Cyberscope

Audit Report

dTRINITY dusd

October 2024

Repository https://github.com/dtrinity/solidity-contracts-preview/tree/main/contracts/dusd

Commit e21bdc5083e1c9a56019b13a51e7e8bc0145aac8

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
● Critical	Highly Likely / High Impact
● Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
● Minor / Informative	Unlikely / Low to no Impact

Review

Repository	https://github.com/dtrinity/solidity-contracts-preview/tree/main/contracts/dusd
Commit	e21bdc5083e1c9a56019b13a51e7e8bc0145aac8

Audit Updates

Initial Audit	21 Oct 2024

Source Files

Filename	SHA256
UniV3AmoVault.sol	85e53e391bb5ac8dadf9d63220071dd610dd85c922b280f35f0e7fdcc53d23d9
Redeemer.sol	819e7d00f2f3ae5c3dc4ad4fe6f6efb94382579826180a7064df4a6764ddf4b3
OracleAware.sol	953595cf6b227c08198e26794d5a171f65183c3c3a7412233ae73e4a81fccf5c
OracleAggregator.sol	8668d7358e9f8cc22c55d6de5cd3584b3163bdd76b0c6f64acca5da68447cdaf
Issuer.sol	60e5486878d797829cf6074f175a26bff713ba9ea7eb886946906b0d7559f128
CollateralVault.sol	cb20e4edca2b2b2b00553922c19c06de5952384afcdb4f11a22ebc8757e91bc2
AmoVault.sol	c36dacee732a2df5ebbd1d9fa0b46172be4ac430d4d21ae6fa9a7b413843baf9
AmoManager.sol	ce117d49aabbdaf1037b58094bf053a5e0be3a7df20f5bff68dd990a79ff0727

Overview

AmoManager Contract Functionality

The AmoManager contract is designed to manage the allocation and deallocation of dUSD tokens to various AMO (Algorithmic Market Operations) vaults. It maintains a set of active AMO vaults and ensures that tokens can only be allocated to and deallocated from authorized vaults. The contract enforces access control for allocating and deallocating dUSD, requiring specific roles to execute these operations. Additionally, it tracks the total allocated supply of dUSD and provides functions to calculate the collateral value of active AMO vaults. The contract also allows administrators to enable or disable AMO vaults, burn dUSD to reduce the AMO supply, and remove vaults from the active list. It emits events for vault changes and token movements to enhance transparency.

AmoVault Contract Functionality

The AmoVault contract is an abstract contract that serves as a vault for managing dUSD and collateral tokens in coordination with the AmoManager contract. It allows for the withdrawal of collateral tokens by addresses with the COLLATERAL_WITHDRAWER_ROLE, as well as the recovery of accidentally sent ERC20 tokens and ETH through the RECOVERER_ROLE. The contract includes mechanisms to approve the AmoManager for spending dUSD on its behalf and enforces access control for various functions to ensure that only authorized roles can execute sensitive operations. It integrates a reentrancy guard to protect against reentrancy attacks and provides functions to check and manage collateral assets.

CollateralVault Contract Functionality

The CollateralVault contract is responsible for managing the deposit, withdrawal, and exchange of collateral assets. It allows for the deposit of collateral tokens into the vault by authorized users and provides functionality for withdrawing collateral with the appropriate access control. The contract uses a price oracle to determine the value of collateral assets and supports the exchange of collateral between different tokens while maintaining equal value. The contract also implements access control mechanisms to ensure that only authorized roles can manage collateral, withdraw assets, and execute strategy-related functions. Additionally, it maintains a list of supported collateral assets, allowing new assets to be added or disallowed by a designated collateral manager.

Issuer Contract Functionality

The Issuer contract manages the issuance of dUSD tokens by accepting collateral deposits from users or third parties. It interacts with a price oracle to ensure the correct conversion of collateral into dUSD based on market value, and it handles the minting of dUSD tokens to users who provide collateral. The contract also has the ability to issue dUSD using excess collateral in the system and increase the AMO supply by minting tokens to an AmoManager. The contract includes access control for minting, issuing tokens, and managing AMO supply, with designated roles for specific functions. Additionally, it tracks the circulating supply of dUSD and the total value of collateral in the system.

