QuillAudits

AUDIT REPORT

July 2025

For

Zoth

Table of Content

Executive Summary

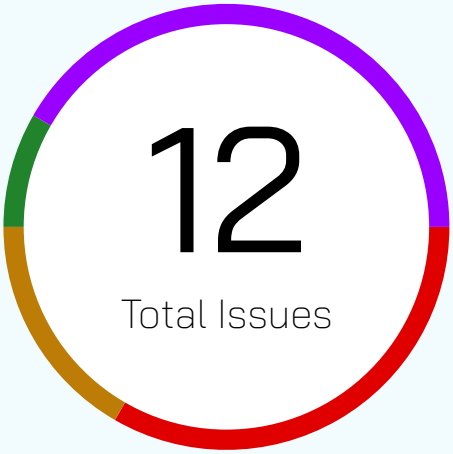
Project name	Zoth
Project URL	https://zoth.io/
Overview	ZeUSD Contracts - A comprehensive DeFi architecture for minting ZeUSD stable token against various collateral types with cross-chain functionality powered by LayerZero. The protocol enables secure, modular, and upgradeable stable token operations with robust access control and emergency safeguards.Features Multi-Collateral Support: Mint ZeUSD using various collateral types like ZTLN Prime and USYCCross-Chain Functionality: Bridge tokens seamlessly across chains via LayerZero integrationModular Architecture: Specialized subvaults for different collateral typesUpgradeability: UUPS proxy pattern for future protocol improvementsRole-Based Access: Granular permissions and whitelist/blacklist functionalityEmergency Controls: Comprehensive pause mechanisms and emergency modes
Audit Scope	The scope of this Audit was to analyze the Zoth Smart Contracts for quality, security, and correctness.
Source Code link	https://github.com/0xZothio/zeusd-contracts

Contracts in Scope

contracts/interfaces/helpers/IOracle.sol
contracts/interfaces/helpers/ITeller.sol
contracts/interfaces/events/IVaultRegistryEvents.sol
contracts/interfaces/events/IWithdrawalSystemEvents.sol
contracts/interfaces/events/IZeDPEvents.sol
contracts/interfaces/events/IVaultEvents.sol
contracts/interfaces/events/IZeUSDEvents.sol
contracts/interfaces/events/IZeUSDRouterEvents.sol
contracts/interfaces/events/IBaseVaultEvents.sol
contracts/interfaces/events/ISystemEvents.sol
contracts/interfaces/IZeUSD.sol
contracts/interfaces/IVaultRegistry.sol
contracts/interfaces/IWithdrawalSystem.sol
contracts/interfaces/IZeUSDRouterV2.sol
contracts/interfaces/IZeDP.sol
contracts/VaultRegistry.sol
contracts/WithdrawalSystem.sol
contracts/interfaces/priceOracles/IPriceOracle.sol
contracts/PriceOracle.sol
contracts/ZeUSDRouterV2.sol
contracts/implementations/ZeUSD.sol
contracts/libraries/WithdrawalSystemTypes.sol
contracts/libraries/DataTypes.sol
contracts/libraries/SystemRoles.sol
contracts/interfaces/errors/IZeUSDErrors.sol
contracts/interfaces/access/IRegistry.sol
contracts/interfaces/vaults/IVault.sol
contracts/interfaces/errors/IZeDPErrors.sol
contracts/interfaces/vaults/IBaseVault.sol
contracts/vaults/BaseVault.sol
contracts/implementations/ZeUSD_CDP.sol
contracts/core/Registry.sol
contracts/utils/ProxyAdmin.sol
contracts/core/AccessManager.sol
contracts/vaults/USYCVault.sol
contracts/interfaces/errors/IBaseVaultErrors.sol
contracts/utils/AccessManager.sol
contracts/interfaces/errors/ISystemErrors.sol
contracts/utils/Create3Factory.sol
contracts/utils/TransparentUpgradeableProxy.sol
contracts/interfaces/errors/IVaultRegistryErrors.sol
contracts/interfaces/errors/IWithdrawalSystemErrors.sol
contracts/interfaces/errors/IZeUSDRouterErrors.sol
contracts/utils/Constants.sol

