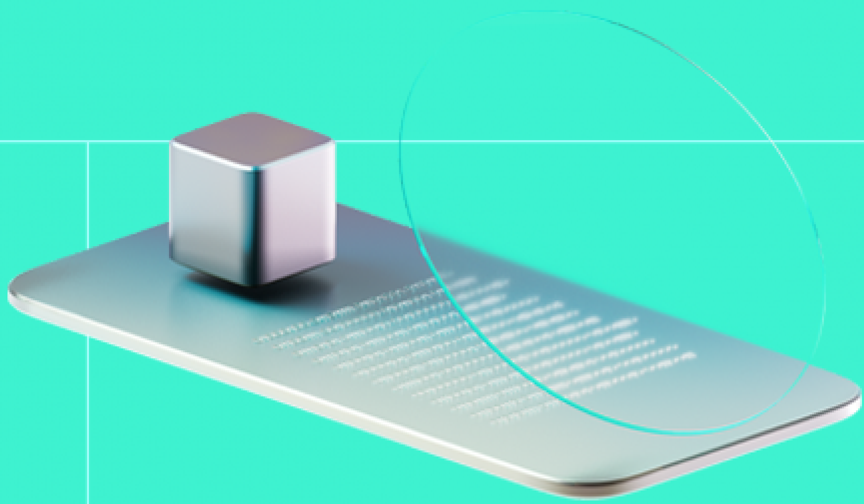
Smart Contract Code Review And Security Analysis Report



Customer: Zoth

Date: 02/01/2025

We express our gratitude to the Zoth team for the collaborative engagement that enabled the execution of this Smart Contract Security Assessment.

Zoth is a retail-focused RWA ecosystem that is bringing fixed-yield generating, institutional grade, high-quality RWAs on-chain. Zoth allows users to collateralize their off-chain or on-chain tokenized assets (TBILLs, ETFs, MMFs etc.) to issue ZeUSD, a yield bearing stable token, which can unlock DeFi access on top of existing RWAs.

Document

Name	Smart Contract Code Review and Security Analysis Report for Zoth
Audited By	Kornel Światłowski, Nataliia Balashova
Approved By	Grzegorz Trawinski
Website	https://zoth.io
Changelog	30/12/2024 - Preliminary Report; 02/01/2025 - Final Report
Platform	Ethereum
Language	Solidity
Tags	Token Sales, Upgradable
Methodology	https://hackenio.cc/sc_methodology

Review Scope

Repository	https://github.com/0xZothio/zeusd-contracts
Commit	a0842125be71adf3784ba91b45fd3476a2ceb7d9
Remediation commit	75a10af095e97785f0a3ed410f91cf02432e171c

Audit Summary

The system users should acknowledge all the risks summed up in the risks section of the report

10	4	6	0
Total Findings	Resolved	Accepted	Mitigated

Findings by Severity

Severity	Count
Critical	1
High	0
Medium	5
Low	4

Vulnerability	Severity
F-2024-7804 - Missing Validation of Collateral Amount Allows User to Drain SubVault From All Tokens	Critical
F-2024-7877 - ADMIN_ROLE Can Front-Run Mint Functions	Medium
F-2024-7992 - ADMIN_ROLE Can Block Burn Operations and Withdraw Assets	Medium
F-2024-8000 - The withdrawEmergency() Allows Withdrawal of Active Collateral Deposits	Medium
F-2024-8013 - Inconsistent Handling of StableCoins and Non-StableCoins in Mint Functions	Medium
F-2024-8040 - Unrestricted Deposit Removal by ADMIN_ROLE in CollateralVault	Medium
F-2024-7820 - Oracle Price Decimal Format Not Validated in setAssetOracle	Low
F-2024-7858 - Missing Revert Allows Collateral Deposit Without Minting ZeUSD	Low
F-2024-8004 - Undefined Decimal Precision for LTV and Price Fields in CollateralDetails	Low
F-2024-8042 - The removeDeposit() Function Reverts or Emits Incorrect Events	Low

Documentation quality

- Functional requirements are present, but only at a high-level.
- Technical description has some gaps.

Code quality

- Best practice violations.
- Unused functions.
- Insufficient Gas modeling.

Test coverage

Code coverage of the project is **17.91%** (branch coverage).

- Negative cases coverage is missed.
- Interactions by several users are not tested thoroughly.
- ZeUSD_Router and USD0PPSubVault contracts are not tested at all.

Table of Contents

System Overview

The **ZeUSD_Router** serves as the central interface for interacting with the ZeUSD protocol. It enables users to mint ZeUSD by depositing collateral or stablecoins, bridge ZeUSD across different blockchains, and burn ZeUSD to retrieve their collateral. Only the whitelisted addresses can mint and burn ZeUSD tokens. The **ZeUSD_Router** interacts with both the **CollateralVault** and **SubVaults** to facilitate these functions.

The **CollateralVault** manages the registry of SubVaults and their collateral configurations, coordinates price data for ZeUSD minting, and tracks user deposits. Only the **ZeUSD_Router** contract can invoke deposit-related functions within the CollateralVault.

The **USD0PPSubVault** contract stores USD0++ tokens used for minting ZeUSD. It interacts with the price oracle to retrieve the current token price. Although the contract inherits from the **ISubVault** interface, it does not support adding additional assets, and related functions are disabled.

The system consists of a single **ZeUSD_Router**, one **CollateralVault**, and multiple **SubVaults**, each handling specific collateral types within the protocol.

Privileged roles

The **ZeUSD_Router** contract uses the **AccessControlUpgradeable** library from OpenZeppelin to restrict access to important functions.

`ADMIN_ROLE` can:

- Sets whitelist status for an account
- Sets multiple whitelist statuses
- Sets global deposit pause status
- Updates LayerZero adapter
- Sets and rests approval for LayerZero adapter

`DEFAULT_ADMIN_ROLE` can:

- Grants and revokes all roles
- Authorizes contract upgrades
- Can perform all admin functions

The **CollateralVault** contract uses the **AccessControlUpgradeable** library from OpenZeppelin to restrict access to important functions.

`ADMIN_ROLE` can:

- Sets the router address
- Registers or updates a subvault configuration
- Updates specific parameters of a subvault
- Removes a subvault registration
- Pauses vault operations
- Unpauses vault operations

`DEFAULT_ADMIN_ROLE` can:

- Grants and revokes all roles
- Authorizes contract upgrades
- Can perform all admin functions

The **USD0PPSubVault** contract uses the **AccessControl** library from OpenZeppelin to restrict access to important functions.

`ADMIN_ROLE` can:

- Sets oracle for an asset
- Adds support for a secondary asset
- Removes support for a secondary asset
- Enables emergency mode
- Disables emergency mode
- Executes emergency withdrawal
- Pauses vault operations
- Unpauses vault operations

Potential Risks

- **Scope Definition and Security Guarantees:** The audit does not cover all code in the repository. Contracts outside the audit scope may introduce vulnerabilities, potentially impacting the overall security due to the interconnected nature of smart contracts.
- **System Reliance on External Contracts:** The functioning of the system significantly relies on specific external contracts. Any flaws or vulnerabilities in these contracts adversely affect the audited project, potentially leading to security breaches or loss of funds.
- **Arbitrary Oracle Address Setting by Admin:** Allowing the admin to set oracle addresses without constraints or verification mechanisms introduces the risk of incorrect or malicious oracle selection, affecting the accuracy of data and potentially leading to financial losses.
- **Owner's Unrestricted State Modification:** The absence of restrictions on state variable modifications by the owner leads to arbitrary changes, affecting contract integrity and user trust, especially during critical operations like minting phases.
- **Flexibility and Risk in Contract Upgrades:** The project's contracts are upgradable, allowing the administrator to update the contract logic at any time. While this provides flexibility in addressing issues and evolving the project, it also introduces risks if upgrade processes are not properly managed or secured, potentially allowing for unauthorized changes that could compromise the project's integrity and security.
- **Absence of Upgrade Window Constraints:** The contract suite allows for immediate upgrades without a mandatory review or waiting period, increasing the risk of rapid deployment of malicious or flawed code, potentially compromising the system's integrity and user assets.
- The `USD0PPSubVault` contract assumes the ERC-20 token in the `USD0PP` variable is a stablecoin with a Chainlink price oracle using a 24-hour heartbeat. The `getOraclePrice()` function enforces this assumption with a hardcoded 24-hour check. If a token with a different heartbeat interval is assigned during contract construction, stale price data may be returned, compromising the accuracy and reliability of the contract's price data.

