Solidity Smart Contracts

dTRINITY

HALBORN

Report a Bug

Solidity Smart Contracts - dTRINITY

Prepared by: HALBORN

Last Updated 12/11/2024

Date of Engagement by: September 23rd, 2024 - October 4th, 2024

Summary

100% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
3	0	1	0	0	2

1. Introduction

dTRINITY engaged Halborn to conduct a security assessment on their smart contracts revisions beginning on September 23th,2024 and ending on October 04th,2024 . The security assessment was scoped to the smart contracts provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided two weeks for the engagement and assigned a full-time security engineer to evaluate the security of the smart contract.

The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that smart contract functions operate as intended.
- Identify potential security issues with the smart contracts.

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were completely addressed by the `dTRINITY team`. The main ones were the following:

- `Always report the actual price returned by the oracle.`
- `Implement CEI Pattern in AMO contracts.`
- `Implement API3 recommendations.`

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy regarding the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance code coverage and quickly identify items that do not follow the security best practices. The following phases and associated tools were used during the assessment:

- Research into architecture and purpose.
- Smart contract manual code review and walkthrough.
- Graphing out functionality and contract logic/connectivity/functions. (`solgraph`)
- Manual assessment of use and safety for the critical Solidity variables and functions in scope to identify any arithmetic related vulnerability classes.
- Manual testing by custom scripts.
- Scanning of solidity files for vulnerabilities, security hot-spots or bugs. (`MythX`)
- Static Analysis of security for scoped contract, and imported functions. (`Slither`)
- Testnet deployment. (`Brownie`, `Anvil`, `Foundry`)

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9

TABLE OF CONTENTS

SEVERITY	SCORE VALUE RANGE
Medium	4.5 - 6.9
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: solidity-contracts-preview

(b) Assessed Commit ID: 61c486b

(c) Items in scope:

- DexOracleWrapper
- OracleAggregator
- OracleAware
- AmoManager
- CollateralVault
- Issuer
- Redeemer
- UniV3AmoVault

Out-of-Scope: Non-specified contracts or libraries, third party dependencies and economic attacks.

REMEDIATION COMMIT ID:

- 6e0323d
- 5f27d42

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW
0	1	0	0

INFORMATIONAL
2

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
PRICE REPORTING FLAW IN ORACLEAGGREGATOR CONTRACT	HIGH	SOLVED - 11/08/2024
CEI PATTERN NOT RESPECTED IN AMO ALLOCATION FUNCTIONS	INFORMATIONAL	SOLVED - 10/21/2024
ORACLE SECURITY RECOMMENDATIONS WITH API3 INTEGRATION	INFORMATIONAL	SOLVED - 10/21/2024

7. FINDINGS & TECH DETAILS

7.1 PRICE REPORTING FLAW IN ORACLEAGGREGATOR CONTRACT

// HIGH

Description

The `OracleAggregator` contract contains a vulnerability in its `_getPriceInfo` function. When the `applyThresholding` flag is set for an asset and its price exceeds the `thresholdUsd`, the function artificially caps the reported price to 1 (`priceUnit`).

```
uint256 public immutable priceUnit;

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

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

function _getPriceInfo(
    address asset
) private view returns (uint256 price, bool isAlive) {
    address oracle = assetOracles[asset];
    if (oracle == address(0)) {
        revert OracleNotSet(asset);
    }

    (price, isAlive) = IOracleWrapper(oracle).getPriceInfo(asset);
```

```
    if (applyThresholding[asset] && price >= thresholdUsd) {
        return (priceUnit, true);
    }

    return (price, isAlive);
}
```

This flaw leads to undervaluation of assets when their prices exceed the threshold. It creates a discontinuity in pricing that is exploitable by malicious actors. This vulnerability enables arbitrage opportunities between this system and external markets using real prices.

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:M/Y:M/R:N/S:C (7.8)

Recommendation

It is recommended to remove the thresholding mechanism entirely. Always report the actual price returned by the oracle:

```
function _getPriceInfo(
    address asset
) private view returns (uint256 price, bool isAlive) {
    address oracle = assetOracles[asset];
    if (oracle == address(0)) {
        revert OracleNotSet(asset);
    }

    return IOracleWrapper(oracle).getPriceInfo(asset);
}
```

If handling extreme price movements is necessary, implement a more sophisticated mechanism that doesn't involve price capping. Consider using a moving average or introducing a delay for large price changes to mitigate the impact of short-term price spikes.

Remediation

SOLVED: The **dTRINITY team** removed the price threshold logic, so it's not an issue anymore.