OracleAggregator Contract Functionality

The OracleAggregator contract aggregates price data from multiple oracles for various assets and applies thresholding where required. It uses a mapping to associate each asset with a designated oracle and an additional mapping to determine whether thresholding should be applied to the price data for a specific asset. The contract allows the setting and updating of oracles and threshold values, which are managed by a role-based access control system. The contract is compatible with protocols such as Aave through its implementation of the IPriceOracleGetter interface, enabling it to return asset prices in USD with a defined number of decimal places. When thresholding is enabled for an asset, if the price exceeds a certain threshold, the price is capped to a fixed value. This design ensures that the contract provides reliable and consistent price data while protecting against extreme fluctuations in asset prices.

OracleAware Contract Functionality

The OracleAware contract is an abstract base contract designed to provide oracle functionality to other contracts. It maintains a reference to an external price oracle, which is used for fetching asset price data. The contract allows the oracle to be updated by accounts with the DEFAULT_ADMIN_ROLE, enabling flexibility in choosing or switching oracles as needed. The contract emits events whenever the oracle is updated, ensuring transparency. It provides an essential foundation for contracts that need reliable price data for collateral valuation or other financial calculations.

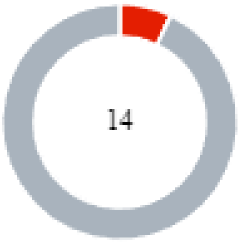
Redeemer Contract Functionality

The Redeemer contract enables the redemption of dUSD tokens in exchange for collateral from the associated CollateralVault. Users can redeem their dUSD tokens for a specified amount of collateral, subject to slippage protection. The contract allows redemptions from the caller's own balance or on behalf of another address. The redeemed dUSD tokens are transferred to the contract, burned, and the equivalent value of collateral is calculated and withdrawn from the vault. The contract uses an external price oracle to ensure accurate conversion between dUSD and collateral assets. The Redeemer contract is governed by access control roles, ensuring only authorized accounts can initiate redemptions and manage collateral settings.

UniV3AmoVault Contract Functionality

The UniV3AmoVault contract extends the functionality of the AmoVault contract and integrates with Uniswap V3 to provide liquidity management for the dUSD token. This contract enables adding and removing liquidity positions in a Uniswap V3 pool, as well as conducting swaps through the Uniswap V3 router. The vault tracks liquidity positions and calculates their value using external price oracles. The contract also allows collecting fees from Uniswap positions and managing collateral, which is used to maintain the value of the dUSD token. The contract ensures that only authorized traders can perform liquidity operations through access control roles.

Findings Breakdown



Critical	1
Medium	0
Minor / Informative	13

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	0	0	0	0
Minor / Informative	13	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	MCWRA	Missing Collateral Withdrawal Role Assignment	Unresolved
	CCR	Contract Centralization Risk	Unresolved
	IRA	Inconsistent Role Assignment	Unresolved
	IVDL	Incorrect Vault Disabling Logic	Unresolved
	MDAC	Missing Deposit Access Control	Unresolved
	MEM	Missing Error Messages	Unresolved
	RC	Redundant Check	Unresolved
	RCAS	Redundant Collateral Address Storage	Unresolved
	UPAF	Unnecessary Public Approval Function	Unresolved
	URS	Unnecessary Return Statements	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L13	Divide before Multiply Operation	Unresolved
	L15	Local Scope Variable Shadowing	Unresolved
	L19	Stable Compiler Version	Unresolved

Severity	Code	Description	Status
●	MCWRA	Missing Collateral Withdrawal Role Assignment	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	IRA	Inconsistent Role Assignment	Unresolved
●	IVDL	Incorrect Vault Disabling Logic	Unresolved
●	MDAC	Missing Deposit Access Control	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	UPAF	Unnecessary Public Approval Function	Unresolved
●	URS	Unnecessary Return Statements	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L15	Local Scope Variable Shadowing	Unresolved
●	L19	Stable Compiler Version	Unresolved