Findings

Vulnerability Details

F-2024-7804 - Missing Validation of Collateral Amount Allows User to Drain SubVault From All Tokens - Critical

Description:

Users can mint `ZeUSD` tokens using supported tokens as collateral. The collateral is transfered to a dedicated `SubVault`. Later, the user can burn `ZeUSD` tokens using the `burn()` function and receive the collateral tokens in return.

The `burn()` and `CollateralVault::deactivateDeposit()` functions do not validate whether:

- The provided `collateralAmount` by the user matches the amount initially used as collateral.
- The provided `zeUSDAmount` by the user matches the initial mint amount.

This lack of validation for `collateralAmount` allows users to drain all collateral from the dedicated `SubVault`. The user can provide the `SubVault`'s collateral token balance and burn only a portion of the `ZeUSD` tokens.

Additionally, the missing check for `zeUSDAmount` could allow partial burning of tokens. A user could burn only a portion of the `ZeUSD` tokens, receive only a part of the collateral, and be unable to retrieve the remaining collateral, effectively locking the collateral tokens in the dedicated `SubVault` contract. Locked tokens can be withdrawn however by the `ADMIN_ROLE` with the `withdrawEmergency()` function

```
// ZeUSD_Router
function burn(address asset, uint256 collateralAmount, uint256 zeUSDAmount, uint256 depositId)
 {
    address subvault = collateralVault.getSubVaultAddress(asset);
    if (subvault == address(0)) revert AssetNotSupported(asset);
    if (!ISubVault(subvault).isPrimaryAsset(asset)) revert AssetNotSupported(asset);

    if (zeusdToken.balanceOf(msg.sender) < zeUSDAmount) revert InsufficientBalance('ZeUSD');

    zeusdToken.burnFrom(msg.sender, zeUSDAmount);

    try ISubVault(subvault).handleWithdraw(msg.sender, asset, collateralAmount)
        returns(bool result) {
        if (!result) revert WithdrawFailed('SubVault operation failed');
        (bool success, ) = collateralVault.deactivateDeposit(msg.sender, depositId);
        if (!success) revert WithdrawFailed('Failed to deactivate deposit');
        emit Burned(msg.sender, asset, zeUSDAmount, subvault);
    }
    catch Error(string memory reason) {
```

```
            revert WithdrawFailed(reason);
        }
    }
```

```
    // SubVault
    function handleWithdraw(address user, address asset,
     uint256 amount) {
        if (amount == 0) revert InvalidAmount();
        if (emergencyMode) revert EmergencyModeEnabled(block.timestamp);

        // Only allow withdrawals of primary asset
        if (asset != USD0PP) revert UnsupportedAsset(asset);

        // Transfer primary asset directly
        IERC20(USD0PP).safeTransfer(user, amount);
        emit PrimaryAssetOperation(user, amount, false);
        return true;
    }
```

```
    // CollateralVault
    function deactivateDeposit(address user, uint256 depositId) {
        if (user == address(0)) revert InvalidAddress(user);

        // Get user's deposits
        DataTypes.UserDeposit[] storage userDepositsList = userDeposits[user];
        bool found = false;
        uint256 depositIndex;

        // Find the specific deposit
        for (uint256 i = 0; i < userDepositsList.length; i++) {
            if (userDepositsList[i].depositId == depositId) {
                depositIndex = i;
                found = true;
                break;
            }
        }

        if (!found) revert DepositNotFound(depositId);

        // Get deposit reference
        DataTypes.UserDeposit storage deposit = userDepositsList[depositIndex];

        // Check if deposit is already inactive
        if (!deposit.active) revert DepositNotActive();

        // Store mint amount before deactivating
        mintAmount = deposit.zeusdMinted;

        // Deactivate deposit
        deposit.active = false;

        emit DepositDeactivated(user, depositId, deposit.asset, deposit.amount, deposit.zeus
    dMinted,
```

```
                                deposit.subVault);

    return (true, mintAmount);
}
```

Assets:

- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Classification

Impact: 5/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Critical

Recommendations

Remediation: It is recommended to remove the `collateralAmount` and `zeUSDAmount` from the `burn()` function. Instead, these values should be retrieved from the deposit information of provided `depositId` in the `CollateralVault::userDepositsList`. All `ZeUSD` tokens should be burned, and the user should receive the initial collateral amount as specified in the deposit records. This will prevent the issues related to incorrect or partial burns and ensure that the correct amount of collateral is returned to the user.

Resolution: The Finding is fixed in the commit `efda4764e68a83e0e73dbcd1c752dc0376f38b37`. `amount` is retrieved from the deposit information of provided `depositId`.

Evidences

PoC

Reproduce:

```
import { expect } from "chai";
import { Signer } from "ethers";
import { ethers, upgrades } from "hardhat";
import {
    CollateralVault,
    MockERC20,
    MockPriceOracle,
    MockProtocol,
```

```
    MockZeUSD_Router,
    MockZTLNtV2,
    ZeUSD,
    USD0PPSubVault
} from '../typechain-types';
import { subVaults } from "../typechain-types/contracts";

describe("Audit", function () {
    // Contract instances
    let router: MockZeUSD_Router;
    let collateralVault: CollateralVault;
    let zeusdToken: ZeUSD;
    let mockSubVault: USD0PPSubVault;
    let mockProtocol: MockProtocol;
    let mockPriceOracle: MockPriceOracle;

    // Signers
    let owner: Signer;
    let admin: Signer;
    let user: Signer;
    let user2: Signer;
    let otherUser: Signer;
    let addresses: {
        router: string;
        collateralVault: string;
        zeusdToken: string;
        mockSubVault: string;
        mockProtocol: string;
        mockPriceOracle: string;
        owner: string;
        admin: string;
        user: string;
        user2: string;
        otherUser: string;
    };

    // Mock ERC20 tokens
    let mockZTLN: MockERC20;
    let mockUSD0PP: MockERC20;
    let mockTokenAddresses: {
        ZTLN: string;
        USD0PP: string;
    };

    // Test constants
    const INITIAL_SUPPLY_ZTLN = ethers.parseUnits("1000000", 18);
    const INITIAL_SUPPLY_USD0PP = ethers.parseUnits("1000000", 6);

    const PRICE = ethers.parseUnits("1", 8); // $1 with 8 decimals
    const LTV = 800000n; // 80% with 6 decimals
    const ZERO_ADDRESS = ethers.ZeroAddress;
    const DUMMY_ADDRESS = "0x1111111111111111111111111111111111111111";

    async function deployContracts() {
```

```
        // Deploy mock tokens
        const MockZTLN = await ethers.getContractFactory("MockZTLNtV2");
        mockZTLN = await MockZTLN.deploy("Mock ZTLN", "ZTLN", 18) as unknown as MockZTLN
tV2;
        const MockUSD0PP = await ethers.getContractFactory("MockERC20");
        mo
```

[See more]

Results:

```
  Audit
    PoC
User1 deposits: 1000000000000
User2 deposits: 1000000
User2 USD0PP initial balance:    1000000000000
subVault USD0PP initial balance: 0
_________________________________________
User2 USD0PP balance before burn:    999999000000
subVault USD0PP before burn:         1000001000000
_________________________________________
User2 USD0PP balance before after:    2000000000000
subVault USD0PP before after:         0
      ✔ Steal tokens from SubVault (60ms)


  1 passing (3s)
```

F-2024-7877 - ADMIN_ROLE Can Front-Run Mint Functions - Medium

Description:

When attempting to mint `ZeUSD` tokens, the mint amount is calculated based on the price of the collateral token, the amount of the collateral token, and the Loan-to-Value (LTV) ratio. The price of the collateral token is retrieved from the dedicated `SubVault`, which fetches it from a price oracle configurable by the `ADMIN_ROLE`. If the oracle is not set, the asset is unsupported, the price is stale, or the call to the price oracle contract reverts, a custom price is used (`1e6` for stablecoins or a configurable price from the asset details). Subsequently, the LTV is applied, and the scaled mint amount is returned.

```
// CollateralVault
function recordDeposit(...) (uint256 depositId, uint256 mintAmount) {
  {...}

  uint256 assetPrice;
  // Calculate mint amount based on collateral details and token type
  if (asset == collateralAddress) {
    // Try to get oracle price first for direct collateral deposit
    (uint256 oraclePrice, bool success) = ISubVault(subVault).getOraclePrice(collateralA
ddress);
    // Use oracle price if available, otherwise fall back to stored price
    assetPrice = success ? oraclePrice / 100 : details.price;
    mintAmount = _calculateMintAmount(amount, assetPrice, details.ltv);
  } else {
    // Other asset deposit (e.g., stablecoins)
    if (!ISubVault(subVault).isAssetSupported(asset)) revert AssetNotSupported(asset);
    (uint256 oraclePrice, bool success) = ISubVault(subVault).getOraclePrice(asset);
    assetPrice = success ? oraclePrice / 100 : STABLE_PRICE * 10 * *6;

    mintAmount = _calculateMintAmount(amount, assetPrice, details.ltv);
  }

  {...}

  return (depositId, scaledMintAmount);
}
```

```
// USD0PPSubVault
function getOraclePrice(address asset) (uint256 price, bool success) {
  if (!supportedAssets[asset]) {
    return (0, false);
  }

  address oracle = assetOracles[asset];
  if (oracle == address(0)) {
    return (0, false);
  }

  try IPriceOracle(oracle).latestRoundData()
      returns(uint80, int256 answer, uint256, uint256 updatedAt, uint80) {
```

```
      // Check if the price is positive
      if (answer <= 0) {
        return (0, false);
      }

      // Check for stale price
      if (block.timestamp - updatedAt > 24 hours) {
        return (0, false);
      }

      return (uint256(answer), true);
    }
    catch {
      return (0, false);
    }
  }
```

The `ADMIN_ROLE` can manipulate parameters to front-run any mint function by submitting a transaction with a higher fee before the execution of the mint function. Several methods can facilitate this:

- `CollateralVault::registerSubVault()` - Changing the LTV parameter.
- `SubVault::setAssetOracle()` - Setting an oracle that provides a lower price than the actual value.
- `SubVault::setAssetOracle()` - Configuring an oracle that reverts on call, causing the fallback price to be used.
- `SubVault::removeAsset()` - Removing support for the asset, causing the fallback price to be used.

These actions can result in mint calculations that provide significantly fewer `ZeUSD` tokens than expected. In the worst-case scenario, only 1 `ZeUSD` token could be minted while consuming collateral worth significantly more, such as 1,000,000 USD or even higher, depending on the manipulated parameters.

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Accepted

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: It is recommended to introduce `expectedZeUSDAmount` and `slippage` function parameters into all mint functions. The `expectedZeUSDAmount` should then be validated against the calculated mint amount returned by the `CollateralVault::recordDeposit()` function, taking into account the slippage tolerance specified by the user.

Resolution: The Finding was accepted with the following statement:

> Thank you for the recommendation. The design currently relies on robust security measures, including the use of a multisig wallet for the `ADMIN_ROLE`, ensuring that privileged actions are executed securely and with consensus.
> Introducing `expectedZeUSDAmount` and slippage parameters into the mint functions is a feature we will evaluate further. While this approach could enhance user protection by validating mint amounts against slippage tolerance, the current system emphasizes multisig governance to secure operations and mitigate risks.

F-2024-7992 - ADMIN_ROLE Can Block Burn Operations and Withdraw Assets - Medium

Description:

`ZeUSD` tokens are minted by providing supported collateral, with the mint amount calculated based on the provided collateral, its current asset value, and the Loan-to-Value (LTV) ratio. The LTV must remain below 100%, ensuring that the value of minted `ZeUSD` tokens is less than the locked collateral. Burn functionality allows `ZeUSD` tokens to be exchanged for the corresponding collateral amount, ensuring that collateral retrieval aligns with the LTV constraint.

Both mint and burn functionalities are restricted to users who are whitelisted in the `ZeUSD_Router`, not blacklisted in the `ZeUSD`, and only when the `ZeUSD_Router` contract is not paused. The `ADMIN_ROLE` has the ability to modify whitelisting and blacklisting statuses arbitrarily and to pause the contract.

```
 modifier whitelistedOnly() {
     if (!_whitelisted[msg.sender]) revert NotWhitelisted(msg.sender);
     _;
}

modifier notBlacklisted() {
     if (zeusdToken.isBlacklisted(msg.sender)) revert Blacklisted(msg.sender);
     _;
}

modifier whenNotPaused() {
     if (depositsPaused) revert DepositsArePaused();
     _;
}

function mintWithCollateral() external override nonReentrant whenNotPaused
     whitelistedOnly notBlacklisted validAmount(amount) {}

function mintWithStable() external override nonReentrant whenNotPaused
     whitelistedOnly notBlacklisted validAmount(amount) {}

function burn() external override nonReentrant whenNotPaused whitelistedOnly
     notBlacklisted validAmount(zeUSDAmount) {}

function mintWithCollateralAndBridge() external payable override nonReentrant
     whenNotPaused whitelistedOnly notBlacklisted validAmount(amount) returns() {
}

function mintWithStableAndBridge() external payable override nonReentrant
     whenNotPaused whitelistedOnly notBlacklisted validAmount(amount) returns() {
}
```

SubVaults, which hold collateral assets, are required to implement the `withdrawEmergency()` function to enable asset recovery in emergency situations. In the current implementation of the `USD0PPSubVault,` the `ADMIN_ROLE` can arbitrarily trigger the emergency state and withdraw all collateral assets. This introduces a potential scenario where users could be prevented from burning `ZeUSD` tokens to retrieve their collateral worth more due to LTV setting, while the `ADMIN_ROLE` can withdraw all tokens, leading to asset locking and potential loss.

```
 function setWhitelistStatus(address account, bool status) external override
     onlyRole(ADMIN_ROLE) {
     if (account == address(0)) revert InvalidAddress(account);
     _whitelisted[account] = status;
     emit WhitelistStatusChanged(account, status);
 }

function setDepositsPaused(bool paused) external override onlyRole(ADMIN_ROLE) {
     depositsPaused = paused;
     emit DepositsStatusChanged(paused);
 }
```

Assets:

- contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Accepted

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: It is recommended to remove the whitelisting, blacklisting, and paused-state modifiers from the `burn()` function to ensure uninterrupted collateral retrieval. As an additional safety measure, the `ADMIN_ROLE` should be assigned to a Multisig wallet, requiring multiple approvals for executing privileged actions, thereby enhancing security and reducing the risk of misuse.

Resolution: The Finding was accepted with the following statement:

> The inclusion of whitelisting, blacklisting, and paused-state modifiers in the `burn()` function is an intentional design choice to ensure the protocol can prevent operations in case of an emergency. These safeguards are critical for mitigating potential risks and responding to unforeseen vulnerabilities or malicious activity.
> We agree with the recommendation to assign the `ADMIN_ROLE` to a Multisig wallet. This will provide an additional layer of security by

requiring multiple approvals for executing privileged actions, aligning with best practices and reducing the likelihood of misuse

F-2024-8000 - The withdrawEmergency() Allows Withdrawal of Active Collateral Deposits - Medium

Description:

ZeUSD tokens are minted by providing supported collateral, with the mint amount calculated based on the collateral amount, its current asset value, and the Loan-to-Value (LTV) ratio. The burn functionality allows ZeUSD tokens to be exchanged for the corresponding collateral amount used during the minting process. All collateral tokens are stored in a dedicated `SubVault` contract.

