// Private Smart Contract Security Assessment 04.24.2024 - 05.07.2024

Protocol V2
Anzen Finance

HALBORN

Protocol V2 - Anzen Finance

Prepared by: HALBORN

Last Updated 05/16/2024

Date of Engagement by: April 24th, 2024 - May 7th, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
27	0	2	1	2	22

TABLE OF CONTENTS

1. Introduction

The `Anzen Finance` team engaged Halborn to conduct a security assessment on their smart contracts beginning on *04/24/2024* and ending on *05/07/2024*. The security assessment was scoped to the smart contracts provided in the GitHub repository. Commit hashes and further details can be found in the Scope section of this report.

2. Assessment Summary

Halborn was provided 2 weeks for the engagement and assigned 1 full-time security engineer to review the security of the smart contracts in scope. The engineer is a blockchain and smart contract security expert with advanced penetration testing and smart contract hacking skills, and deep knowledge of multiple blockchain protocols.

The purpose of the assessment is to:

- Identify potential security issues within the smart contracts.
- Ensure that smart contract functionality operates as intended.

In summary, Halborn identified 27 security findings that were mostly addressed by the `Anzen Finance team`.

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of the code and can 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 (`Foundry`).

Out-Of-Scope

- External libraries and financial-related attacks.
- New features/implementations after/with the **remediation commit IDs**.

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

EXPLOITABILIY METRIC (m_e)	METRIC VALUE	NUMERICAL VALUE
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
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
Medium	4.5 - 6.9

SEVERITY	SCORE VALUE RANGE
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: protocol-v2

(b) Assessed Commit ID: 7f3e5ce

(c) Items in scope:

- src/Canto_childUSDz.sol
- src/interfaces/ISPCTPool.sol
- src/interfaces/IsUSDz.sol
- src/interfaces/ISPCTPriceOracle.sol
- src/interfaces/ITurnstile.sol
- src/interfaces/IUSDz.sol
- src/interfaces/IchildUSDz.sol
- src/interfaces/IUSDzFlat.sol
- src/interfaces/IUSDzPriceOracle.sol
- src/sUSDz.sol
- src/USDz.sol
- src/USDzFlat.sol
- src/SPCTPriceOracle.sol
- src/vault.sol
- src/utils/SafeMath.sol
- src/childUSDz.sol
- src/USDzPriceOracle.sol
- src/SPCTPool.sol

Out-of-Scope:

REMEDIATION COMMIT ID:

- 1f4c9be1f4c9be
- 61fac2861fac28
- 1b020391b02039
- 254968f254968f
- 9c749aa9c749aa
- 783f640783f640
- 759210175921O1
- c4b57e2c4b57e2

- 59114835911483
- 012a9b1012a9b1
- 194032019403201
- d47e109d47e109
- 6e543ae6e543ae
- 9b375189b37518
- a3e40a7a3e40a7