Branch	Main
Commit Hash	0b2693eca37895047515732ec864c54ac6b023fb
Language	Solidity
Blockchain	Ethereum
Method	Manual Analysis, Functional Testing, Automated Testing
Review 1	10th June 2025 - 27th June 2025
Updated Code Received	1st July 2025
Review 2	2nd July 2025 - 6th July 2025
Fixed In	7e59fc69e6b1cfc3365ff6b6de91c3ab198539dc
Note	We strongly recommend that the Zoth team should conduct thorough testing and fuzzing before deploying the contract on the mainnet with good coverage, as the current test coverage is very low. At the end of the audit, the Zoth team implemented changes to the codebase that have not been reviewed by QuillAudits. Below are the specific changes that have not been reviewed by the QuillAudits team:- change in the WithdrawalSystem contract due to business constraints. modified logic to sell only the amount of USYC equivalent to the user's original USDC deposit at the time of request.

Number of Issues per Severity



High	4 (33.33%)
Medium	2 (16.67%)
Low	1 (8.33%)
Informational	5 (41.67%)

Severity

Issues	High	Medium	Low	Informational
Open	0	0	0	0
Resolved	**4**	**2**	**1**	**4**
Acknowledged	0	0	0	**1**
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

Precision–loss in pricing & mint calculations causes excessive slippage and under–minting

Resolved

Path

contracts/vaults/USYCVault.sol#L231 & contracts/VaultRegistry.sol#L335

Function

deposit()

Description

The router entry–point deposit() contracts/ZeUSDRouterV2.sol# L261 lets a whitelisted user deposit a secondary collateral asset (e.g., USDC) in order to

- purchase the primary asset from a Teller contract, and
- immediately borrow an LTV adjusted amount of zeUSD against that primary asset through the vault.

To protect the user from adverse price movements, deposit() tries to estimate a minimum amount of primary asset i.e minExpectedOutput the user should receive, based on oracle prices and a vault provided slippage tolerance, before forwarding the call to the Teller contract.

In current logic, Oracle price is silently truncated from 8 to 6 decimals In contracts/PriceOracle.sol# L214:

```
return (uint256(answer) / 100, true);
```

The Chainlink answer (8–decimals) is divided by 100, coercing it to 6–decimals and discarding the final two decimal digits.
Again at contracts/vaults/USYCVault.sol# L230–231 Early division compounds the error

```
uint256 oracleBasedOutput = (amount * assetPrice) / primaryPrice;   // 6-dec
minExpectedOutput = (oracleBasedOutput * (10000 - base.maxSlippageBps)) /
10000;
```

Because the oracle price has already lost two decimals, the oracleBasedOutput calculation is biased low. The slippage buffer is then applied to an already deflated number, further widening the gap between expectation and reality.

Similarly early truncation infects zeUSD mint logic at contracts/VaultRegistry.sol# L335–347 performs:

```
collateralValue := div(collateralValue, 1_000_000)   // normalize to 6-dec
mintAmount      := div(mul(collateralValue, ltv), 1_000_000)
```

The 8–dec collateral valuation is divided by 1e6, ensuring the borrower mints less zeUSD than economically warranted.

Consequently, Users may receive lesser primary tokens than market prices, During minting, users obtain less zeUSD, directly reducing their borrowing power. Correspondingly the vault's multisig receives fewer primary assets per deposit, creating an accounting mismatch between liabilities (zeUSD) and assets on balance–sheet. Long–term, the discrepancy can accumulate into a reserve deficit, harming solvency metrics and user confidence.

Likelihood of this is very high because of the following

- The truncation is deterministic and affects all oracle reads flowing through PriceOracle.
- Everyday users often deposit fractional token amounts (e.g., 123.456 USDC). Any non–integer amount magnifies rounding errors because two entire decimal digits are thrown away, not merely rounded.

Given that deposits and borrow operations are routine, the vulnerability will be hit repeatedly, causing math issue (even if passive) almost certain.