The `ADMIN_ROLE` in the `USD0PPSubVault` contract has the ability to withdraw all collateral tokens via the `withdrawEmergency()` function. This function requires the contract to be in an emergency state, which can be enabled at any time by the `ADMIN_ROLE`. As a result, the `ADMIN_ROLE` can access collateral associated with active deposits used for minting ZeUSD tokens. If all collateral is withdrawn, users will be unable to retrieve their collateral through the burn functionality, leading to a potential loss of assets.

```
/// @notice Executes emergency withdrawal
function withdrawEmergency(address asset, address to, uint256 amount,
                           string calldata reason) external override nonReentrant
    onlyRole(ADMIN_ROLE) returns(bool) {
    if (!emergencyMode) revert EmergencyModeNotEnabled();
    if (block.timestamp < lastEmergencyAction + EMERGENCY_DELAY) revert EmergencyDelayNo
tPassed();
    if (amount == 0) revert InvalidAmount();
    if (!supportedAssets[asset]) revert UnsupportedAsset(asset);

    uint256 balance = IERC20(asset).balanceOf(address(this));
    uint256 withdrawAmount = amount > balance ? balance : amount;

    if (asset == USD0PP) {
        _revokeApproval(USD0PP, address(USD0PP));
    }

    IERC20(asset).safeTransfer(to, withdrawAmount);

    lastEmergencyAction = block.timestamp;
    emit EmergencyWithdrawalExecuted(asset, to, withdrawAmount, reason);

    return true;
}
```

Assets:

- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Accepted

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: It is recommended to restrict the `withdrawEmergency()` function to allow withdrawal only of surplus tokens and tokens from inactive deposits, such as those resulting from partial burns. A new variable should be introduced to track the total amount of active deposits, ensuring that collateral associated with active deposits remains inaccessible for emergency withdrawal.

Resolution: The Finding was accepted with the following statement:

> This behavior is intentional and designed for emergency scenarios where the ADMIN_ROLE must safeguard or reallocate collateral promptly. The `withdrawEmergency()` function ensures that collateral can be accessed only when the contract is explicitly placed in an emergency state, a measure we deem necessary to mitigate critical risks.

F-2024-8013 - Inconsistent Handling of StableCoins and Non-StableCoins in Mint Functions - Medium

Description:

ZeUSD tokens can be minted by providing supported collateral through various mint functions in the `ZeUSD-Router` contract:

- `mintWithCollateral()`: Mints ZeUSD using a `collateralAddress` deposit.
- `mintWithStable()`: Mints ZeUSD using stablecoins.
- `mintWithCollateralAndBridge()`: Mints and bridges ZeUSD via LayerZero.
- `mintWithStableAndBridge()`: Mints and bridges stablecoins via LayerZero.

When a new `SubVault` is registered in the `CollateralVault` contract, the `ADMIN_ROLE` specifies the token type as either `StableCoin` or `NotStableCoin`. However, no validation exists in the mint functions to ensure the correct usage of the collateral asset type based on this designation.

The `USD0PPSubVault` contract is designed to hold USD0++ tokens, which are stablecoins. Despite this, users can mint ZeUSD using USD0++ tokens through `mintWithCollateral()` and `mintWithCollateralAndBridge()`, which are intended for `NotStableCoin` types. Attempts to mint using `mintWithStable()` result in transaction reversion due to the `USD0PPSubVault::handleDeposit()` function call.

This inconsistency introduces confusion and prevents a clear and predictable flow for minting ZeUSD tokens with stablecoins.

Assets:

- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Accepted

Classification

Impact: 2/5

Likelihood: 5/5

Exploitability: Independent

Complexity: Simple

Severity: Medium

Recommendations

Remediation:

It is recommended to address the issue by either:

Merging and Refactoring Functions: Merge `mintWithCollateral()` and `mintWithStable()` into a unified function. Modify the logic and contract flow to ensure compatibility with both `StableCoin` and `NotStableCoin` types, improving clarity and reducing contract size.

Adding Validation and Separation: Add explicit validation using `require` statements to enforce that stablecoins are used exclusively with `mintWithStable()` and `mintWithStableAndBridge()`. This approach will ensure specific behavior and flow for minting operations based on collateral type while maintaining functional separation.

Resolution: The Finding was accepted with the following statement:

> The described behavior is intentional, as USD0++ tokens are specifically designed to serve as collateral within our system. While USD0++ tokens are stablecoins, they are treated as collateral for minting ZeUSD through the `mintWithCollateral()` and `mintWithCollateralAndBridge()` functions.
> This approach ensures flexibility and aligns with our architecture, where USD0++ is leveraged for its unique properties as a composite stable asset. The transaction reversion for `mintWithStable()` is also intentional, as this function is not meant to handle USD0++ tokens. By design, we classify and handle USD0++ differently to maintain a clear segregation between standard stablecoins and our designated collateral assets.

F-2024-8040 - Unrestricted Deposit Removal by ADMIN_ROLE in CollateralVault - Medium

Description:

ZeUSD tokens can be minted by providing supported collateral, with details about the collateral, amounts, and related data stored in the `CollateralVault`.

The `removeDeposit()` and `removeBulkDeposits()` functions in the `CollateralVault` contract are used to delete deposit records. However, the `ADMIN_ROLE` has the ability to remove any deposit, including active ones. The data stored in the deposit struct is used in the `burn()` mechanism, which facilitates the retrieval of locked collateral by burning ZeUSD tokens. If an active record is removed, the associated collateral cannot be retrieved.

Given the Loan-to-Value (LTV) constraint (LTV < 100%), the locked collateral is always worth more than the minted ZeUSD tokens. This creates a scenario where users may be unable to retrieve collateral that exceeds the value of the acquired ZeUSD tokens.

```
// ZeUSD_Router
function removeDeposit(address user, uint256 depositId) external override onlyRole(ADMIN
_ROLE)
    returns(bool success, uint256 mintAmount) {
    return collateralVault.removeDeposit(user, depositId);
}

function removeBulkDeposits(address user, uint256[] calldata depositIds) external overri
de
    onlyRole(ADMIN_ROLE) returns(bool success, uint256 totalMintAmount) {
    return collateralVault.removeBulkDeposits(user, depositIds);
}
```

```
// CollateralVault
function removeDeposit(address user, uint256 depositId) external onlyRouter whenNotPause
d
    validAddress(user) returns(bool success, uint256 mintAmount) {
    // Get user's deposits
    DataTypes.UserDeposit[] storage userDepositsList = userDeposits[user];
    bool found = false;
    uint256 depositIndex;
    // Find the specific deposit
    for (uint256 i = 0; i < userDepositsList.length; i++) {
        if (userDepositsList[i].depositId == depositId) {
            depositIndex = i;
            found = true;
            break;
        }
    }

    if (!found) revert DepositNotFound(depositId);

    // Store mint amount before removing
```

```
        mintAmount = userDepositsList[depositIndex].zeusdMinted;

        // Remove deposit by swapping with last element and popping
        uint256 lastIndex = userDepositsList.length - 1;
        if (depositIndex != lastIndex) {
            userDepositsList[depositIndex] = userDepositsList[lastIndex];
        }
        userDepositsList.pop();

        emit DepositRemoved(user, depositId, userDepositsList[depositIndex].asset,
                            userDepositsList[depositIndex].amount, mintAmount,
                            userDepositsList[depositIndex].subVault);

        return (true, mintAmount);
    }

    function removeBulkDeposits(address user,
                                uint256[] calldata depositIds) external onlyRouter whenNotPa
used
        validAddress(user) returns(bool success, uint256 totalMintAmount) {
        if (depositIds.length == 0) revert InvalidParameters('Empty depositIds array');

        DataTypes.UserDeposit[] storage userDepositsList = userDeposits[user];
        uint256[] memory indexesToRemove = new uint256[](depositIds.length);
        uint256 validCount = 0;

        // First pass: validate and collect indexes
        for (uint256 i = 0; i < depositIds.length; i++) {
            bool found = false;
            for (uint256 j = 0; j < userDepositsList.length; j++) {
                if (userDepositsList[j].depositId == depositIds[i]) {
                    indexesToRemove[validCount] = j;
                    totalMintAmount += userDepositsList[j].zeusdMinted;
                    validCount++;
                    found = true;
                    break;
                }
            }
            if (!found) revert DepositNotFound(depositIds[i]);
        }

        // Second pass: remove deposits (from highest index to lowest)
        for (uint256 i = validCount; i > 0; i--) {
            uint256 indexToRemove = indexesToRemove[i - 1];
            DataTypes.UserDeposit memory depositToRemove = userDepositsList[indexToRemove];

            // Remove deposit by swapping with last element and popping
            uint256 lastIndex = userDepositsList.length - 1;
            if (indexToRemove != lastIndex) {
                userDepositsList[indexToRemove] = userDepositsList[lastIndex];
            }
            userDepositsList.pop();

            emit DepositRemoved(user, depositToRemove.depositId, depositToRemove.asset,
```

```
                        depositToRemove.amount, depositToRemove.zeusdMinted,
                        depositToRemove.subVault);
        }

        return (true, totalMintAmount);
    }
```