MCWRA - Missing Collateral Withdrawal Role Assignment

Criticality	Critical
Location	Redeemer.sol#L52,74,98 CollateralVault.sol#L113

Status	Unresolved

Description

The `redeemFrom` function in the `Redeemer` contract calls the `withdrawFrom` function in the CollateralVault contract to withdraw collateral. However, the `withdrawFrom` function in `CollateralVault` is protected by the `COLLATERAL_WITHDRAWER_ROLE` access control. This means that for the `redeemFrom` function to successfully withdraw collateral, the `Redeemer` contract must have the `COLLATERAL_WITHDRAWER_ROLE` assigned. Without this role being granted, the `redeemFrom` function will fail when attempting to withdraw collateral, which could disrupt the redemption process.

```
function redeem(
        uint256 dusdAmount,
        address collateralAsset,
        uint256 minCollateral
    ) external onlyRole(REDEMPTION_MANAGER_ROLE) {
        _redeem(
            msg.sender,
            msg.sender,
            dusdAmount,
            collateralAsset,
            minCollateral
        );
    }

function _redeem(
        address withdrawer,
        address receiver,
        uint256 dusdAmount,
        address collateralAsset,
        uint256 minCollateral
    ) internal {
        // Transfer dUSD from withdrawer to this contract
        require(
            dusd.transferFrom(withdrawer, address(this),
dusdAmount),
            "dUSD transfer failed"
        );

        // Burn the dUSD
        dusd.burn(dusdAmount);

        // Calculate collateral amount
        uint256 dusdValue = dusdAmountToUsdValue(dusdAmount);
        uint256 collateralAmount =
collateralVault.collateralAmountFromValue(
            dusdValue,
            collateralAsset
        );
        require(
            collateralAmount >= minCollateral,
            "Too much slippage during redemption"
        );

        // Withdraw collateral from the vault
        collateralVault.withdrawFrom(
            receiver,
            collateralAmount,
            collateralAsset
        );
```

```
        // No invariant checks here, since only redemption
manager can redeem,
        // and may need to redeem during adverse market
conditions
    }

function withdrawFrom(
        address withdrawer,
        uint256 collateralAmount,
        address collateralAsset
    ) public onlyRole(COLLATERAL_WITHDRAWER_ROLE) {
        return _withdraw(withdrawer, collateralAmount,
collateralAsset);
    }
```

Recommendation

It is recommended to ensure that the `COLLATERAL_WITHDRAWER_ROLE` is granted to the `Redeemer` contract, either during the initialization process in the constructor of the `CollateralVault` contract. This would allow the `redeemFrom` function to execute the collateral withdrawal as intended, ensuring the redemption process can proceed without access control issues.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	AmoManager.sol#L149,160,171 AmoVault.sol#L65,86,105 CollateralVault.sol#L100,149,184,293,313 Issuer.sol#L116,201,213 OracleAggregator.sol#L74,95,107 OracleAware.sol#L27 Redeemer.sol#L154 UniV3AmoVault.sol#L110,158,202,245,281
Status	Unresolved

Description

The contracts' functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function enableAmoVault(
        address amoVault
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_amoVaults.add(amoVault), "AMO vault already
enabled");
        emit AmoVaultSet(amoVault, true);
    }

function disableAmoVault(
        address amoVault
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_amoVaults.remove(amoVault), "AMO vault not
found");
        emit AmoVaultSet(amoVault, false);
    }

...
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

IRA - Inconsistent Role Assignment

Criticality	Minor / Informative
Location	OracleAggregator.sol#L60
Status	Unresolved

Description

In the `OracleAggregator` contract, the constructor uses _grantRole for assigning both the `DEFAULT_ADMIN_ROLE` and `ORACLE_MANAGER_ROLE` to `msg.sender`. However, in other contracts within the same codebase, roles are assigned in a more consistent manner, where `_grantRole` is used to assign the `DEFAULT_ADMIN_ROLE`, and subsequent roles are assigned using the external `grantRole` function. This inconsistency can lead to confusion, as it deviates from the standard approach used throughout the project. Additionally, using `grantRole` ensures that role assignment events are emitted, providing better traceability on-chain.

```
constructor(uint8 _priceDecimals, uint256 _thresholdUsd) {
        priceDecimals = _priceDecimals;
        priceUnit = 10 ** _priceDecimals;
        thresholdUsd = _thresholdUsd;

        _grantRole(DEFAULT_ADMIN_ROLE, msg.sender);
        _grantRole(ORACLE_MANAGER_ROLE, msg.sender);
    }
```

Recommendation

It is recommended to align the role assignment in the `OracleAggregator` contract with the pattern used in other contracts. The `DEFAULT_ADMIN_ROLE` should be assigned using `_grantRole`, while subsequent roles, such as `ORACLE_MANAGER_ROLE`, should be assigned using grantRole to ensure consistency and proper event emission across the codebase.