Remediation Hash

https://github.com/dtrinity/solidity-contracts-preview/commit/6e0323d1f24e41be0b60a34b8a8cf81fe6d7b072

References

dtrinity/solidity-contracts-preview/contracts/dusd/OracleAggregator.sol#L171

7.2 CEI PATTERN NOT RESPECTED IN AMO ALLOCATION FUNCTIONS

// INFORMATIONAL

Description

The `AmoManager` contract's `allocateAmo` and `deallocateAmo` functions do not adhere to the Checks-Effects-Interactions (CEI) pattern. These functions perform external interactions before updating the contract's state, violating a crucial smart contract security principle:

```
// allocateAmo
require(dusd.transfer(amoVault, dusdAmount), "dUSD transfer failed")
totalAllocated += dusdAmount;

// deallocateAmo
require(dusd.transferFrom(amoVault, address(this), dusdAmount),"dUSD
totalAllocated -= dusdAmount;
```

The state variable `totalAllocated` is updated after the external call, which is contrary to best practices.

Due to the fact that `dusd` is created by `dTRINITY Team`, there is no risk, but it is better to follow CEI pattern: https://github.com/dtrinity/solidity-contracts-preview/blob/main/contracts/dusd/AmoManager.sol#L48

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

It is recommended to reorder the operations in both functions to follow the CEI pattern:
For `allocateAmo`:

```
totalAllocated += dusdAmount;
require(dusd.transfer(amoVault, dusdAmount), "dUSD transfer failed")
```

For `deallocateAmo`:

```
totalAllocated -= dusdAmount;
require(
    dusd.transferFrom(amoVault, address(this), dusdAmount),
    "dUSD transfer failed"
);
```

Remediation

SOLVED: CEI pattern is now respected.

Remediation Hash

https://github.com/dtrinity/solidity-contracts-preview/commit/5f27d42856bc253d8f6ceccc97727a3cb7e68f11

References

dtrinity/solidity-contracts-preview/contracts/dusd/AmoManager.sol#L48

7.3 ORACLE SECURITY RECOMMENDATIONS WITH API3 INTEGRATION

// INFORMATIONAL

Description

The current implementation of `DexOracleWrapper.sol` can be improved by leveraging API3's dAPI (decentralized API) system. API3 provides a robust oracle solution with built-in security features. However, a better implementation is crucial to maximize security benefits.

Score

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:N/R:N/S:C (0.0)

Recommendation

Recommended Implementation:
1. Use API3's IProxy Interface: Replace the current oracle implementation with API3's `IProxy` interface:
2. Implement Strict Validation: Add rigorous checks for the returned values:

```
import "@api3/contracts/api3-server-v1/proxies/interfaces/IProxy.sol

contract DexOracleWrapper is IOracleWrapper {
    IProxy public immutable proxy;
    address public proxyAddress;

    constructor(address _proxy) {
        proxy = IProxy(_proxy);
    }

    function getPriceInfo(address asset) external view returns (uint
```

```
        (int224 value, uint256 timestamp) = proxy.read();
        require(value > 0, "Invalid negative price");
        require(block.timestamp - timestamp <= 1 days, "Price data to
        price = uint256(value);
        isAlive = true;
        return (price, isAlive);
    }

     function setProxyAddress(address _proxyAddress) external onlyOwn
        proxyAddress = _proxyAddress;
        proxy = IProxy(_proxyAddress);
     }
}
```

Implementation Example

Remediation

SOLVED: The **dTRINITY team** is now doing the security checks within API3 integration.

Remediation Hash

https://github.com/dtrinity/solidity-contracts-preview/commit/5f27d42856bc253d8f6ceccc97727a3cb7e68f11

References

dtrinity/solidity-contracts-preview/contracts/dusd/wrapper/DexOracleWrapper.sol#L22

8. AUTOMATED TESTING

Halborn used automated testing techniques to enhance the coverage of certain areas of the smart contracts in scope. Among the tools used was Slither, a Solidity static analysis framework.