Recommendation

- Maintain consistent decimal precision end–to–end. Keep prices in 8–decimals internally. To achieve
- that remove division by 100 at contracts/PriceOracle.sol#L214.
  Calculate minExpectedOutput using below styled math i.e delay division until the very end

```
minExpectedOutput = (amount * assetPrice * (10000 - base.maxSlippageBps)) /
10000 * primaryPrice;
```

- Similar maths can be applied during calculation of the mint amount, re-written the function has below

```
function _calculateScaledMintAmount(uint256 amount, uint256 price, uint256 ltv)
        internal
        pure
        returns (uint256 mintAmount)
    {
        // Mint amount is calculated as:
        // Mint amount = Collateral Value * LTV / 1e6
        // where Collateral Value = amount * price / 1e6
        // We scale the mint amount to 18 decimals for ZeUSD
        // So we multiply by 1e12 (18 - 6 = 12) = scalingFactor
        // Scaled Mint Amount = Mint Amount * scalingFactor
        return (amount * price) * ltv;
    }
```

Specific Fixed In Commit

https://github.com/0xZothio/zeusd-contracts/pull/44

Duplicate secondary-asset transfer in withdrawal flow causes double-Spends and vault insolvency

Resolved

Path

contracts/WithdrawalSystem.sol#L508 & contracts/WithdrawalSystem.sol#l255

Function

processRequest() & processBatchRequest()

Description

processRequest() and processBatchRequest() in WithdrawalSystem.sol let an authorized executor dequeue one or more pending withdrawals. The expected flow for each withdrawal is:

- Sell the position's primary asset via the Teller contract.
- Receive the resulting secondary asset (e.g., USDC) into the vault (USYCVault).
- Transfer the secondary asset from the vault to the withdrawing user.

The helper _handleSecondaryAssetWithdraw() in contracts/vaults/USYCVault.sol#L367 is designed to perform step 3 atomically for the vault.

After _handleSecondaryAssetWithdraw() has already sent the secondary asset to the user, the outer withdrawal logic transfers the same amount a second time at _processBatchGroup() at contracts/WithdrawalSystem.sol#415 and processRequest() at contracts/WithdrawalSystem.sol#525. Both sites assume the vault still holds the freshly acquired secondary asset and execute an additional safeTransferFrom to the user.

Because the vault's balance was already debited in _handleSecondaryAssetWithdraw(), the second transfer draws again from the vault's reserves, effectively creating a double-spend for every processed withdrawal.

Consequently,

- **Users are over-paid**: Each withdrawal receives twice the correct secondary asset amount.
- **Vault insolvency**: The vault hemorrhages funds, quickly moving its liabilities (outstanding zeUSD) far above its remaining collateral.
- **Withdrawal gridlock**: As soon as the vault runs low, later withdrawals revert for insufficient balance, trapping user funds indefinitely.
- **Protocol death spiral**: Insolvency undermines peg confidence, incentivizing bank-run behavior and potential bad-debt socialization.

The likelihood is extremely High — the faulty transfer is present in every execution path of processRequest() and processBatchRequest(). Thus every withdrawal processed will trigger the bug, guaranteeing rapid depletion of vault reserves once the feature is live

Recommendation

- Remove the redundant transfer in WithdrawalSystem (both single and batch branches). Rely solely on _handleSecondaryAssetWithdraw() to remit funds.
- Add balance-change invariants after each withdrawal: vault balance before – after == expected amount.
- Unit-test a withdrawal that inspects final vault balance and user payout to catch double-spend regressions.

Specific Fixed In Commit

0f1622c2c3abdc0445a3a6cf99629198e5b3148c

Incorrect collateralization assumptions leads incorrect nature of zeUSD.

Resolved

Path

contracts/VaultRegistry.sol#319

Function

validateAndPrepareDeposit()

Description

The validateAndPrepareDeposit() function is responsible for validating a deposit and computing how much zeUSD should be minted for a given user. It does this by calling the internal _calculateMintAmount() function, which computes the mint amount based on the provided collateral's value, price, and the applicable loan-to-value (LTV) ratio.

