**
Variables should be Immutable	Gas	**Fixed [28/01/2024]**

Table: Summary of findings and status of remediation

5. Bug Reports

Bug ID #1 [Fixed]

Potential Underflow in `rewardsValue` Function

Vulnerability Type
Arithmetic Underflow

Severity
Medium

Description
The `rewardsValue` function in the provided smart contract has a potential arithmetic underflow when calculating the difference (`diff`) between the current token price (`currentvalueWei`) and the last token price (`last_lsttokenvalueWei`). If the new price is less than the last one, this subtraction could result in an underflow.

Vulnerable Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L263-L271

Impacts
An arithmetic underflow can lead to unexpected behavior and incorrect results. In this context, if the new token price is less than the last one, the subtraction could result in a revert underflow. This may lead to DoS.

Remediation
To mitigate the risk of underflow, it is recommended to add a check to ensure that the new token price is greater than or equal to the last one before performing the subtraction.

Retest

Underflow validation is now handled and appropriate value is returned.

Bug ID #2 [Fixed]

Floating and Outdated Pragma

Vulnerability Type

Floating Pragma (SWC-103)

Severity

Low

Description

Locking the pragma helps ensure that the contracts do not accidentally get deployed using an older version of the Solidity compiler affected by vulnerabilities.
The contract allowed floating or unlocked pragma to be used, i.e., >= 0.8.9. This allows the contracts to be compiled with all the solidity compiler versions above the limit specified. The following contracts were found to be affected -

Affected Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/nuAssets/nuAssetManager.sol#L2

Impacts

If the smart contract gets compiled and deployed with an older or too recent version of the solidity compiler, there's a chance that it may get compromised due to the bugs present in the older versions or unidentified exploits in the new versions.
Incompatibility issues may also arise if the contract code does not support features in other compiler versions, therefore, breaking the logic.
The likelihood of exploitation is low.

Remediation

Keep the compiler versions consistent in all the smart contract files. Do not allow floating pragmas anywhere. It is suggested to use the 0.8.22 pragma version
Reference: https://swcregistry.io/docs/SWC-103

Retest

The pragma has been fixed and updated to 0.8.20.

Bug ID #3 [Fixed]

Use Ownable2Step

Vulnerability Type

Missing Best Practices

Severity

Low

Description

The "Ownable2Step" pattern is an improvement over the traditional "Ownable" pattern, designed to enhance the security of ownership transfer functionality in a smart contract. Unlike the original "Ownable" pattern, where ownership can be transferred directly to a specified address, the "Ownable2Step" pattern introduces an additional step in the ownership transfer process. Ownership transfer only completes when the proposed new owner explicitly accepts the ownership, mitigating the risk of accidental or unintended ownership transfers to mistyped addresses.

Affected Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/nuAssets/nuAssetManager.sol#L30
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/VaultManager.sol#L18
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L90
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaOracle.sol#L18

Impacts

Without the "Ownable2Step" pattern, the contract owner might inadvertently transfer ownership to an unintended or mistyped address, potentially leading to a loss of control over the contract. By adopting the "Ownable2Step" pattern, the smart contract becomes more resilient against external attacks aimed at seizing ownership or manipulating the contract's behavior.

Remediation

It is recommended to use either Ownable2Step or Ownable2StepUpgradeable depending on the smart contract.

Retest

The contracts are now using Ownable2step instead of Ownable.

Bug ID #4 [Fixed]

Missing Events in Functions

Vulnerability Type

Missing Best Practices

Severity

Low

Description

Events are inheritable members of contracts. When you call them, they cause the arguments to be stored in the transaction's log—a special data structure in the blockchain. These logs are associated with the address of the contract which can then be used by developers and auditors to keep track of the transactions. The contract was found to be missing these events on certain critical functions which would make it difficult or impossible to track these transactions off-chain.

Affected Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/nuAssets/nuAssetManager.sol#L57-L69
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/nuAssets/nuAssetManager.sol#L75-L91
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/VaultManager.sol#L33-L37
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/VaultManager.sol#L42-L46
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L113-L117
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L273-L278

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L324-L328
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L334-L338

Impacts

Events are used to track the transactions off-chain, and missing these events on critical functions makes it difficult to audit these logs if they're needed at a later stage.

Remediation

Consider emitting events for the functions mentioned above. It is also recommended to have the addresses indexed.

Retest

The functions are now emitting events.

Bug ID #5 [Fixed]

Centralized Token Withdrawal NumaVault::withdrawToken

Vulnerability Type
Centralization Issue

Severity
Informational

Description
The contract includes a function (withdrawToken) that allows the owner to withdraw a specified amount of tokens, transferring them to the contract owner. The audit issue raised questions about the centralization aspect of transferring all tokens to the owner without clear justification.