After Halborn verified the smart contracts in the repository and was able to compile them correctly into their abis and binary format, Slither was run against the contracts. This tool can statically verify mathematical relationships between Solidity variables to detect invalid or inconsistent usage of the contracts' APIs across the entire code-base.

```
➜  solidity-contracts-preview-61c486ba925a891eb6818c1626b8478e6a535fdf slither . --include-paths ./contracts/dusd --exclude-low --exclude-informational
'npx hardhat clean' running (wd: /Users/liliancariou/Desktop/Halborn/audits/solidity-contracts-preview-61c486ba925a891eb6818c1626b8478e6a535fdf)
'npx hardhat clean --global' running (wd: /Users/liliancariou/Desktop/Halborn/audits/solidity-contracts-preview-61c486ba925a891eb6818c1626b8478e6a535fdf)
Problem deserializing hardhat configuration, using defaults: Expecting value: line 1 column 1 (char 0)
'npx hardhat compile --force' running (wd: /Users/liliancariou/Desktop/Halborn/audits/solidity-contracts-preview-61c486ba925a891eb6818c1626b8478e6a535fdf)
INFO:Detectors:
AmoManager.deallocateAmo(address,uint256) (contracts/dusd/AmoManager.sol#75-97) uses arbitrary from in transferFrom: require(bool,string)(dusd.transferFrom(amoVault,address(this)
,dusdAmount),dUSD transfer failed) (contracts/dusd/AmoManager.sol#84-87)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#arbitrary-from-in-transferfrom
INFO:Detectors:
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) has bitwise-xor operator ^ instead of the exponentiation oper
ator **:
        - inv = (3 * denominator) ^ 2 (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#88)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
ISwapRouter is re-used:
        - ISwapRouter (contracts/dex/periphery/interfaces/ISwapRouter.sol#9-75)
        - ISwapRouter (contracts/dusd/dependencies/uniswap-v3/periphery/interfaces/ISwapRouter.sol#7-73)
        - ISwapRouter (contracts/lending/periphery/adapters/dswap/interfaces/ISwapRouter.sol#7-73)
TransferHelper is re-used:
        - TransferHelper (contracts/dex/core/libraries/TransferHelper.sol#8-23)
        - TransferHelper (contracts/dusd/dependencies/uniswap-v3/core/libraries/TransferHelper.sol#5-48)
        - TransferHelper (contracts/lending/periphery/adapters/dswap/TransferHelper.sol#6-71)
FixedPoint96 is re-used:
        - FixedPoint96 (contracts/dex/core/libraries/FixedPoint96.sol#7-10)
        - FixedPoint96 (contracts/dusd/dependencies/uniswap-v3/core/libraries/FixedPoint96.sol#7-10)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#name-reused
```

```
INFO:Detectors:
CollateralVault.maxExchangeAmount(uint256,address,address) (contracts/dusd/CollateralVault.sol#214-234) performs a multiplication on the result of a division:
        - fromCollateralValue = (fromCollateralPrice * fromCollateralAmount) / (10 ** fromCollateralDecimals) (contracts/dusd/CollateralVault.sol#226-227)
        - toCollateralAmount = (fromCollateralValue * (10 ** toCollateralDecimals)) / toCollateralPrice (contracts/dusd/CollateralVault.sol#229-231)
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) performs a multiplication on the result of a division:
        - denominator = denominator / twos (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#68)
        - inv = (3 * denominator) ^ 2 (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#88)
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) performs a multiplication on the result of a division:
        - denominator = denominator / twos (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#68)
        - inv *= 2 - denominator * inv (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#92)
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) performs a multiplication on the result of a division:
        - denominator = denominator / twos (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#68)
        - inv *= 2 - denominator * inv (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#93)
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) performs a multiplication on the result of a division:
        - denominator = denominator / twos (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#68)
        - inv *= 2 - denominator * inv (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#94)
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) performs a multiplication on the result of a division:
        - denominator = denominator / twos (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#68)
        - inv *= 2 - denominator * inv (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#95)
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) performs a multiplication on the result of a division:
        - denominator = denominator / twos (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#68)
        - inv *= 2 - denominator * inv (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#96)
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) performs a multiplication on the result of a division:
        - denominator = denominator / twos (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#68)
        - inv *= 2 - denominator * inv (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#97)
FullMath.mulDiv(uint256,uint256,uint256) (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#14-108) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (contracts/dusd/dependencies/uniswap-v3/core/libraries/FullMath.sol#73)
```

```
INFO:Detectors:
Loop condition i < _positionsArray.length (contracts/dusd/UniV3AmoVault.sol#228) should use cached array length instead of referencing `length` member of the storage array.
 Loop condition i < _positionsArray.length (contracts/dusd/UniV3AmoVault.sol#265) should use cached array length instead of referencing `length` member of the storage array.
 Loop condition i < _positionsArray.length (contracts/dusd/UniV3AmoVault.sol#89) should use cached array length instead of referencing `length` member of the storage array.
 Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#cache-array-length
INFO:Detectors:
DexOracleWrapper.priceOracle (contracts/dusd/wrapper/DexOracleWrapper.sol#8) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Detectors:
AmoManager.dusd (contracts/dusd/AmoManager.sol#17) should be immutable
Issuer.dusd (contracts/dusd/Issuer.sol#17) should be immutable
Redeemer.dusd (contracts/dusd/Redeemer.sol#15) should be immutable
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-immutable
INFO:Slither:. analyzed (828 contracts with 58 detectors), 50 result(s) found
```

All issues identified by `Slither` were proved to be false positives or have been added to the issue list in this report.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.