In the intended system design, users should:

- Deposit a secondary asset (e.g., USDC),
- Use that asset to purchase a primary asset (e.g., USYC or another RWA-backed token) through the
- Teller contract,
  Then use the acquired primary asset as collateral to borrow zeUSD.

This design ensures that zeUSD is over-collateralized and backed by real-world asset exposure (via the primary asset).

Currently, the logic in validateAndPrepareDeposit() computes the zeUSD mint amount directly using the secondary asset (e.g., USDC) as collateral:

```
mintAmount = _calculateMintAmount(secondaryAssetAmount, secondaryAssetPrice, LTV);
```

This statement assumes secondary assets as the collateral instead of primary asset. As a result:

- zeUSD is minted against USDC, not the RWA-backed primary asset.
- The user effectively gets the benefit of two assets from a single input i.e zeUSD from minting and the primary asset via the Teller.

This fundamentally violates the system's over-collateralization design and decouples zeUSD's backing from real-world assets, contradicting its intended economic model.

This bug will be triggered on every deposit, as it is baked into the default logic of validateAndPrepareDeposit(). Unless explicitly guarded or redesigned, all minted zeUSD will be based on secondary assets, leading to systemic degradation of collateral assumptions over time.

Recommendation

We recommend to move contracts/VaultRegistry.sol#L335-336 after contracts/VaultRegistry.sol#L393 and call _calculateMintAmount() function with primary asset amount and price as the param instead of the secondary asset.

```
mintAmount = _calculateMintAmount(primaryAssetAmount, primaryAssetPrice, LTV);
```

Specific Fixed In Commit

a7f72c5810f13f487205296cf45fe29ff581a49b

rebalanceZeUSD() lets users self-declare mint amount, enabling unlimited zeUSD creation.

Resolved

Path

contracts/ZeUSDRouterV2.sol#L331

Function

rebalanceZeUSD()

Description

The rebalanceZeUSD() function is intended to help users migrate an existing CDP (Collateralized Debt Position)—and any legacy zeUSD they hold—to the new zeUSD system. The flow is:

- Repay / burn the user's old zeUSD.
- Mint a new zeUSD position and issue a fresh CDP NFT that tracks the debt in the upgraded vault.
- Return the new NFT to the user, completing migration.

To coordinate step 2, the caller supplies a DataTypes.DepositMetadata struct that, among other fields, declares how many new zeUSD tokens should be minted.

rebalanceZeUSD() trusts the caller-supplied DepositMetadata at face value. There is no verification that the zeUsdtomint (or equivalent field) in the struct corresponds to the amount of old zeUSD actually burned. Because the struct is passed directly from the user and is not cross-checked, any caller can specify an arbitrarily large mint amount. The function dutifully mints that quantity of new zeUSD and issues an NFT reflecting the debt.

Consequently,

- **Infinite mint exploit:** Attackers can create unbacked zeUSD at will, extracting real economic value by swapping it on secondary markets.
- **Immediate de-peg:** Oversupply of uncollateralized zeUSD will erode market confidence, driving its price below $1 and breaking integrations.
- **Systemic insolvency:** Vault accounting will show massive negative equity, jeopardizing every legitimate depositor and borrower.
- **Cascade failure:** Protocol governance, oracles, and insurance mechanisms may all be overwhelmed, leading to shutdown or hard fork.

The bug is deterministic and can be triggered by any user who calls rebalanceZeUSD() with a maliciously crafted DepositMetadata so the likelihood is very high.

Recommendation

We recommend minting the same amount of total zeUSD old balance or verify the provided metadata in merkle root verification

Specific Fixed In Commit

https://github.com/0xZothio/zeusd-contracts/pull/45

Medium Severity Issues

Borrow position NFT not burned after withdrawal, causing permanent token lockup

Resolved

Path

contracts/WithdrawalSystem.sol#L508 & contracts/WithdrawalSystem.sol#L384

Function

processRequest() & _processBatchGroup()

Description

In the designed withdrawal flow, a user initiates a withdrawal request by referencing their borrow position, which is represented by a unique tokenId (an NFT). As part of the request submission:
The position NFT is transferred from the user to the WithdrawalSystem contract to indicate that the user has relinquished their position and triggered the withdrawal process.