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

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	2	1	2	22

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-24 - BLACKLISTED ADDRESSES CAN STILL SEND USDZ AND SUSDZ	High	SOLVED - 05/13/2024
HAL-27 - INCORRECT STAKE CALCULATIONS AFTER ADDING YIELD	High	SOLVED - 05/13/2024
HAL-23 - CHAINLINK'S ORACLE MIGHT RETURN STALE OR INCORRECT DATA	Medium	SOLVED - 05/13/2024
HAL-25 - SINGLE-STEP OWNERSHIP TRANSFER PROCESS	Low	SOLVED - 05/13/2024
HAL-03 - INEFFECTIVE PAUSING MECHANISM	Low	PARTIALLY SOLVED - 05/13/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - UNSAFE ERC20 OPERATIONS USED	Informational	PARTIALLY SOLVED - 05/13/2024
HAL-26 - OWNER CAN RENOUNCE OWNERSHIP	Informational	ACKNOWLEDGED
HAL-02 - LACK OF INPUT VALIDATION	Informational	PARTIALLY SOLVED - 05/13/2024
HAL-04 - INCONSISTENT REQUIRE STATEMENTS	Informational	SOLVED - 05/13/2024
HAL-05 - IMPORTANT VALUES NOT EXPLICITLY SET IN THE CONSTRUCTOR	Informational	ACKNOWLEDGED
HAL-21 - INCOMPLETE NATSPEC DOCUMENTATION AND TEST COVERAGE	Informational	ACKNOWLEDGED
HAL-09 - USE OF REVERT STRINGS INSTEAD OF CUSTOM ERRORS	Informational	ACKNOWLEDGED
HAL-06 - WRONG ORACLE FEED ADDRESSES	Informational	ACKNOWLEDGED
HAL-07 - LACK OF EVENT EMISSION	Informational	ACKNOWLEDGED
HAL-08 - EVENTS MISSING INDEXED FIELDS	Informational	PARTIALLY SOLVED - 05/13/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-10 - INTERFACES DO NOT MATCH THE SOURCE CODE	Informational	ACKNOWLEDGED
HAL-11 - UNUSED IMPORTS	Informational	SOLVED - 05/13/2024
HAL-12 - DEPRECATED LIBRARIES IN USE	Informational	ACKNOWLEDGED
HAL-13 - FLOATING PRAGMA IN SOLIDITY CONTRACTS	Informational	PARTIALLY SOLVED - 05/13/2024
HAL-14 - GLOBAL STATE VARIABLE CAN BE A CONSTANT	Informational	SOLVED - 05/13/2024
HAL-15 - CONSTANTS VISIBILITY NOT SET	Informational	SOLVED - 05/13/2024
HAL-16 - MISLEADING OR OUTDATED COMMENTS	Informational	SOLVED - 05/13/2024
HAL-17 - MULTIPLE TYPOS IN COMMENTS AND ERROR MESSAGES	Informational	SOLVED - 05/13/2024
HAL-18 - NAMING CONVENTION NOT FOLLOWED	Informational	PARTIALLY SOLVED - 05/13/2024
HAL-19 - TAUTOLOGY AND BOOLEAN EQUALITIES	Informational	PARTIALLY SOLVED - 05/13/2024

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-20 - INTERNAL FUNCTIONS CALLED ONLY ONCE CAN BE INLINED	Informational	ACKNOWLEDGED
HAL-22 - UNUSED EVENTS	Informational	SOLVED - 05/13/2024

7. FINDINGS & TECH DETAILS

7.1 (HAL-24) BLACKLISTED ADDRESSES CAN STILL SEND USDZ AND SUSDZ

// HIGH

Description

Upon inspection, it was noted that blacklisted accounts retain the ability to send both USDz and sUSDz tokens without restriction. The issue stems from the `_update()` function, present in both USDz and sUSDz contracts, responsible for updating token balances. While this function appropriately prevents blacklisted addresses from receiving tokens, it lacks a corresponding check to restrict these addresses from initiating token transfers.
Notice below that the `_update()` function is only preventing blacklisted addresses from receiving tokens, but not from sending them:

```
/**
 * @notice Moves `_amount` tokens from `_sender` to `_recipient`.
 * Emits a `Transfer` event.
 */
function _update(address _sender, address _recipient, uint256 _amount)
internal override {
    require(!_blacklist[_recipient], "RECIPIENT_IN_BLACKLIST");

    super._update(_sender, _recipient, _amount);
}
```

Proof of Concept

The following `Foundry` test was used in order to prove the aforementioned issue:

```
function testFailUSDztransferBlacklist() public {
    _mockDeposits();

    assertEq(usdz.balanceOf(user), 1e20);
    assertEq(usdz.balanceOf(user2), 2e20);

    vm.prank(user);
    usdz.transfer(user2, 1);
    assertEq(usdz.balanceOf(user), 1e20 - 1);
    assertEq(usdz.balanceOf(user2), 2e20 + 1);
```

```
    // Adding user to blacklist
    testUSDzaddToBlacklist();

    vm.expectRevert("RECIPIENT_IN_BLACKLIST");
    vm.prank(user2);
    usdz.transfer(user, 1);
    assertEq(usdz.balanceOf(user), 1e20 - 1);
    assertEq(usdz.balanceOf(user2), 2e20 + 1);

    vm.expectRevert();
    vm.prank(user);
    usdz.transfer(user2, 1);
    assertEq(usdz.balanceOf(user), 1e20 - 1);
    assertEq(usdz.balanceOf(user2), 2e20 + 1);
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:M/A:M/D:M/Y:N/R:N/S:U (7.5)

Recommendation

To address this vulnerability, it is necessary to enhance the `_update()` function to include a check that prohibits blacklisted accounts from sending tokens as well. By extending the validation logic to cover outgoing transactions, the protocol can effectively prevent blacklisted addresses from participating in token transfers altogether, thereby upholding the integrity and security of the USDz and sUSDz contracts.