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Accepted

Classification

Impact: 5/5

Likelihood: 3/5

Exploitability: Dependent

Complexity: Simple

Severity: Medium

Recommendations

Remediation: It is recommended to restrict the `removeDeposit()` and `removeBulkDeposits()` functions to prevent the removal of active deposits. Additional validation should ensure that only deposits marked as inactive or with zero collateral amounts can be removed.

Resolution: The Finding was accepted with the following statement:

> The `removeDeposit()` and `removeBulkDeposits()` functions are part of an experimental feature intended to test new SubVault integrations related to RWAs. Ideally, no active deposits should be removed; these functions were included solely to address stale deposits associated with SubVaults that are no longer part of the collateral system.
> In future upgrades, once the architecture stabilizes, these functions will be deprecated and removed to align with the finalized design.

F-2024-7820 - Oracle Price Decimal Format Not Validated in setAssetOracle - Low

Description: The `CollateralVault::recordDeposit()` function calculates the mint amount based on the provided collateral asset and amount. The collateral asset must have a dedicated SubVault and be supported. The asset price is retrieved from the oracle price feed configured by the given SubVault's `ADMIN_ROLE`.

The `CollateralVault` assumes that the price retrieved from the oracle has 8 decimal precision. However, some price oracles, especially those involving ETH as the secondary asset, may return prices in an 18-decimal format. This mismatch can lead to unexpected behavior or incorrect mint amounts.

Assets:

- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Accepted

Classification

Impact: 5/5

Likelihood: 2/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to add a validation in the `setAssetOracle()` function to ensure that the configured oracle price feed provides prices in an 8-decimal format. This validation would prevent misconfigurations and ensure consistent behavior in mint amount calculations.

Resolution: The Finding was accepted with the following statement:

> The 8-decimal precision assumption is intentional, as our platform exclusively supports RWAs, which typically adhere to this standard. Non-standard decimal formats like 18-decimal are not supported by design to ensure simplicity and consistency.

F-2024-7858 - Missing Revert Allows Collateral Deposit Without Minting ZeUSD - Low

Description: The `_calculateMintAmount()` function does not handle scenarios with smaller collateral deposits or low asset prices correctly. When used directly or invoked through the `recordDeposit()` function, the calculation may return `0` instead of reverting. This behavior results in collateral tokens being taken from the user, deposited into the dedicated vault, but no `ZeUSD` tokens being minted.

For example:

Case 1:

- Collateral Price: `1e8` (value returned from the oracle)
- Collateral: ERC20 with 18 decimals
- Collateral Amount: `1e11`
- Result: `_calculateMintAmount()` returns `0`, minting `0` `ZeUSD` for `1e11` collateral tokens.

Case 2:

- Collateral Price: `1e2` (value returned from the oracle)
- Collateral: ERC20 with 18 decimals
- Collateral Amount: `1e18`
- Result: `_calculateMintAmount()` returns `0`, minting `0` `ZeUSD` for `1e18` collateral tokens.

This behavior may cause loss of user collateral without receiving any `ZeUSD` tokens in return.

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Classification

Impact: 3/5

Likelihood: 2/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to modify the `_calculateMintAmount()` function to revert when a result of `0` is returned. This ensures that deposits of collateral with insufficient value to mint `ZeUSD` tokens are not accepted, preventing unintended collateral loss.

Resolution:

The Finding was fixed in commit `b575bff49cb4ce986e1fad045a1c2ca52d69adfb`.

The `_scaleAmount()` function, used in both `calculateMintAmount()` and mint functions, reverts if it returns `0`. This behavior prevents the minting of zero tokens and taking small amounts of collateral from user.

F-2024-8004 - Undefined Decimal Precision for LTV and Price Fields in CollateralDetails - Low

Description: The `CollateralDetails` struct contains `LTV` and `price` variables, but their decimal precision is not explicitly defined. This information is neither documented nor included in the NatSpec comments for the struct. While some provided tests indicate the precision for `LTV`, no information is available regarding the precision for the collateral `price`.

This lack of clarity can lead to confusion and incorrect assumptions during implementation, testing, or integration, potentially resulting in unintended behavior or miscalculations.

```
/// @notice Detailed information about collateral assets
/// @dev Struct ordering optimized for packing into storage slots
/// @param integrationType Type of integration used
/// @param collateralAddress Address of the collateral token
/// @param subVaultAddress Address of the associated subvault
/// @param price Current price of the collateral
/// @param ltv Loan-to-Value ratio for the collateral
/// @param isActive Whether this collateral is currently active
/// @param registeredAt Timestamp of collateral registration
/// @param lastUpdatedAt Timestamp of last update
/// @param tokenType Classification of the collateral token
struct CollateralDetails {
    string integrationType;
    address collateralAddress;
    address subVaultAddress;
    uint256 price;
    uint256 ltv;
    bool isActive;
    uint256 registeredAt;
    uint256 lastUpdatedAt;
    TokenType tokenType;  // Added enum field
}
```

Assets:

- contracts/libraries/DataTypes.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Classification

Impact: 2/5

Likelihood: 3/5

Exploitability: Independent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to explicitly define the decimal precision for both the `LTV` and `price` variables within the `CollateralDetails` struct. This should be documented in the NatSpec comments for the struct and included in documentation. Additionally, tests should verify the correct handling of these precisions to ensure consistency across the contract.

Resolution: The Finding is fixed in the commit `ca89cdac2dcde06dded8c177fe90d33ec38267c1`.

The decimal precision is defined.

F-2024-8042 - The removeDeposit() Function Reverts or Emits Incorrect Events - Low

Description:

The `removeDeposit()` function in the `CollateralVault` contract is designed to remove a deposit associated with a specific user and `depositID`. The function is callable exclusively by the `ZeUSD_Router` contract, which requires the `ADMIN_ROLE` to execute the transaction.

When the `depositID` corresponds to the last element in the array retrieved from the `userDeposits` mapping, the function reverts. This also occurs when the array contains only one element. The reversion happens because the function attempts to access values from the last array element after it has already been deleted.

Additionally, if the `depositID` points to an element in the middle of the array, the event emitted by the function uses values from the swapped element instead of the removed element, leading to incorrect event arguments.

These issues result in inconsistent behavior during deposit removal and could impact the reliability of the system by causing exceptions and emitting misleading event data.

```
function removeDeposit(address user, uint256 depositId) external onlyRouter whenNotPause
d
    validAddress(user) returns(bool success, uint256 mintAmount) {
    // Get user's deposits
    DataTypes.UserDeposit[] storage userDepositsList = userDeposits[user];
    bool found = false;
    uint256 depositIndex;
    // Find the specific deposit
    for (uint256 i = 0; i < userDepositsList.length; i++) {
        if (userDepositsList[i].depositId == depositId) {
            depositIndex = i;
            found = true;
            break;
        }
    }

    if (!found) revert DepositNotFound(depositId);

    // Store mint amount before removing
    mintAmount = userDepositsList[depositIndex].zeusdMinted;

    // Remove deposit by swapping with last element and popping
    uint256 lastIndex = userDepositsList.length - 1;
    if (depositIndex != lastIndex) {
        userDepositsList[depositIndex] = userDepositsList[lastIndex];
    }
    userDepositsList.pop();

    emit DepositRemoved(user, depositId, userDepositsList[depositIndex].asset,
                        userDepositsList[depositIndex].amount, mintAmount,
```

```
                userDepositsList[depositIndex].subVault);

    return (true, mintAmount);
}
```

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Classification

Impact: 2/5

Likelihood: 5/5

Exploitability: Dependent

Complexity: Simple

Severity: Low

Recommendations

Remediation: It is recommended to cache the values of the element to be removed before its deletion. This cached data should then be used for event emission and any subsequent operations. This approach will prevent reversion when handling the last array element and ensure that events use the correct arguments.