Once the withdrawal is executed using either processRequest() or _processBatchGroup(), the system:
• Sells the primary asset on behalf of the position via the Teller contract,
• Returns the corresponding secondary asset (e.g., USDC) to the user,
• And considers the withdrawal process complete.

Despite completing the withdrawal and rendering the borrow position economically obsolete, the tokenId (i.e., the borrow position NFT) is never burned from the WithdrawalSystem contract.
Instead, the NFT remains held indefinitely by the contract, with no pathway to retrieve or remove it, resulting in a zombie NFT that no longer represents a valid or usable borrow position.

Consequently, It leads to poor UX and protocol hygiene, Users may expect their NFT to be returned or burned upon completion of the withdrawal, and the lack of clear feedback creates confusion and loss of trust. If future contract upgrades assume that NFTs represent active positions, the presence of stale, burned-but-not-burned NFTs could lead to incorrect behavior in other parts of the protocol.

The likelihood of this behavior will occur for every withdrawal, as the NFT is always transferred in but never burned.

Recommendation

Burn the position NFT once the withdrawal is fully processed to reflect that the position has been closed. This can be done inside processRequest() or _processBatchGroup() via a call to the position NFT contract's burn() method.

Specific Fixed In Commit

9352e59fa3d979483f94595f545ae254c76aa456

processRequest() fails to transfer required primary asset, rendering withdrawals inoperable

Resolved

Path

contracts/WithdrawalSystem.sol#L508

Function

processRequest()

Description

The processRequest() function is responsible for finalizing withdrawal requests for a given position identified by its tokenId. As part of the withdrawal process, primary asset sold via the Teller contract's sellFor() function. The proceeds, in the form of the user's secondary asset (e.g., USDC), are then returned to the position owner.

To execute successfully, the contract (WithdrawalSystem) handling processRequest() must hold the correct amount of primary asset beforehand so it can be passed to the Teller for liquidation.

The processRequest() implementation omits the transfer of the primary asset to the WithdrawalSystem contract before attempting to call the Teller's sellFor() method. Since sellFor() requires the contract to already hold the primary asset it intends to sell, this omission guarantees that the Teller will revert due to insufficient asset balance leads to the entire processRequest() execution to fail.

The failure is deterministic. Without a call to transferFrom() of the primary asset into the withdrawal system contract prior to invoking sellFor(), every call to processRequest() will revert at runtime. However it can be solved by transferring the primary asset first to the contract in first transaction and then execute processRequest() in another transaction, But this process will not give the atomicity to the execution and increase the attack the surface.

Recommendation

We recommend to ensure that processRequest() includes logic to transfer the required primary asset into the vault contract before calling sellFor().

Specific Fixed In Commit

244398fc581182cd29cbf011f847bc227986466c

Low Severity Issues

Missing initialization of ReentrancyGuardUpgradeable

Resolved

Path

contracts/VaultRegistry.sol#L140 , contracts/implementations/ZeUSD_CDP.sol#L142

Function

initialize()

Description

The contract uses ReentrancyGuardUpgradeable from OpenZeppelin to protect critical functions against reentrancy attacks. This pattern is typically used in upgradeable contracts to ensure that once a nonReentrant function is entered, it cannot be called again (e.g., via external call before the first invocation completes), thereby preventing potential exploitation such as draining tokens, altering state unexpectedly, or bypassing access logic.

To activate this protection in upgradeable contracts, the __ReentrancyGuard_init() function must be explicitly called during the initialization phase of the contract usually in the initialize().

The contract inherits from ReentrancyGuardUpgradeable, but does not call __ReentrancyGuard_init() in its initializer. As a result, the internal _status variable used to track entry into nonReentrant functions remains uninitialized, meaning nonReentrant modifiers silently do not enforce the reentrancy lock, and reentrancy protections are non-functional, even though they are declared in the codebase. This is especially dangerous in functions involving external calls (e.g., token transfers, low-level calls to other contracts), which are common targets for reentrancy attacks.