```
/**
 * @notice Moves `_amount` tokens from `_sender` to `_recipient`.
 * Emits a `Transfer` event.
 */
function _update(address _sender, address _recipient, uint256 _amount)
internal override {
    require(!_blacklist[_recipient], "RECIPIENT_IN_BLACKLIST");
    require(!_blacklist[_sender], "SENDER_IN_BLACKLIST");

    super._update(_sender, _recipient, _amount);
}
```

Remediation Plan

SOLVED: The **Anzen team** solved the issue by adding a check to the including a check that prohibits blacklisted accounts from sending tokens.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/1f4c9be633a6a2733431d999132b548117a10011

7.2 (HAL-27) INCORRECT STAKE CALCULATIONS AFTER ADDING YIELD

// HIGH

Description

During the security assessment, it was found that the `sUSDz` contract was not properly calculating staked amounts after some yield was added using the `addYield()` function.
Specifically, the contract was not properly updating the `pooledUSDz` state variable. This was because the `totalAssets()` function, when there was some vesting unlocked, was returning the stored `pooledUSDz` `plus` the unlocked vesting amount. However, this was not stored in the globals state variable.

```
/**
 * @notice Total vested USDe in this contract.
 * @dev To prevent ERC4626 Inflation Attacks. We use pooledUSDz to
calculate totalAssets instead of balanceOf().
 *
 * https://blog.openzeppelin.com/a-novel-defense-against-erc4626-inflation-
attacks
 *
 */
function totalAssets() public view override returns (uint256) {
    // If all vested
    uint256 timeGap = block.timestamp.sub(lastDistributionTime);
    if (timeGap >= vestingPeriod) {
        if (vestingAmount != 0) {
            return pooledUSDz.add(vestingAmount);
        }
        return pooledUSDz;
    } else {
        uint256 unvestedAmount =
((vestingPeriod.sub(timeGap)).mul(vestingAmount)).div(vestingPeriod);
        uint256 vestedAmount = vestingAmount.sub(unvestedAmount);
        return pooledUSDz.add(vestedAmount);
    }
}
```

See the PoC section for a miscalculation example.

Proof of Concept

The following `Foundry` test was used in order to prove the aforementioned issue:

```
function testDepositAddYieldAndUnstake2Users() public {
    uint256 depositAmount = 1e20; // User to deposit 100 USDz
    uint256 depositAmount2 = 2e20; // User2 to deposit 200 USDz
    uint256 yieldAmount = 1e20; // 100 USDz yield
    uint256 initialUSDzBalance = usdzToken.balanceOf(user);
    uint256 initialUSDzBalance2 = usdzToken.balanceOf(user2);
    uint256 initialStakedUSDzBalance = vault.balanceOf(user);
    uint256 initialStakedUSDzBalance2 = vault.balanceOf(user2);

    // Logging the ratio
    console.log("1_000 assets to shares:", vault.convertToShares(1_000));

    testDeposit();
    assertEq(
        vault.balanceOf(user),
        initialStakedUSDzBalance + depositAmount,
        "User balance of sUSDz should be equal to initial deposit"
    );
    assertEq(
        vault.balanceOf(user2),
        initialStakedUSDzBalance2 + depositAmount2,
        "User2 balance of sUSDz should be equal to initial deposit"
    );

    // add yieldAmount to yield
    vm.prank(yieldManager);
    vault.addYield(yieldAmount);

    assertEq(
        vault.balanceOf(user),
        initialStakedUSDzBalance + depositAmount,
        "User balance of sUSDz should be unchanged right after yield is
added"
    );
    assertEq(
        vault.balanceOf(user2),
        initialStakedUSDzBalance2 + depositAmount2,
        "User2 balance of sUSDz should be unchanged right after yield is
added"
    );

    vm.warp(block.timestamp + vault.vestingPeriod());
```

```
    console.log("1_000 assets to shares changed:",
vault.convertToShares(1_000));

    vm.startPrank(user);
    vault.deposit(depositAmount, user);
    assertEq(
        vault.balanceOf(user),
        initialStakedUSDzBalance + depositAmount +
vault.convertToShares(depositAmount),
        "User balance of sUSDz should've increased by depositAmount"
    );