Resolution: The Finding was fixed in commit `2037918d748f0417ccdca21f9c035127c751afa7`.

The deleted values are cached and subsequently used during the emission of event.

Observation Details

F-2024-7753 - Floating Pragma - Info

Description:

In Solidity development, the pragma directive specifies the compiler version to be used, ensuring consistent compilation and reducing the risk of issues caused by version changes. However, using a floating pragma (e.g., `^0.8.xx`) introduces uncertainty, as it allows contracts to be compiled with any version within a specified range. This can result in discrepancies between the compiler used in testing and the one used in deployment, increasing the likelihood of vulnerabilities or unexpected behavior due to changes in compiler versions.

The project currently uses floating pragma declarations (`^0.8.20` and `^0.8.24`) in its Solidity contracts. This increases the risk of deploying with a compiler version different from the one tested, potentially reintroducing known bugs from older versions or causing unexpected behavior with newer versions. These inconsistencies could result in security vulnerabilities, system instability, or financial loss. Locking the pragma version to a specific, tested version is essential to prevent these risks and ensure consistent contract behavior.

Assets:

- contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/implementations/ZeUSD.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation:

It is recommended to **lock the pragma version** to the specific version that was used during development and testing. This ensures that the contract will always be compiled with a known, stable compiler version, preventing unexpected changes in behavior due to compiler updates. For example, instead of using `^0.8.xx`, explicitly define the version with `pragma solidity 0.8.19;`.

Before selecting a version, review known bugs and vulnerabilities associated with each Solidity compiler release. This can be done by referencing the official Solidity compiler release notes: Solidity GitHub releases or Solidity Bugs by Version. Choose a compiler version with a good track record for stability and security.

Resolution: The Finding is fixed in or before the commit `50db70c6ed5be63a6126e02da4f58b4f2f42abd7`.

The compiler version in pinned to `0.8.24`.

F-2024-7754 - Redundant Imports - Info

Description:

The contract `Initializable` is imported in `ZeUSD_Router` contract, but `Initializable` is already part of `UUPSUpgradeable`.

The contract `Initializable` is imported in `CollateralVault` contract, but `Initializable` is already part of `UUPSUpgradeable`.

The interface `IERC20` is imported in `USD0PPSubVault` contract, but `IERC20` is already part of `SafeERC20`.

The interface `IFundVaultV2` is imported in `USD0PPSubVault` contract, but it is never used.

The library `DataTypes` is imported in `IZeUSDRouter` interface, but it is never used.

This redundancy in import operations has the potential to result in unnecessary gas consumption during deployment and could potentially impact the overall code quality.

Assets:

- contracts/interfaces/IZeUSDRouter.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation: Remove redundant imports, and ensure that the contract is imported only in the required locations, avoiding unnecessary duplications.

Resolution: The Finding is fixed in the commit `6db2f8ae8b989df2c68bd6ead84380edfd7f89cc`.

The redundant imports are removed.

F-2024-7798 - Misleading Documentation of TokenType Enum Numeric Representation - Info

Description:

The `DataTypes::TokenType` enum defines two token types: stable coins and non-stable coins. Its implementation is as follows:

```
enum TokenType {
    /// @notice Represents non-stablecoin tokens (e.g., ETH, BTC)
    /// @dev Value = 1, optimized for gas when checking non-stable status
    NotStableCoin,
    /// @notice Represents stablecoins (e.g., USDC, DAI)
    /// @dev Value = 0, optimized for gas when checking stable status
    StableCoin
}
```

According to the Solidity documentation:

> The data representation is the same as for enums in C: The options are represented by subsequent unsigned integer values starting from `0`.

The comments in the code incorrectly suggest that `NotStableCoin` has a value of 1 and `StableCoin` has a value of 0. Based on the default behavior of Solidity enums, `NotStableCoin` has a numeric value of 0, and `StableCoin` has a numeric value of 1. This inconsistency may cause confusion and lead to misconfiguration during the setup of new sub-vaults.

Assets:

- contracts/libraries/DataTypes.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation:

It is recommended to update the comments in the `TokenType` enum to accurately reflect the actual numeric values of its fields as determined by Solidity's enum behavior. Correct and consistent descriptions will help prevent misinterpretations and errors in the configuration of system components.

Resolution:

The Finding is fixed in the commit `8c06119690f5ce4b400fd75c90f88b6fb734507a`.

Comments in the `TokenType` enum accurately reflect the actual numeric values of its fields.

F-2024-7807 - STABLE_PRICE Calculation Introduces Additional Complexity and Gas Consumption - Info

Description:

When the price of the collateral cannot be retrieved successfully, the `STABLE_PRICE` is used as a fallback. However, in calculations, the `STABLE_PRICE` is multiplied by `10**6` every time it is used. This approach introduces unnecessary complexity to the code and increases gas consumption. Instead of storing the base value (`1`) and performing the multiplication during calculations, the result of `1 * 10**6` (i.e., `1e6`) could be stored directly in the `STABLE_PRICE` variable.

```
uint8 public immutable STABLE_PRICE = 1;

function calculateMintAmount(...) {
    {...}
    assetPrice = success ? oraclePrice / 100 : STABLE_PRICE * 10 ** 6;
    {...}
}

function recordDeposit(...) {
    {...}
    assetPrice = success ? oraclePrice / 100 : STABLE_PRICE * 10 ** 6;
    {...}
}
```

This optimization would simplify the implementation and reduce computational overhead, leading to improved efficiency.

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation:

It is recommended to update the `STABLE_PRICE` variable to store the precomputed value of `1e6` instead of the base value. This change would eliminate the need for repeated multiplications in calculations, reduce gas consumption, and simplify the codebase.

Resolution:

The Finding was fixed in commit `1abadb84f3bf39398ffca32a107b89cc0bbf8a24`.

The `STABLE_PRICE` constant variable value was updated to 1e6 and redundant mathematical operations were removed.

F-2024-7845 - Missing SafeERC20 Usage in ZeUSD_Router Contract - Info

Description:

In the `ZeUSD_Router` contract, the `mintWithCollateral()`, `mintWithStable()`, `mintWithCollateralAndBridge()` and `mintWithStableAndBridge()` functions utilize the `ERC20::transferFrom()` function directly, instead of using the `SafeERC20::safeTransferFrom()` method. The absence of SafeERC20 usage in the mint related functions increases the risk of unintended reverts when interacting with non-compliant `ERC20` tokens, potentially leading to transaction failures and denial of service for users. For example Tether (USDT)'s `transfer()` and `transferFrom()` functions on L1 do not return booleans as the specification requires, and instead have no return value.

Assets:

- contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation:

It is recommended to replace the `ERC20::transferFrom()` call in the `mintWithCollateral()`, `mintWithStable()`, `mintWithCollateralAndBridge()` and `mintWithStableAndBridge()` functions with the `SafeERC20::safeTransferFrom()` method. This ensures compatibility with non-standard `ERC20` token implementations and reduces the risk of reversion. Adopting `SafeERC20` methods consistently throughout the contract improves reliability and mitigates compatibility issues, particularly when interacting with tokens that do not strictly adhere to the `ERC20` standard.

Resolution:

The Finding was fixed in commit `9bb9500045a25cf89d8b4904e36bd09e4f9b7223`.

The `ERC20::transferFrom()` calls were replaced by the `SafeERC20::safeTransferFrom()`.