Consequently, any function marked nonReentrant will not actually prevent reentrancy, giving a false sense of security to developers and auditors amd increase the attacke surface and lead potential leaks. The likelihood of this issue is high — Without calling __ReentrancyGuard_init(), every nonReentrant modifier in the contract is non-functional.

Recommendation

Call __ReentrancyGuard_init() in the initializer function of the VaultRegistry and ZeUSD_CDP contract, ideally alongside other inherited initializers.

Specific Fixed In Commit

f35d28e3a494848c5564b6c39ce1218f98743a75

Informational Severity Issues

Inconsistency may be introduced in slippage value during vault registration.

Resolved

Path

contracts/VaultRegistry.sol#L175

Function

registerVault

Description

The registerVault() function in the VaultRegistry contract is used to register a new vault along with its configuration, including a slippage value. This slippage value is meant to define the acceptable bounds for price execution when minting or withdrawing assets through the vault.

Each vault contract also maintains its own internal slippage setting, which governs how the vault itself processes transactions.

The registerVault() function accepts slippage as an explicit external parameter, rather than reading it directly from the vault contract being registered. This creates a risk of slippage mismatches between the slippage value stored in the VaultRegistry, and The actual slippage logic implemented within the vault contract.

If the provided value during registration does not match the vault's internal configuration, it can result in incorrect assumptions about how the vault will behave during execution, Misleading slippage expectations for downstream contracts or users, And potential failures or unintended outcomes in operations relying on consistent slippage enforcement.

Specific Fixed In Commit

https://github.com/0xZothio/zeusd-contracts/pull/38

Updating the contract address is not needed

Resolved

Path

contracts/core/Registry.sol#L175

Function

updateContract

Description

Because each contract in the protocol is upgradeable, its address stays constant while the underlying logic can change. This means the address-update logic at contracts/core/Registry.sol#L175 is unnecessary and should be removed. Instead, add an unregisterContract() function so the registry can cleanly remove a contract entry, providing a proper counterpart to registerContract().

Unnecessary inheritance of OwnableUpgradeable

Resolved

Path

contracts/implementations/ZeUSD.sol#L64

Description

OwnableUpgradeable contract did get used in the ZeUSD contract so it is better to remove it for reducing the attack surface.

Specific Fixed In Commit

d89303f1f33d3b3ddbd6e419eaedac483d9a9907

Incorrect revert message leads confusion

Resolved

Path

contracts/vaults/BaseVault.sol#L277 & contracts/vaults/BaseVault.sol#L303

Function

disableEmergencyMode() & withdrawEmergency()

Description

Fix the revert message by replacing it with Vault_EmergencyModeAlreadyInActive to reflect the correct cause of reverting the transaction at both places i.e contracts/vaults/BaseVault.sol#L277 & contracts/vaults/BaseVault.sol#L303

Specific Fixed In Commit

9a97067b344711a60c9d76a0c6c2de64aba03319

Inconsistency in code style makes code unreadable

Acknowledged

Path

contracts/ZeUSDRouterV2.sol

Description

In ZeUSDRouterV2 contract, Hypernative integration logic has been added which has altogether different code practices like it uses require statement in conjunction with error strings while through out the codebase if statement in conjunction with revert statements get used, Similarly there are differences in style and hypernative integration code is not require for core business logic of router contract consequently having the hypernative integration code within the same contract makes contract less readable.
We recommend having a separate abstract contract which consists all the required functionality of hypernative integration and introduce initialize function within it to initialize the prams and make ZeUSDRouterV2 inherit that abstract contract and call its initialize function within the initialize of ZeUSDRouterV2 contract. This will makes code clean and readable

Speci ic Fixed In Commit

We'll keep this integration as it is as this code is for and from hypernative.

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 Zoth. We performed our audit according to the procedure described above.

Issues of High,Medium,Low and Informational severity were found. Zoth team acknowledged one of the issues and resolved the rest of them

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 pro-gram 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

QuillAudits

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

Follow Our Journey

AUDIT REPORT

July 2025

For

Zoth

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com audits@quillaudits.com