    // Cooldown all user assets and unstake
    vm.startPrank(user);
    vault.CDShares(vault.balanceOf(user));
    vm.warp(block.timestamp + vault.CDPeriod() + 1);

    console.log("1_000 assets to shares:", vault.convertToShares(1_000));
    assertEq(vault.balanceOf(user), 0, "User does not have sUSDz anymore
");
    assertEq(
        usdzToken.balanceOf(address(vault.flat())),
        depositAmount * 2 + yieldAmount / 3,
        "User's USDz are being help by USDzFlat"
    );
    assertEq(
        vault.balanceOf(user2),
        initialStakedUSDzBalance2 + depositAmount2,
        "User2 balance of sUSDz should be unchanged right after yield is
added"
    );

    vm.startPrank(user);
    vault.unstake();
    assertEq(
        usdzToken.balanceOf(user),
        initialUSDzBalance + yieldAmount / 3,
        "User should receive deposit back plus yield"
    );
    assertEq(
        usdzToken.balanceOf(user2),
        initialUSDzBalance2 - depositAmount2,
        "User2 should have the initial minus the deposit, it hasn't unstake
yet"
```

```
    );
    assertEq(vault.balanceOf(user), 0, "User does not have sUSDz anymore");
    assertEq(
        vault.balanceOf(user2),
        initialStakedUSDzBalance2 + depositAmount2,
        "User2 balance of sUSDz should be unchanged"
    );

    console.log("1_000 assets to shares:", vault.convertToShares(1_000));

    // Now user2
    // Cooldown all user2 assets and unstake
    // add yieldAmount to yield
    vm.stopPrank();
    vm.prank(yieldManager);
    vault.addYield(yieldAmount);
    vm.warp(block.timestamp + vault.vestingPeriod());

    assertEq(vault.balanceOf(user), 0, "User balance of sUSDz still has 0
sUSDz");
    assertEq(
        vault.balanceOf(user2),
        initialStakedUSDzBalance2 + depositAmount2,
        "User2 balance of sUSDz should be unchanged right after yield is
added"
    );

    vm.startPrank(user2);
    vault.CDShares(vault.balanceOf(user2));
    vm.warp(block.timestamp + vault.CDPeriod() + 1);

    console.log("1_000 assets to shares:", vault.convertToShares(1_000));

    vault.unstake();
    assertEq(
        usdzToken.balanceOf(user2),
        initialUSDzBalance2 + 2 * yieldAmount / 3 + yieldAmount,
        "User2 should receive deposit back plus 2/3 of the 1st yield +
whole 2nd yield"
    );

    console.log("1_000 assets to shares:", vault.convertToShares(1_000));

    console.log("sUSDz user: ", vault.balanceOf(user));
```

```
    console.log("sUSDz user2: ", vault.balanceOf(user2));
    console.log("sUSDz totalSupply: ", vault.totalSupply());
}
```

To run it, just execute the following forge command:

```
forge test --mt testDepositAddYieldAndUnstake2Users -vvvv
```

Observe that the test fails with an arithmetic error due to the second call to `CDShares()`. This is because, as indicated in the definition section, the `pooledUSDz` global state variable was not properly updated and the attempt to call the internal function `_withdraw` reverted.

```
    ├─ [49321] sUSDz::CDShares(20000000000000000000 [2e20])
    │   └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)
    └─ ← [Revert] panic: arithmetic underflow or overflow (0x11)
Suite result: FAILED. 0 passed; 1 failed; 0 skipped; finished in 8.74ms (1.13ms CPU time)
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:N/D:N/Y:H/R:N/S:U (7.5)

Recommendation

It is advisable to conduct a thorough review of the `addYield()` function to ensure that the introduction of additional yield does not disrupt the staking calculations. Specifically, make sure that any new yield amount is accurately reflected in the `pooledUSDz` and `vestingAmount` global state variables and that the unvested amount is properly calculated.