F-2024-7865 - Missing Router Role Assignment During Initialization - Info

Description: The `CollateralVault` contract relies on a `router` address to perform key operations, such as managing deposits and interacting with subvaults. However, the `router` role is not assigned during the contract's `initialize` function, leaving it dependent on a subsequent manual role assignment through the `setRouter` function. This oversight can delay the operational readiness of the protocol and introduces potential risks of misconfiguration or misuse during the assignment process.

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Accepted

Recommendations

Remediation: It is recommended to update the `initialize` function to include a router address as an input parameter and assign the appropriate role to it during initialization.

Resolution: The Finding was accepted with the following statement:

> The decision to assign the router role through the setRouter function, instead of during initialization, is intentional and designed for flexibility and security.
> Key reasons for this approach include:
> > **Deployment Flexibility**: The router address may not always be known during initialization. Assigning it later ensures accurate configuration once all components are ready.
> > **Enhanced Security**: Using a dedicated function allows for verification of the router address before it's set, reducing the risk of misconfiguration.
> > **Simplified Initialization:** Keeping the initialization process simple lowers the risk of deployment errors.
> While this requires an extra step, it ensures the router is assigned carefully and aligns with the protocol's focus on secure and reliable operations.

F-2024-7873 - Unused Deposit Removal Functions Increase Code Complexity And Size - Info

Description: The `CollateralVault` contract contains functions that are exclusively callable by the `ZeUSD_Router` contract, protected by the `onlyRouter` modifier. These functions are designed to track user deposit data. However, the `removeDeposit()` and `removeBulkDeposits()` functions are not utilized in the current version of the `ZeUSD_Router` contract.

The inclusion of these unused functions introduces unnecessary logic, increasing deployment size, reducing code readability, and adding complexity. Additionally, unused logic may create potential attack vectors in future updates. As the contract is upgradeable, the functionality of these methods can be implemented later if needed.

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Retest Requested

Recommendations

Remediation: It is recommended to remove the logic within the `removeDeposit()` and `removeBulkDeposits()` functions and replace their implementation with a revert statement, including a revert message indicating that these functions are not supported (similar as in the `USD0PPSubVault::addAsset()`).

Potential fix:

```
function removeBulkDeposits(address user, uint256[] calldata depositIds)
    external onlyRouter whenNotPaused returns (bool success, uint256 totalMintAmount) {
    revert('Not supported in this version');
    }

function removeDeposit(address user, uint256 depositId)
    external onlyRouter whenNotPaused returns (bool success, uint256 mintAmount) {
    revert('Not supported in this version');
    }
```

Resolution: The Finding is fixed in or before the commit `50db70c6ed5be63a6126e02da4f58b4f2f42abd7`.

`removeDeposit()` and `removeBulkDeposits()` functions have been included in the router.

F-2024-7893 - Repeated Zero Address Checks - Info

Description: Throughout the codebase, it is common to see repeated checks for conditions like zero addresses across multiple functions. For instance, validating that an address is not `address(0)` is essential to ensure the integrity of operations, as null addresses can lead to vulnerabilities or unintended behavior. However, duplicating these checks in multiple places increases code redundancy.

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation: It is recommended to incapsulate this logic in a reusable modifier or function, similar to the `validAmount` modifier used for zero-value checks. A dedicated modifier like `validAddress(address)` can validate that an address is non-zero and thus streamline the codebase.

Resolution: The Finding is fixed in the commit `56850878695dd422a735354b376b65fa89453f0c`.

The modifier `validAddress` was implemented and is used in the majority of the codebase.

F-2024-7894 - Public Functions that Can be External - Info

Description: Functions that are meant to be exclusively invoked from external sources should be designated as `external` rather than `public`.

The visibility of the following functions can be restricted:

ZeUSD:

- `burnFrom`
- `decimals`
- `setBlacklistStatus`

Assets:

- contracts/implementations/ZeUSD.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation: Consider updating functions which are exclusively utilized by external entities from their current `public` visibility to `external` visibility.

Resolution: The Finding is fixed in the commit `d523644ffe8175d802ffd2d627108292031151 43`.

Visibility on the `setBlacklistStatus` function is set to external.

F-2024-7896 - Revert String Size Optimization - Info

Description:

Several functions in the contract use long revert strings that exceed 32 bytes, leading to increased gas costs during both deployment and runtime. When a revert condition is triggered with a long string, the EVM requires additional operations such as an extra `mstore` to handle the longer data, as well as overhead for calculating memory offsets. This results in higher gas consumption compared to shorter revert strings that fit within 32 bytes. Examples of such functions and modifiers include:

ZeUSD_Router:

- `setupInitialApproval()`
- `resetApproval()`

ZeUSD:

- `onlyRouter()`
- `notBlacklisted()`

USD0PPSubVault:

- `handleDeposit()`
- `addAsset()`
- `removeAsset()`

Shortening the revert strings to fit within 32 bytes will decrease deployment time and decrease runtime Gas when the revert condition is met.

Assets:

- contracts/implementations/ZeUSD.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status:

Fixed

Recommendations

Remediation:

To optimize Gas usage in your Solidity contract, it is recommended to keep revert strings as short as possible and to ensure that they fit within 32 bytes. It is possible to use abbreviations or simplified error messages to keep the string length short.

Resolution:

The Finding is fixed in the commit `bb7c015d2232635fbdd591d45d2175cd8bb9f752`.

All revert strings now fit within 32 bytes.

F-2024-7897 - Unimplemented Function - Info

Description:

The `USD0PPSubVault` contract does not support deposits but inherits the `ISubVault` interface, which requires the implementation of the `handleDeposit()` function. To allow compilation, the `handleDeposit()` function is overridden in the `USD0PPSubVault` contract.

Typically, in such cases, a simple revert statement is used to indicate that the logic is not supported. However, the current implementation of the `handleDeposit()` function performs additional checks before reverting. These redundant checks increase the contract size and can cause confusion regarding the intended functionality of the function.

```
 /// @notice Handles deposit of supported assets
/// @dev Only handles primary asset deposits, reverts for secondary assets
/// @param asset Address of asset being deposited
/// @param amount Amount to deposit
/// @return success Whether deposit was successful
function handleDeposit(
    //@audit is different from other sub vaults and does not use grant approval
    address,
    address asset,
    uint256 amount
) external override nonReentrant onlyRouter whenNotPaused returns (bool) {
    if (amount == 0) revert InvalidAmount();
    if (emergencyMode) revert EmergencyModeEnabled(block.timestamp);

    // Only allow deposits of primary asset
    if (asset != USD0PP) revert UnsupportedAsset(asset);

    // Primary asset deposits not supported in this vault
    revert('Deposits not supported in this vault');
}
```

Assets:

- contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Accepted

Recommendations

Remediation:

It is recommended to remove the safety checks in the `handleDeposit` function and retain only the revert statement. This approach will simplify the function, reduce contract size, and avoid confusion about unsupported functionality.

Potential fix:

```
function handleDeposit(
    address, address, uint256
) external override nonReentrant onlyRouter whenNotPaused returns(bool) {
```

```
        // Primary asset deposits not supported in this vault
        revert('Deposits not supported in this vault');
    }
```

Resolution:

The Finding was accepted with the following statement:

> This function will be upgraded later on. As support for other asset will be added.

F-2024-7898 - Unused Error Definition - Info

Description:

The `ICollateralVaultErrors`, `ISubVaultErrors`, and `IZeUSDRouterErrors` interfaces contain error definitions that are not utilized in the corresponding contracts. These interfaces are inherited by their respective contracts, but several error definitions remain unused.

The presence of unused error definitions unnecessarily increases the contract size, creates potential confusion about implemented error-handling logic, and adds to the overall complexity of the system.

Unused errors:

- `ICollateralVaultErrors`: CalculationOverflow(),
- `ISubVaultErrors`: AssetAlreadySupported(), DepositFailed(), WithdrawFailed(), InsufficientBalance(), CannotRemovePrimaryAsset(), PrimaryAssetOperationFailed(), SecondaryAssetOperationFailed()
- `IZeUSDRouterErrors`: IntegrationNotFound(), IntegrationAlreadyExists(), IntegrationNotActive(), SubVaultPaused(), AdminRequired(), NoLockedAssets(), InsufficientLockedAssets(), AssetAlreadyRegistered(), SubvaultAlreadyRegistered()

Assets:

- contracts/interfaces/ISubVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/interfaces/ICollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]
- contracts/interfaces/IZeUSDRouter.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation:

It is recommended to remove unused error definitions from the `ICollateralVaultErrors`, `ISubVaultErrors`, and `IZeUSDRouterErrors` interfaces. This will reduce contract size, simplify the codebase, and improve clarity regarding implemented error-handling logic.