Vulnerable Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L562-L566

Impacts
The centralized token withdrawal may raise concerns about transparency and decentralization, as all tokens are transferred to the owner without clear justification. This could lead to a lack of accountability or unintended consequences.

Remediation

1. Clearly document the reasons for centralizing token withdrawals to the owner.
2. Consider implementing a more decentralized approach, such as a multi-sig or governance mechanism, for handling token withdrawals.
3. Ensure the contract owner's actions align with the project's goals and governance structure.

Retest
The function has been removed.

Bug ID #6 [Fixed]

Require with Empty Message

Vulnerability Type
Code optimization

Severity
Informational

Description
During analysis; multiple **require** statements were detected with empty messages. The statement takes two parameters, and the message part is optional. This is shown to the user when and if the **require** statement evaluates to false. This message gives more information about the conditional and why it gave a false response.

Affected Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L286

Impacts
Having a short descriptive message in the **require** statement gives users and developers more details as to why the conditional statement failed and helps in debugging the transactions.

Remediation
It is recommended to add a descriptive message, no longer than 32 bytes, inside the **require** statement to give more detail to the user about why the condition failed.

Retest
A descriptive message is now added to the require statement.

Bug ID #7 [Fixed]

Gas Optimization in Require Statements

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The **require()** statement takes an input string to show errors if the validation fails.
The strings inside these functions that are longer than **32 bytes** require at least one additional MSTORE, along with additional overhead for computing memory offset and other parameters. For this purpose, having strings lesser than 32 bytes saves a significant amount of gas.

Vulnerable Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L380
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L404
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L424

Impacts
Having longer require strings than **32 bytes** costs a significant amount of gas.

Remediation
It is recommended to shorten the strings passed inside **require()** statements to fit under **32 bytes**. This will decrease the gas usage at the time of deployment and at runtime when the validation condition is met.

Retest

The require messages are shortened to less than 32 bytes to save gas.

Bug ID #8 [Fixed]

Public Constants can be Private

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Public constant variables cost more gas because the EVM automatically creates getter functions for them and adds entries to the method ID table. The values can be read from the source code instead.

Affected Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L66
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L67
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L68

Impacts
Public constants are more costly due to the default getter functions created for them, increasing the overall gas cost.

Remediation
If reading the values for the constants is not necessary, consider changing the public visibility to private.

Retest
The public constants have been made private.

Bug ID #9 [Fixed]

Unused Imports

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was importing some contracts or libraries that were not used anywhere in the code. This increases the gas cost and the overall contract's complexity.

Affected Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/VaultOracle.sol#L6
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/VaultOracle.sol#L7
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/nuAssets/nuAssetManager.sol#L7
- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L10

Impacts
Unused imports in smart contracts can lead to an increase in the size of the code, making it more difficult to verify and potentially slowing down its execution. Moreover, having unused code in a smart contract can also increase the attack surface by potentially introducing vulnerabilities that can be exploited by malicious actors. This can lead to security issues and compromise the integrity of the contract.

Additionally, including unused imports in smart contracts can also increase deployment and gas costs, making it more expensive to deploy and run the contract on the Ethereum network.

Remediation
It is recommended to remove the import statement if the external contracts or libraries are not used anywhere in the contract.

Retest
Unused imported contracts have been removed.

Bug ID #10 [Fixed]

Variables should be Immutable

Vulnerability Type
Gas Optimization

Severity
Gas

Description
Declaring state variables that are not updated following deployment as immutable can save gas costs in smart contract deployments and function executions. Immutable state variables are those that cannot be changed once they are initialized, and their values are set permanently.

By declaring state variables as immutable, the compiler can optimize their storage in a way that reduces gas costs. Specifically, the compiler can store the value directly in the bytecode of the contract, rather than in storage, which is a more expensive operation.

Affected Code

- https://github.com/NumaMoney/Numa/blob/067235f9c1a95e1f63c8099cbeb3ecbaf6e51ab3/contracts/NumaProtocol/NumaVault.sol#L55

Impacts
Gas usage is increased if the variables that are not updated outside of the constructor are not set as immutable.

Remediation
An "`immutable`" attribute should be added in the parameters that are never updated outside of the constructor to save the gas.

Retest
The variable has been set to be immutable.

6. Disclosure

The Reports provided by CredShields is not an endorsement or condemnation of any specific project or team and do not guarantee the security of any specific project. The contents of this report are not intended to be used to make decisions about buying or selling tokens, products, services, or any other assets and should not be interpreted as such.

Emerging technologies such as Smart Contracts and Solidity carry a high level of technical risk and uncertainty. CredShields does not provide any warranty or representation about the quality of code, the business model or the proprietors of any such business model, or the legal compliance of any business. The report is not intended to be used as investment advice and should not be relied upon as such.

CredShields Audit team is not responsible for any decisions or actions taken by any third party based on the report.