IVDL - Incorrect Vault Disabling Logic

Criticality	Minor / Informative
Location	AmoManager.sol#L160
Status	Unresolved

Description

The `disableAmoVault` function in the contract is intended to disable an AMO vault, which suggests that the vault should remain in the set but be marked as inactive. However, the function currently removes the vault from the `_amoVaults` set entirely. This causes the vault to be inaccessible for any future actions, such as reactivating the vault or deallocating any assets it holds. The function’s current behavior is inconsistent with its name and description, as it performs a removal operation rather than simply disabling the vault.

```
function disableAmoVault(
        address amoVault
    ) public onlyRole(DEFAULT_ADMIN_ROLE) {
        require(_amoVaults.remove(amoVault), "AMO vault not
found");
        emit AmoVaultSet(amoVault, false);
    }
```

Recommendation

To align with the intended functionality of disabling a vault, the implementation should be adjusted so that the vault remains in the set but its status is updated to inactive. Consider introducing a mechanism that tracks whether a vault is active or inactive without removing it from the list. This would allow administrators to deactivate vaults without removing them entirely, ensuring that future reactivation or asset management operations are possible.

MDAC - Missing Deposit Access Control

Criticality	Minor / Informative
Location	CollateralVault.sol#L61,75
Status	Unresolved

Description

The `depositFrom` function in the `CollateralVault` contract lacks access control, allowing any external account to call the function and initiate a deposit on behalf of a depositor. Although the function relies on the ERC20 `safeTransferFrom` mechanism, which requires the depositor to grant token allowance to the contract, this alone may not be sufficient to prevent misuse or unintended calls. Without proper access control, unauthorized accounts could potentially initiate deposits, leading to unexpected behavior or security risks.

```
function depositFrom(
        address depositer,
        uint256 collateralAmount,
        address collateralAsset
    ) public {
        return _deposit(depositer, collateralAmount, collateralAsset);
    }

function _deposit(
        address depositer,
        uint256 collateralAmount,
        address collateralAsset
    ) internal {
        // Make sure the collateral is active
        require(
            _supportedCollaterals.contains(collateralAsset),
            "Unsupported collateral"
        );

        IERC20Metadata(collateralAsset).safeTransferFrom(
            depositer,
            address(this),
            collateralAmount
        );
    }
```

Recommendation

It is recommended to implement access control to restrict who can call the `depositFrom` function. This ensures that only trusted and authorized roles, such as a collateral manager, are able to initiate deposits on behalf of other users. This additional layer of security would prevent abuse and enhance the overall integrity of the system by ensuring that sensitive operations are only performed by authorized accounts.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	AmoManager.sol#L65,94
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(endingAmoSupply == startingAmoSupply)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

RC - Redundant Check

Criticality	Minor / Informative
Location	AmoManager.sol#L112
Status	Unresolved

Description

In the `totalCollateralValue` function, the contract checks whether each vault in the `_amoVaults` set is active before calculating its collateral value. This check is redundant because the vaults that are inactive or removed would already be excluded from the `_amoVaults` set due to the behavior of the `disableAmoVault` and `removeAmoVault` functions. Both of these functions ensure that inactive vaults are either disabled or completely removed from the set, making it unnecessary to recheck their status during the collateral calculation. This adds unnecessary complexity to the logic and may result in inefficiencies.

```
function totalCollateralValue() public view returns (uint256) {
        uint256 totalValue = 0;
        for (uint256 i = 0; i < _amoVaults.length(); i++) {
            address vaultAddress = _amoVaults.at(i);
            if (isAmoActive(vaultAddress)) {
                IAmoVault vault = IAmoVault(vaultAddress);
                totalValue += vault.totalCollateralValue();
            }
        }
        return totalValue;
    }
```

Recommendation

The `isAmoActive` check should be removed from the `totalCollateralValue` function since the vaults in the `_amoVaults` set are guaranteed to be active. Simplifying this function will improve code clarity and reduce gas costs by avoiding an unnecessary condition.

