Audited by

CredShields

Smart Contract Audit

January 6th, 2025 • CONFIDENTIAL

Description

This document details the process and result of the Smart Contract audit performed by CredShields Technologies PTE. LTD. on behalf of Metaco Intelligence Corporation between December 26th, 2024, and January 2nd, 2025. A retest was performed on January 3rd, 2025.

Author
Shashank (Co-founder, CredShields) shashank@CredShields.com

Reviewers
Aditya Dixit (Research Team Lead), Shreyas Koli(Auditor), Naman Jain (Auditor), Sanket Salavi (Auditor)

Prepared for
Metaco Intelligence Corporation

Table of Contents

1. Executive Summary --------------------

Metaco Intelligence Corporation engaged CredShields to perform a smart contract audit from December 26th, 2024, to January 2nd, 2025. During this timeframe, 13 vulnerabilities were identified. **A retest was performed on January 3rd, 2025, and all the bugs have been addressed.**

During the audit, 3 vulnerabilities were found with a severity rating of either High or Critical. These vulnerabilities represent the greatest immediate risk to "Metaco Intelligence Corporation" and should be prioritized for remediation.

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
ZeUSD Contracts	1	2	3	4	0	3	**13**
	1	**2**	**3**	**4**	**0**	**3**	**13**

Table: Vulnerabilities Per Asset in Scope

The CredShields team conducted the security audit to focus on identifying vulnerabilities in ZeUSD's scope during the testing window while abiding by the policies set forth by Metaco Intelligence Corporation'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 Metaco Intelligence Corporation'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 Metaco Intelligence Corporation can implement both manual and automated procedures to eliminate entire classes of vulnerabilities in the future. By taking a proactive approach, Metaco Intelligence Corporation can future-proof its security posture and protect its assets.

2. The Methodology --------------------

Metaco Intelligence Corporation engaged CredShields to perform a ZeUSD 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 December 26th, 2024, to January 2nd, 2025, 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/0xZothio/zeusd-contracts/tree/efda4764e68a83e0e73dbcd1c752dc0376f38b37

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

Metaco Intelligence Corporation 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, and 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	● None	● 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 about the engagement or this document.

3. Findings Summary --------------------

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, 13 security vulnerabilities were identified in the asset.

VULNERABILITY TITLE	SEVERITY	SWC | Vulnerability Type
Incomplete logic in _handleSecondaryAssetWithdraw()	Critical	Incomplete Implementation
Uninitialized teller variable causing DoS	High	Denial of service (Dos)
Failed to inherit UUPSUpgradable contract	High	Missing Functionality
Incorrect emergencyMode activation logic	Medium	Incorrect Logic
Missing storage gap in upgradable contracts	Medium	Storage Layout Conflict
Chainlink Oracle min/max price validation	Medium	Missing Input Validation
Failed to initialize AccessControlUpgradeable	Low	Missing Functionality
Use of Multiple Pragma Versions	Low	Missing Best Practices

Floating and Outdated Pragma	Low	Floating Pragma (SWC-103)
Use Ownable2step	Low	Missing Best Practices
Cheaper inequalities in if()	Gas	Gas Optimization
Public constants can be private	Gas	Gas Optimization
Gas optimization in increments	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	Vulnerable	Bug ID #9
SWC-103	Floating Pragma	Vulnerable	Bug ID #9
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	Vulnerable	Bug ID #2

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

4. Remediation Status ------------------

Metaco Intelligence Corporation is actively partnering with CredShields from this engagement to validate the discovered vulnerabilities' remediations. **A retest was performed on January 3rd, 2025, and all the issues have been addressed.**