Remediation Plan

SOLVED: The **Anzen team** solved the issue by revising the `addYield()`, `totalAssets()` and `getUnvestedAmount()` functions to ensure that the new yield is accurately incorporated into the calculations.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/61fac28f37102f2578cc00a84996ec8e52ce05fe

7.3 (HAL-23) CHAINLINK'S ORACLE MIGHT RETURN STALE OR INCORRECT DATA

// MEDIUM

Description

The `USDzPriceOracle` and `SPCTPriceOracle` contracts use Chainlink as their price oracle. When requesting the price via the `getPrice()` function, the contracts query Chainlink for the underlying token price using the `latestRoundData()` function. This function returns `uint80 roundId`, `int256 answer`, `uint256 startedAt`, `uint256 updatedAt` and `uint80 answeredInRound`. `roundId` denotes the identifier of the most recent update round, `answer` is the price of the asset, `startedAt` is the timestamp at which the round started and `updatedAt` is the timestamp at which the feed was updated.
The `getPrice()` function does not check if the feed was updated at the most recent round nor does it verify that `startedAt` (timestamp at which the round started) is not 0, and this can result in accepting stale data which may threaten the stability of the protocol in a volatile market.

```
function getPrice() external view returns (uint256) {
    (, int256 price,, uint256 updatedAt,) = priceFeed.latestRoundData();
    require(price > 0, "invalid price");
    require(block.timestamp - updatedAt <= heartbeat, "stale price");

    return uint256(price) * 10 ** (18 - uint256(priceFeed.decimals()));
}
```

Finally, be aware that, in the unlikely event of `priceFeed.decimals()` returning a value greater than 18, the whole function would break due to the last operation underflowing (`18 - uint256(priceFeed.decimals())`).

BVSS

AO:A/AC:L/AX:M/C:N/I:N/A:H/D:N/Y:N/R:N/S:U (5.0)

Recommendation

Consider improving checks for stale data. Specifically, make sure `startedAt` is not 0 (which would indicate that the round has not completed) and that `answeredInRound` is greater or equal to `roundId`.
Finally, consider adding a check for `priceFeed.decimals()` to make sure it is lower than or equal to 18.

```
function getPrice() external view returns (uint256) {
    (uint80 roundId, int256 price, uint256 startedAt, uint256 updatedAt,
uint80 answeredInRound) = priceFeed.latestRoundData();
    require(price > 0, "invalid price");
    require(block.timestamp - updatedAt <= heartbeat, "stale price");
```

```
    require(startedAt > 0, "Round not complete");
    require(answeredInRound >= roundId, "stale price");

    uint256 priceFeedDecimals = uint256(priceFeed.decimals();
    require(priceFeedDecimals <= 18, "Incorrect decimals");

    return uint256(price) * 10 ** (18 - uint256(priceFeed.decimals()));
}
```

Remediation Plan

SOLVED: The **Anzen team** solved the issue by implementing the proposed improvement.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/1b0203995bf643801a18ff39e4f2d69aec7b3400

7.4 (HAL-25) SINGLE-STEP OWNERSHIP TRANSFER PROCESS

// LOW

Description

It was identified that the `USDz` contract inherited from OpenZeppelin's `Ownable` library. Ownership of the contracts that are inherited from the `Ownable` module can be lost, as the ownership is transferred in a single-step process. The address that the ownership is changed to should be verified to be active or willing to act as the `owner`. `Ownable2Step` is safer than `Ownable` for smart contracts because the owner cannot accidentally transfer smart contract ownership to a mistyped address. Rather than directly transferring to the new owner, the transfer only completes when the new owner accepts ownership.

Proof of Concept

The following `Foundry` test was used in order to prove the aforementioned issue:

```
function testUSDztransferOwnership() public {
  usdz.transferOwnership(address(1));
}
```

To run it, just execute the following `forge` command:

```
forge test --mt testUSDztransferOwnership -vvvv
```