RCAS - Redundant Collateral Address Storage

Criticality	Minor / Informative
Location	UniV3AmoVault.sol#L140
Status	Unresolved

Description

The `Position` struct in the `UniV3AmoVault` contract stores the collateral token's address for each position, even though the collateral token is an immutable state variable. Since the collateral token address remains constant across all positions, storing this value in each Position instance is unnecessary and increases storage costs. This redundancy leads to inefficient use of gas and contract storage.

```
Position memory newPosition = Position({
            tokenId: tokenId,
            collateral: address(collateralToken),
            liquidity: liquidity,
            tickLower: params.tickLower,
            tickUpper: params.tickUpper
        });
```

Recommendation

Remove the redundant storage of the collateral token address from the `Position` struct. Instead, rely on the immutable collateral token state variable to access the collateral token's address when needed. This optimization will save storage costs and improve the overall efficiency of the contract without affecting its functionality.

UPAF - Unnecessary Public Approval Function

Criticality	Minor / Informative
Location	AmoVault.sol#L76
Status	Unresolved

Description

The `approveAmoManager` function provides maximum approval for the `AmoManager` to spend dUSD tokens on behalf of the contract. Since the `AmoManager` is immutable and cannot be changed after deployment, there is no need for this function to be callable multiple times. The approval granted during the contract's construction ensures that the `AmoManager` has sufficient permission to transfer dUSD without further interaction.

```
function approveAmoManager() public
onlyRole(DEFAULT_ADMIN_ROLE) {
        dusd.approve(address(amoManager), type(uint256).max);
    }
```

Recommendation

To simplify the contract and reduce the potential for misuse, consider restricting the `approveAmoManager` function to the constructor, ensuring that it is only called once during deployment. Removing the public access to this function ensures that no further unnecessary approvals can be made, while maintaining the intended functionality of granting maximum approval to the `AmoManager`.

URS - Unnecessary Return Statements

Criticality	Minor / Informative
Location	CollateralVault.sol#L51,66
Status	Unresolved

Description

The `deposit` and `depositFrom` functions include `return` statements but do not return any value. This is misleading, as the use of return suggests that a value or result is expected, when in fact the functions only call internal logic without any return value.

```
function deposit(uint256 collateralAmount, address
collateralAsset) public {
        return _deposit(msg.sender, collateralAmount,
collateralAsset);
    }

function depositFrom(
        address depositer,
        uint256 collateralAmount,
        address collateralAsset
    ) public {
        return _deposit(depositer, collateralAmount,
collateralAsset);
    }
```

Recommendation

It is recommended to remove the unnecessary `return` statements in these functions to improve code clarity and maintainability. The absence of a return value should be clear from the function implementation.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Redeemer.sol#L155 OracleAggregator.sol#L96,119,127 Issuer.sol#L202,214
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
address _collateralVault
uint256 _thresholdUsd

function BASE_CURRENCY() external pure returns (address) {
        return BASE_CURRENCY_USD;
    }