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

VULNERABILITY TITLE	SEVERITY	REMEDIATION STATUS
Incomplete logic in _handleSecondaryAssetWithdraw()	Critical	**Fixed** [Jan 3rd, 2025]
Uninitialized teller variable causing DoS	High	**Fixed** [Jan 3rd, 2025]
Failed to inherit UUPSUpgradable contract	High	**Fixed** [Jan 3rd, 2025]
Incorrect emergencyMode activation logic	Medium	**Fixed** [Jan 3rd, 2025]
Missing storage gap in upgradable contracts	Medium	**Fixed** [Jan 3rd, 2025]
Chainlink Oracle min/max price validation	Medium	**Won't Fix** [Jan 3rd, 2025]
Failed to initialize AccessControlUpgradeable	Low	**Fixed** [Jan 3rd, 2025]
Use of Multiple Pragma Versions	Low	**Fixed** [Jan 3rd, 2025]
Floating and Outdated Pragma	Low	**Partially Fixed** [Jan 3rd, 2025]
Use Ownable2step	Low	**Won't Fix** [Jan 3rd, 2025]
Cheaper inequalities in if()	Gas	**Won't Fix** [Jan 3rd, 2025]

Public constants can be private	Gas	**Won't Fix** [Jan 3rd, 2025]
Gas optimization in increments	Gas	**Fixed** [Jan 3rd, 2025]

Table: Summary of findings and status of remediation

5. Bug Reports ------------------------

Bug ID #1 [Fixed]

Incomplete logic in _handleSecondaryAssetWithdraw()

Vulnerability Type
Incomplete Implementation

Severity
Critical

Description
The _handleSecondaryAssetWithdraw() function in the contract is designed to handle the withdrawal of secondary assets by making an external call to a USYC-based contract. However, the logic within the function is commented out, leaving the function without any implementation. As a result, any call to _handleSecondaryAssetWithdraw() will fail to process the withdrawal of secondary assets. This causes users to be unable to withdraw their funds, leading to significant disruption in the functionality of the contract. Since _handleSecondaryAssetWithdraw() is called in handleWithdraw() when the asset is not USYC, all withdrawals of secondary assets will fail.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L262-L273

Impacts
Users won't be able to withdraw secondary assets makes the contract non-functional for users holding non-USYC assets.

Remediation
It is recommended to uncomment the exiting code or add new implementation of the _handleSecondaryAssetWithdraw() function.

Retest
This issue has been fixed by adding a logic in _handleSecondaryAssetWithdraw() function.

Bug ID #2 [Fixed]

Uninitialized teller variable causing DoS

Vulnerability Type
Denial of service (Dos)

Severity
High

Description
The USYCSubVault.sol contract defines a teller variable, which is used as the target address for external calls. However, its value is not initialized or set anywhere in the contract. This causes the handleDeposit() function to revert whenever it attempts to call the buy() function on the teller address. Since the teller address is not initialized, it defaults to the zero address (0x0). Any external call to the zero address will revert, causing the handleDeposit() function always to fail when invoked.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L214

Impacts
The handleDeposit() function will revert for all calls, rendering deposits non-functional.

Remediation
It is recommended to ensure that the teller variable is assigned a valid address during the contract's deployment or initialization

Retest
This issue has been fixed by adding a setTeller() function which updates the teller's address

Bug ID #3 [Fixed]

Failed to inherit UUPSUpgradable contract

Vulnerability Type
Missing Functionality

Severity
High

Description
The contract as written does not include any mechanism to facilitate upgrades, despite containing an _authorizeUpgrade() function that appears intended to restrict who can authorize such actions. This discrepancy can lead to confusion, potentially causing the contract to be deployed with the false assumption that it is upgradeable. The absence of inheritance from a recognized upgradeable module, such as OpenZeppelin's UUPSUpgradeable, renders the _authorizeUpgrade function non-functional. Consequently, any attempts to upgrade the contract will fail, effectively locking the contract logic in its initial state.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol

Impacts
If deployed without correction, the contract cannot adapt to future requirements, leaving it vulnerable to external threats or obsolescence.

Remediation
To remediate this issue, the contract should explicitly inherit from a proxy-compatible upgrade mechanism such as UUPSUpgradeable. Additionally, the initialize function must include a call to the appropriate initialization function for the selected upgradeable module, ensuring proper setup of upgradeability features.

Retest
This issue is partially resolved by inheriting the UUPSUpgradeable.sol contract.