Resolution:

The Finding is fixed in the commit `61bab92a5336754a0449740b4a017fca62cfa71d`.

The unused Errors are removed.

F-2024-7964 - Missing _disableInitializers() in Upgradable Contract Constructor - Info

Description: The `CollateralVault` contract is designed as an upgradable contract. After deploying the implementation contract on the blockchain, functions must be invoked via the proxy to establish basic functionalities. However, the implementation contract does not call `_disableInitializers()` in its `constructor()`. This omission allows external actors to directly initialize the implementation contract, potentially disrupting the intended upgrade workflow and leading to unintended contract behavior.

Assets:

- contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation: It is recommended to include a call to `_disableInitializers()` in the `constructor()` of the implementation contract. This will prevent any direct initialization of the implementation contract and ensure the security of the upgrade process.

Potential fix:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
   _disableInitializers();
}
```

Resolution: The Finding is fixed in the commit 56850878695dd422a735354b376b65fa89453f0c.

A call to `_disableInitializers()` in the `constructor()` was included.

F-2024-7967 - Lack of @custom:oz-upgrades-unsafe-allow Comment in the ZeUSD_Router Contract - Info

Description:

The `ZeUSD_Router` is an upgradable contract that implements the `UUPS` pattern from OpenZeppelin. In upgradable contracts, constructors should not be used to initialize the contract state. Instead, initialization is performed through the `initialize()` function.

```
constructor() {
    _disableInitializers();
}
```

In OpenZeppelin version 4.6, the `_disableInitializers()` function was introduced to prevent uninitialized contracts from being taken over by attackers. When `_disableInitializers()` is used in the constructor, OpenZeppelin recommends including the NatSpec comment `@custom:oz-upgrades-unsafe-allow constructor` to disable the safety check during deployment with their Upgrades Plugins.

The `ZeUSD_Router` contract lacks this NatSpec comment in its `constructor()`, leading to a deviation from best practices. This omission results in compilation errors when the contract is compiled using Foundry with OpenZeppelin Upgrades Plugins.

Assets:

- contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]

Status: Fixed

Recommendations

Remediation:

It is recommended to include the NatSpec comment `@custom:oz-upgrades-unsafe-allow constructor` in the constructor of the ZeUSD_Router contract where `_disableInitializers()` is invoked.

Potential fix:

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

Resolution:

The Finding is fixed in the commit `56850878695dd422a735354b376b65fa89453f0c`.

NatSpec comment `@custom:oz-upgrades-unsafe-allow constructor` added to the constructor.

Disclaimers

Hacken Disclaimer

The smart contracts given for audit have been analyzed based on best industry practices at the time of the writing of this report, with cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report (Source Code); the Source Code compilation, deployment, and functionality (performing the intended functions).

The report contains no statements or warranties on the identification of all vulnerabilities and security of the code. The report covers the code submitted and reviewed, so it may not be relevant after any modifications. Do not consider this report as a final and sufficient assessment regarding the utility and safety of the code, bug-free status, or any other contract statements.

While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only — we recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contracts.

English is the original language of the report. The Consultant is not responsible for the correctness of the translated versions.

Technical Disclaimer

Smart contracts are deployed and executed on a blockchain platform. The platform, its programming language, and other software related to the smart contract can have vulnerabilities that can lead to hacks. Thus, the Consultant cannot guarantee the explicit security of the audited smart contracts.

Appendix 1. Definitions

Severities

When auditing smart contracts, Hacken is using a risk-based approach that considers **Likelihood**, **Impact**, **Exploitability** and **Complexity** metrics to evaluate findings and score severities.

Reference on how risk scoring is done is available through the repository in our Github organization:

hknio/severity-formula

Severity	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to the loss of user funds or contract state manipulation.
High	High vulnerabilities are usually harder to exploit, requiring specific conditions, or have a more limited scope, but can still lead to the loss of user funds or contract state manipulation.
Medium	Medium vulnerabilities are usually limited to state manipulations and, in most cases, cannot lead to asset loss. Contradictions and requirements violations. Major deviations from best practices are also in this category.
Low	Major deviations from best practices or major Gas inefficiency. These issues will not have a significant impact on code execution.

Potential Risks

The "Potential Risks" section identifies issues that are not direct security vulnerabilities but could still affect the project’s performance, reliability, or user trust. These risks arise from design choices, architectural decisions, or operational practices that, while not immediately exploitable, may lead to problems under certain conditions. Additionally, potential risks can impact the quality of the audit itself, as they may involve external factors or components beyond the scope of the audit, leading to incomplete assessments or oversight of key areas. This section aims to provide a broader perspective on factors that could affect the project's long-term security, functionality, and the comprehensiveness of the audit findings.

Appendix 2. Scope

The scope of the project includes the following smart contracts from the provided repository:

Scope Details	
Repository	https://github.com/0xZothio/zeusd-contracts
Commit	a0842125be71adf3784ba91b45fd3476a2ceb7d9
Remediation commit	75a10af095e97785f0a3ed410f91cf02432e171c
Whitepaper	-
Requirements	https://docs.google.com/document/d/14nwJD55OjzAgiy82sCeyxQ5WefFnF6AVGPKJYGCjXc8/edit?usp=sharing
Technical Requirements	https://docs.google.com/document/d/14nwJD55OjzAgiy82sCeyxQ5WefFnF6AVGPKJYGCjXc8/edit?usp=sharing

Asset	Type
contracts/CollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/interfaces/ICollateralVault.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/interfaces/IFundVaultV2.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/interfaces/IPriceOracle.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/interfaces/ISubVault.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/interfaces/IZeUSD.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/interfaces/IZeUSDOFT.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/interfaces/IZeUSDRouter.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/libraries/DataTypes.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/subVaults/USD0PPSubVault.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract
contracts/ZeUSD_Router.sol [https://github.com/0xZothio/zeusd-contracts]	Smart Contract

Appendix 3. Additional Valuables

Verification of System Invariants

During the audit of Zoth / Zeusd-Contracts, Hacken followed its methodology by performing fuzz-testing on the project's main functions. [Foundry](), a tool used for fuzz-testing, was employed to check how the protocol behaves under various inputs. Due to the complex and dynamic interactions within the protocol, unexpected edge cases might arise. Therefore, it was important to use fuzz-testing to ensure that several system invariants hold true in all situations.

Fuzz-testing allows the input of many random data points into the system, helping to identify issues that regular testing might miss. A specific Echidna fuzzing suite was prepared for this task, and throughout the assessment, 5 fuzz scenarios were tested over 1M runs each. This thorough testing ensured that the system works correctly even with unexpected or unusual inputs.

Invariant	Test Result	Run Count
Mint ZeUSD with random amount of USD0++ and price 1	Passed	1M
Mint ZeUSD with random amount of USD0++ and price 0.5	Passed	1M
Mint ZeUSD with random amount of USD0++ and price 100	Passed	1M
Burn random amount of ZeUSD	Passed	1M
Mint ZeUSD with random LTV	Passed	1M

Additional Recommendations

The smart contracts in the scope of this audit could benefit from the introduction of automatic emergency actions for critical activities, such as unauthorized operations like ownership changes or proxy upgrades, as well as unexpected fund manipulations, including large withdrawals or minting events. Adding such mechanisms would enable the protocol to react automatically to unusual activity, ensuring that the contract remains secure and functions as intended.

To improve functionality, these emergency actions could be designed to trigger under specific conditions, such as:

- Detecting changes to ownership or critical permissions.
- Monitoring large or unexpected transactions and minting events.
- Pausing operations when irregularities are identified.

These enhancements would provide an added layer of security, making the contract more robust and better equipped to handle unexpected situations while maintaining smooth operations.