function BASE_CURRENCY_UNIT() external view returns (uint256) {
        return priceUnit;
    }
address _amoManager
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	CollateralVault.sol#L226,229
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 fromCollateralValue = (fromCollateralPrice *
            fromCollateralAmount) / (10 **
fromCollateralDecimals)
toCollateralAmount =
            (fromCollateralValue * (10 **
toCollateralDecimals)) /
            toCollateralPrice
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	Redeemer.sol#L35 Issuer.sol#L41 CollateralVault.sol#L37
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

```
IPriceOracleGetter oracle
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	UniV3AmoVault.sol#L2 Redeemer.sol#L2 OracleAware.sol#L2 Issuer.sol#L2 CollateralVault.sol#L2 AmoVault.sol#L2 AmoManager.sol#L2
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
UniV3AmoVault	Implementation	AmoVault, OracleAware		
		Public	✓	AmoVault OracleAware
	totalCollateralValue	Public		-
	mint	External	✓	onlyRole
	burn	External	✓	onlyRole
	increaseLiquidity	External	✓	onlyRole
	decreaseLiquidity	External	✓	onlyRole
	collectFees	Public	✓	onlyRole
	swapExactOutputSingle	External	✓	onlyRole
	swapExactInputSingle	External	✓	onlyRole
	getPosition	Public		-
	getPositionsCount	Public		-
	getPositionByTokenId	Public		-
	_getPositionValueExcludingDusd	Internal		
	_isCollateral	Internal		
Redeemer	Implementation	AccessControl, Constants, OracleAware		
		Public	✓	OracleAware
	redeem	External	✓	onlyRole

	redeemFrom	External	✓	onlyRole
	_redeem	Internal	✓	
	dusdAmountToUsdValue	Public		-
	setCollateralVault	External	✓	onlyRole
OracleAware	Implementation	AccessControl		
		Public	✓	-
	setOracle	Public	✓	onlyRole
OracleAggregator	Implementation	AccessControl, IPriceOracleGetter		
		Public	✓	-
	setOracle	External	✓	onlyRole
	setThreshold	External	✓	onlyRole
	setApplyThresholding	External	✓	onlyRole
	BASE_CURRENCY	External		-
	BASE_CURRENCY_UNIT	External		-
	getAssetPrice	External		-
	getPriceInfo	Public		-
	_getPriceInfo	Private		
Issuer	Implementation	AccessControl, Constants, OracleAware		
		Public	✓	OracleAware

	issue	External	✓	-
	issueFrom	External	✓	-
	issueUsingExcessCollateral	External	✓	onlyRole
	increaseAmoSupply	External	✓	onlyRole
	_issue	Internal	✓	
	circulatingDusd	Public		-
	collateralInDusd	Public		-
	usdValueToDusdAmount	Public		-
	setAmoManager	External	✓	onlyRole
	setCollateralVault	External	✓	onlyRole
IUniswapV3Pool	Interface	IUniswapV3PoolImmutables, IUniswapV3PoolState, IUniswapV3PoolDerivedState, IUniswapV3PoolActions, IUniswapV3PoolOwnerActions, IUniswapV3PoolEvents		
IPriceOracleGetter	Interface			
	BASE_CURRENCY	External		-
	BASE_CURRENCY_UNIT	External		-
	getAssetPrice	External		-
IOracleWrapper	Interface			

	getPriceInfo	External		-
	getPriceDecimals	External		-
IERC20Stablecoin	Interface	IERC20		
	mint	External	✓	-
	burn	External	✓	-
	burnFrom	External	✓	-
	decimals	External		-
DexOracleWrapper	Implementation	IOracleWrapper		
		Public	✓	-
	getPriceInfo	External		-
	getPriceDecimals	External		-
	_calculatePriceDecimals	Private		
CollateralVault	Implementation	AccessControl, OracleAware		
		Public	✓	OracleAware
	deposit	Public	✓	-
	depositFrom	Public	✓	-
	_deposit	Internal	✓	
	withdraw	Public	✓	onlyRole
	withdrawFrom	Public	✓	onlyRole
	_withdraw	Internal	✓	

	exchangeCollateral	Public	✓	onlyRole
	exchangeMaxCollateral	Public	✓	onlyRole
	maxExchangeAmount	Public		-
	totalValue	Public		-
	collateralValueFromAmount	Public		-
	collateralAmountFromValue	Public		-
	allowCollateral	Public	✓	onlyRole
	disallowCollateral	Public	✓	onlyRole
	isCollateralSupported	Public		-
	listCollateral	Public		-
IRecoverable	Interface			
	recoverERC20	External	✓	-
	recoverETH	External	✓	-
AmoVault	Implementation	AccessControl, IRecoverable, IAmoVault, ReentrancyGuard		
		Public	✓	-
	withdrawCollateral	External	✓	onlyRole nonReentrant
	approveAmoManager	Public	✓	onlyRole
	recoverERC20	External	✓	onlyRole nonReentrant
	recoverETH	External	✓	onlyRole
	_isCollateral	Internal		

		External	Payable	-
AmoManager	Implementation	AccessControl		
		Public	✓	-
	allocateAmo	Public	✓	onlyRole
	deallocateAmo	Public	✓	onlyRole
	totalAmoSupply	Public		-
	totalCollateralValue	Public		-
	decreaseAmoSupply	Public	✓	onlyRole
	isAmoActive	Public		-
	enableAmoVault	Public	✓	onlyRole
	disableAmoVault	Public	✓	onlyRole
	removeAmoVault	Public	✓	onlyRole
	amoVaults	Public		-
IUniswapV3PoolState	Interface			
	slot0	External		-
	feeGrowthGlobal0X128	External		-
	feeGrowthGlobal1X128	External		-
	protocolFees	External		-
	liquidity	External		-
	ticks	External		-
	tickBitmap	External		-
	positions	External		-

	observations	External		-

Summary

dTRINITY dusd contracts implement a system for managing the issuance, redemption, and collateralization of dUSD stablecoins. This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope’s prior written consent. This report is not nor should be considered an “endorsement” or “disapproval” of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any “product” or “asset” created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope’s position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope’s goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io