Bug ID #4 [Fixed]

Incorrect emergencyMode activation logic

Vulnerability Type
Incorrect Logic

Severity
Medium

Description
The enableEmergencyMode() function is responsible for enabling the contract's emergency mode. During this mode, the lastEmergencyAction timestamp is set to the current block's timestamp (block.timestamp). The admin can withdraw funds after a predefined delay (EMERGENCY_DELAY) by calling the withdrawEmergency() function. However, the current implementation allows the admin to repeatedly call the enableEmergencyMode() function, which resets the lastEmergencyAction timestamp every time it is invoked. This behavior enables the admin to manipulate the delay period (EMERGENCY_DELAY) indefinitely by resetting the lastEmergencyAction timestamp, thus preventing the timely withdrawal of funds in emergencies or enabling malicious intent.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L324-L329
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/ZTLNPrimeSubVault.sol#L324-L329

Impacts
Legitimate withdrawals in emergencies can be delayed indefinitely by resetting lastEmergencyAction. A malicious admin could manipulate the system, potentially locking users' funds indefinitely.

Remediation
It is recommended to add a condition in enableEmergencyMode() to ensure that it can only be invoked if emergencyMode is currently false.

Retest

This issue has been fixed by adding a condition to restrict enableEmergencyMode() to only be callable when emergencyMode is false.

Bug ID #5 [Fixed]

Missing storage gap in upgradable contracts

Vulnerability Type
Storage Layout Conflict

Severity
Medium

Description
The contracts are an upgradeable contract using the UUPS pattern. The contract defines storage variables but lacks a storage gap. The contract defines different variables. Due to the absence of a storage gap in the contracts, adding new storage variables can potentially overwrite the storage layout of the contract, leading to critical misbehaviors.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/ZeUSD.sol
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/ZTLNPrimeSubVault.sol

Impacts
Without a storage gap, new storage variables introduced in the contracts can overwrite the beginning of the storage layout, causing unexpected behavior and potentially severe vulnerabilities.

Remediation
Introduce a storage gap in each contract to reserve space for future storage variables without affecting the inherited contract's storage layout. Or you can use namespace variables.

Retest
This issue has been fixed by implementing dedicated namespaced storage slots.

Bug ID #6 [Won't Fix]

Chainlink Oracle min/max price validation

Vulnerability Type
Missing Input Validation

Severity
Medium

Description
Chainlink has a library AggregatorV3Interface with a function called latestRoundData(). This function returns the price feed among other details for the latest round.
Chainlink aggregators have a built-in circuit breaker if the price of an asset goes outside of a predetermined price band. The result is that if an asset experiences a huge drop in value, the price of the oracle will continue to return the minPrice instead of the actual price of the asset.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/ZTLNPrimeSubVault.sol#L160-L194
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L160-L194

Impacts
This would allow users to store their allocations with the asset but at the wrong price.

Remediation
The contract should check the returned answer/price against the minPrice/maxPrice and revert if the answer is outside of the bounds.

```
if (price >= maxPrice or price <= minPrice) revert(); // eg
```

Retest
Client Comment: The additional checks in this specific context would not provide tangible benefits. However, for a generic implementation or use with more volatile assets, implementing such validations would be a prudent measure. We appreciate the suggestion and will consider adding such checks in future contracts where the oracle's safeguards or the asset's volatility necessitate it.

Bug ID #7 [Fixed]

Failed to initialize AccessControlUpgradeable

Vulnerability Type
Missing Functionality

Severity
Low

Description
The contract inherits AccessControlUpgradeable but does not call its initializer function, __AccessControl_init(), within its initialize() function. While the initializer is currently empty and does not directly affect the contract's behavior, omitting its invocation creates a potential risk for future maintenance and compatibility. Should the parent contract's initializer logic be updated in a future version of the OpenZeppelin library, contracts that fail to call this method could remain uninitialized, leading to undefined or insecure behavior.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/ZeUSD.sol#L51-L64

Impacts
This issue impacts the maintainers and users of the contract by introducing the possibility of future incompatibilities. If OpenZeppelin updates the AccessControlUpgradeable implementation to include critical setup logic, the lack of proper initialization in the current contract could result in functional failures.

Remediation
To address this, the initialize() function should explicitly invoke __AccessControl_init().

Retest
This issue has been fixed by initializing access control during the initialization.

Bug ID #8 [Fixed]

Use of Multiple Pragma Versions

Vulnerability Type
Missing Best Practices

Severity
Low

Description
The contracts were found to be using multiple Solidity Compiler versions across different solidity files. This is not a good coding practice because different versions of the compiler have different caveats, breaking changes and introducing vulnerabilities.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/LZAdapter.sol#L2
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/ZeUSD.sol#L3

Impacts
Having different pragma versions across multiple contracts increases the chances of introducing vulnerabilities since each solidity version have their own set of issues and coding practices. Some major version upgrades may also break the contract logic if not handled properly.

Remediation
Instead of using different versions of the Solidity compiler with different bugs and security checks, it is better to use one version across all contracts.

Retest
This vulnerability is fixed.

Bug ID #9 [Partially 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., ^0.8.22, 0.8.24. 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/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L2
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/ZTLNPrimeSubVault.sol#L2
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/LZAdapter.sol#L2
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/ZeUSD.sol#L3
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/ZeUSD_OFT.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 0.8.28 pragma version.
Reference: https://swcregistry.io/docs/SWC-103

Retest

Client Comment: Solidity 0.8.23 has been selected to align with the compatibility requirements of the Metis chain, which does not yet support EVM mCodes introduced in later Solidity versions like 0.8.28.

Bug ID #10 [Won’t Fix]

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/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/LZAdapter.sol#L8
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/ZeUSD.sol#L17

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:
Client Comment: Thank you for suggesting the adoption of the "Ownable2Step" pattern. While we recognize the added security benefits of requiring explicit acceptance for ownership transfers, we

believe the current implementation of Ownable adequately meets our security and operational needs.

Bug ID #11 [Won't Fix]

Cheaper inequalities in if( )

Vulnerability Type
Gas Optimization

Severity
Gas

Description
The contract was found to be doing comparisons using inequalities inside the "if" statement. When inside the "if" statements, non-strict inequalities (>=, <=) are usually cheaper than the strict equalities (>, <).

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L334
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L355

Impacts
Using strict inequalities inside "if" statements costs more gas.

Remediation
It is recommended to go through the code logic, and, **if possible**, modify the strict inequalities with the non-strict ones to save gas as long as the logic of the code is not affected.

Retest:
We prioritize precise control and security in these scenarios, which is why the strict inequality checks remain a deliberate design choice.

Bug ID #12 [Won't Fix]

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/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/ZeUSD_OFT.sol#L15
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/implementations/ZeUSD.sol#L28
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/ZTLNPrimeSubVault.sol#L68
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/ZTLNPrimeSubVault.sol#L83
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L66
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L83

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

Client Comments: By maintaining the ADMIN_ROLE as public, the contract ensures robust security, seamless integration, and adherence to best practices, providing long-term benefits over short-term gas savings.

Bug ID #13 [Fixed]

Gas optimization in increments

Vulnerability Type
Gas optimization

Severity
Gas

Description
The contract uses two for loops**,** which use post increments for the variable "**i**".
The contract can save some gas by changing this to **++i**.
++i costs less gas compared to **i++** or **i += 1** for unsigned integers. In **i++**, the compiler has to create a temporary variable to store the initial value. This is not the case with **++i** in which the value is directly incremented and returned, thus, making it a cheaper alternative.

Affected Code

- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/ZTLNPrimeSubVault.sol#L305
- https://github.com/0xZothio/zeusd-contracts/blob/efda4764e68a83e0e73dbcd1c752dc0376f38b37/contracts/subVaults/USYCSubVault.sol#L305

Impacts
Using **i++** instead of **++i** costs the contract deployment around 600 more gas units.

Remediation
It is recommended to switch to **++i** and change the code accordingly so the function logic remains the same and meanwhile saves some gas.

Retest
This vulnerability is fixed.

6. The Disclosure -----------------------

The Reports provided by CredShields are 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.

YOUR SECURE FUTURE STARTS HERE

CRED SHiELDS

At CredShields, we're more than just auditors. We're your strategic partner in ensuring a secure Web3 future. Our commitment to your success extends beyond the report, offering ongoing support and guidance to protect your digital assets

Audited by CRED SHiELDS