Observe that the test passes and the new owner is now the invalid `address(1)`.

```
[PASS] testUSDztransferOwnership() (gas: 10355)
Traces:
  [10355] HalbornUSDzTest::testUSDztransferOwnership
    ├─ [7240] USDz::transferOwnership(0x0000000000000000000000000000000000000001)
    │   ├─ emit OwnershipTransferred(previousOwner: HalbornUSDzTest:
[0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], newOwner:
0x0000000000000000000000000000000000000001)
    │   └─ ← [Stop]
    └─ ← [Stop]
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:M/D:N/Y:N/R:P/S:U (2.5)

Recommendation

To mitigate the risks associated with single-step ownership transitions and enhance contract security, it is recommended to adopt a two-step ownership transition mechanism, such as OpenZeppelin's `Ownable2Step`. This approach introduces an additional step in the ownership transfer process, requiring the new owner to accept ownership before the transition is finalized. The process typically involves the current owner calling a function to nominate a new owner, and the nominee then calling another function to accept ownership.
Implementing `Ownable2Step` provides several benefits:
1. **Reduces Risk of Accidental Loss of Ownership**: By requiring explicit acceptance of ownership, the risk of accidentally transferring ownership to an incorrect or zero address is significantly reduced.
2. **Enhanced Security**: It adds another layer of security by ensuring that the new owner is prepared and willing to take over the responsibilities associated with contract ownership.
3. **Flexibility in Ownership Transitions**: Allows for a smoother transition of ownership, as the nominee has the opportunity to prepare for the acceptance of their new role.
By adopting `Ownable2Step`, contract administrators can ensure a more secure and controlled process for transferring ownership, safeguarding against the risks associated with accidental or unauthorized ownership changes.
It was noted that the affected contracts use the `Ownable` library though the `LayerZeroLabs`'s `OFT` contracts. If directly changing from `Ownable` to `Ownable2Step` is not possible, consider overriding the affected functions to implement a two-step ownership transfer.

Remediation Plan

SOLVED: The **Anzen team** solved the issue by adding the logic to implement a two-step ownership transfer. While it was identified that the used logic was extracted from the well-known OpenZeppelin library, it is also recommended to consider adding checks for cases for when the contract is paused or for addresses that are blacklisted. Some projects may not want contracts to be able to modify the owner address while paused or propose owners in the blacklisted list. Note that this is a consideration the project should deliberate on.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/254968f562d138e3fa6f2c86d809c41552312138

7.5 (HAL-03) INEFFECTIVE PAUSING MECHANISM

// LOW

Description

In the `USDz` contract, the `checkCollateralRate()` modifier is calling the `_checkCollateralRate()` internal function which, if the price returned by the oracle is less than 1, it pauses the contract and reverts with the `UNDERCOLLATERAL_RATE,SMART_CONTRACT_IS_PAUSED_NOW` error message. See:

```
modifier checkCollateralRate() {
  _checkCollateralRate();
  _;
}

/**
 * @notice Check collateral rate.
 */
function _checkCollateralRate() internal {
  if (oracle.getPrice() / 1e18 < collateralRate) {
      _pause();
      revert("UNDER_COLLATERAL_RATE,SMART_CONTRACT_IS_PAUSED_NOW");
  }
}
```

The modifier above is being used by the functions `deposit()`, `depositBySPCT()`, `redeem()` and `redeemBackSPCT()` to make sure the contract is paused immediately if the price falls below `collateralRate`. However, due to the nature of EVM transactions, the contract remains unpaused due to the `revert()` at the end.

Proof of Concept

```
function testUSDzcheckCollateralRateNotPausing() public {
    // Price decreases
    oracle.setPrice(1e12);

    // Cannot deposit due to under collateralization,
    // the USDz contract indicates that it has paused
    vm.expectRevert("UNDER_COLLATERAL_RATE,SMART_CONTRACT_IS_PAUSED_NOW");
    vm.prank(user);
    usdz.deposit(100e6);

    // According to the message above, USDz should be paused now
```

```
    // However, it isn't because the transaction reverted
    assertFalse(usdz.paused());
}
```

BVSS

AO:A/AC:L/AX:L/C:N/I:H/A:L/D:N/Y:N/R:F/S:U (2.0)

Recommendation

To mitigate this issue, it's recommended to revise the pausing mechanism to ensure effective contract pausing upon detecting undercollateralization. This may involve redesigning the modifier to properly handle pausing functionalities.

Remediation Plan

PARTIALLY SOLVED: The **Anzen team** solved the issue by removing the misleading call to the `_pause()` function. However, this finding has been marked as `Partially Solved` instead of `Solved` because the error message when undercollateralized is still giving the wrong information that the contract is now paused: `"UNDER_COLLATERAL_RATE,SMART_CONTRACT_IS_PAUSED_NOW"`.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/254968f562d138e3fa6f2c86d809c41552312138

7.6 (HAL-01) UNSAFE ERC20 OPERATIONS USED

// INFORMATIONAL

Description

Even though the contracts in scope are importing the OpenZeppelin's `SafeERC20`'s library, they are not always making use of its secure functions (`safeTransfer()`, `safeTransferFrom()` , `safeIncreaseAllowance()` and `safeDecreaseAllowance()`) and use the insecure versions instead (`transfer()`, `transferFrom()` and `approve()`). Notice that, in some cases the insecure function is not even checking the returned value, see the example below using the insecure `transferFrom()` and not checking its returned value:

```
function deposit(uint256 _amount) external whenNotPaused {
  require(_amount > 0, "DEPOSIT_AMOUNT_IS_ZERO");

  usdc.transferFrom(msg.sender, address(this), _amount);
```

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:P/S:U (1.3)

Recommendation

Use OpenZeppelin's `SafeERC20`'s `safeTransfer()`, `safeTransferFrom()` , `safeIncreaseAllowance()` and `safeDecreaseAllowance()` instead `transfer()`, `transferFrom()` and `approve()`.

Remediation Plan

PARTIALLY SOLVED: The **Anzen team** fixed some occurrences of unsafe function calls not checking the returned values. However, there are still some instances using unsafe calls like the `deposit()` function of `SPCTPool` which is using `transferFrom()`.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/9c749aa914d61b13285db1c2741c5f1d0e343da2

7.7 (HAL-26) OWNER CAN RENOUNCE OWNERSHIP

// INFORMATIONAL

Description

It was identified that the `USDz` contract inherited from OpenZeppelin's `Ownable` library. In the `Ownable` contracts, the `renounceOwnership()` function is used to renounce the `Owner` permission. Renouncing ownership before transferring would result in the contract having no owner, eliminating the ability to call privileged functions.

```
/**
 * @dev Leaves the contract without owner. It will not be possible to call
 * `onlyOwner` functions. Can only be called by the current owner.
 *
 * NOTE: Renouncing ownership will leave the contract without an owner,
 * thereby disabling any functionality that is only available to the owner.
 */
function renounceOwnership() public virtual onlyOwner {
    _transferOwnership(address(0));
}
```

Proof of Concept

The following `Foundry` test was used in order to prove the aforementioned issue:

```
function testUSDzrenounceOwnership() public {
  usdz.renounceOwnership();
}
```

To run it, just execute the following `forge` command:

```
forge test --mt testUSDzrenounceOwnership -vvvv
```

Observe that the test passes and the ownership has been renounced (`address(0)`).

```
[PASS] testUSDzrenounceOwnership() (gas: 5291)
Traces:
  [8073] HalbornUSDzTest::testUSDzrenounceOwnership()
    ├─ [7052] USDz::renounceOwnership()
    │   ├─ emit OwnershipTransferred(previousOwner: HalbornUSDzTest:
[0x7FA9385bE102ac3EAc297483Dd6233D62b3e1496], newOwner:
0x0000000000000000000000000000000000000000)
```

```
    │   └─ ← [Stop]
    └─ ← [Stop]
```

BVSS

AO:A/AC:L/AX:L/C:N/I:N/A:L/D:N/Y:N/R:P/S:U (1.3)

Recommendation

It is recommended that the Owner cannot call `renounceOwnership()` without first transferring ownership to another address. In addition, if a multi-signature wallet is used, the call to the `renounceOwnership()` function should be confirmed for two or more users.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.8 (HAL-02) LACK OF INPUT VALIDATION

// INFORMATIONAL

Description

During the security assessment, it was found that critical parameters could be set to undesired values such as `address(0)` or unrealistic numbers. For instance:

1. **Constructor Zero Address Checks**: Some constructors of the project lack validation to ensure that zero addresses are not accepted for essential parameters. This omission could lead to unexpected behavior due to non-initialized values.
2. **Constructor Unbounded Values**: Apart from addresses, some of the constructors contained integer values that were not checked against valid ranges. For example, on `sUSDz`, any cooldown period (`_CDPeriod`) would be accepted even though not any value may be desirable.
3. **Setter Functions Validation**: Furthermore, even some setters did not have address 0 or boundary checks either. Notice a `newPeriod` or `0` or `type(uint256).max` could be set using the `setNewVestingPeriod()` function of `sUSDz`:

```
function setNewVestingPeriod(uint256 newPeriod) external
onlyRole(POOL_MANAGER_ROLE) {
  vestingPeriod = newPeriod;
  emit VestingPeriodChanged(newPeriod, block.timestamp);
}
```

BVSS

AO:S/AC:M/AX:M/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (0.7)

Recommendation

To address these issues, consider implementing input validation checks to ensure that zero addresses or unwanted values are not accepted for essential parameters.

Remediation Plan

PARTIALLY SOLVED: The **Anzen team** partially solved the issue by implemented a valid check in the `SUSDz` constructor. However, there are still other functions missing range or address validations.
Finally, it was noted that one of the added fixes is not provideiding a useful check. This is because it is not properly checking any range value. See the new require statement in the `setNewVestingPeriod(uint256 newPeriod)` function:

```
require(newPeriod > 0 && newPeriod < type(uint256).max,
"SHOULD_BE_LESS_THAN_UINT256_MAX_AND_GREATER_THAN_ZERO");
```

Notice that verifying if the `newPeriod` value is between the values 0 and `type(uint256).max` is like not doing any verification at all, as all `uint256` fall into that range.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/783f6403428ae6f2336aede93bb71d63fceb4380

7.9 (HAL-04) INCONSISTENT REQUIRE STATEMENTS

// INFORMATIONAL

Description

The `USDz` and `SPCTPool` contracts contained several `require()` statements like the following:

```
require(newMintFeeRate <= maxMintFeeRate, "SHOULD_BE_LESS_THAN_1P");
```

It was observed that, even though the error message says `SHOULD_BE_LESS_THAN_1P`, the value could actually be equal to 1% (`maxMintFeeRate`).

BVSS

AO:S/AC:M/AX:M/C:N/I:H/A:N/D:N/Y:N/R:N/S:U (0.7)

Recommendation

In order to improve consistency, consider adapting the require statements to:

```
require(newMintFeeRate < maxMintFeeRate, "SHOULD_BE_LESS_THAN_1P");
```

or

```
require(newMintFeeRate <= maxMintFeeRate, "SHOULD_BE_LESS_THAN_OR_EQUAL_TO_1P");
```

Remediation Plan

SOLVED: The **Anzen team** solved the issue by adjusting the error message to the actual behaviour of the condition.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/7592101f6fff002c0ef975ae1d02c1bea31a1895

7.10 (HAL-05) IMPORTANT VALUES NOT EXPLICITLY SET IN THE CONSTRUCTOR

// INFORMATIONAL

Description

Essential contract settings like mint fee rates, redeem fee rates and treasury addresses are not configured in the constructors of the affected contracts. This may break certain functionalities if those values are partially set afterwards.
For instance, in the USDz contract, if `mintFeeRate` is set without a corresponding treasury address, functionalities like `deposit()` and `depositBySPCT()` may stop working.

BVSS

AO:S/AC:L/AX:L/C:N/I:H/A:H/D:H/Y:H/R:F/S:U (0.7)

Recommendation

Though the risk associated with this finding is low, it is advisable to enhance contract robustness by including explicit initializations for mint fees, redeem fees, and treasury addresses (if applicable) in the constructor. This preventive measure ensures a complete configuration during contract deployment, minimizing the possibility of operational disruptions due to partially set values.
Additionally, addressing this issue during deployment could simplify contract management and maintenance, contributing to a more streamlined and reliable operational framework.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.11 (HAL-21) INCOMPLETE NATSPEC DOCUMENTATION AND TEST COVERAGE

// INFORMATIONAL

Description

At least all `public` and `external` functions that are not `view` or `pure` should have `NatSpec` comments.
During the security review, it was observed that the multiple publicly-accessible functions did not have any `NatSpec` comments.
Moreover, the test suite's coverage is insufficient, representing a less-than-ideal practice and failing to ensure thorough validation of function behaviors and overall contract functionality.

BVSS

AO:A/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:F/S:U (0.6)

Recommendation

Consider adding `NatSpec` documentation to all functions, specially those that are publicly-accessible. Additionally, expanding the test suite's coverage is essential to effectively validate function functionalities and ensure thorough contract testing. Enhancing the test suite represents a best practice, instilling confidence in contract behavior, mitigating the risk of undetected bugs, and streamlining the development process. By addressing these areas, the project can foster transparency, reliability, and robustness, thereby enhancing its overall quality and user experience.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.12 (HAL-09) USE OF REVERT STRINGS INSTEAD OF CUSTOM ERRORS

// INFORMATIONAL

Description

In Solidity smart contract development, replacing hard-coded revert message strings with the `Error()` syntax is an optimization strategy that can significantly reduce gas costs. Hard-coded strings, stored on the blockchain, increase the size and cost of deploying and executing contracts.
The `Error()` syntax allows for the definition of reusable, parameterized custom errors, leading to a more efficient use of storage and reduced gas consumption. This approach not only optimizes gas usage during deployment and interaction with the contract but also enhances code maintainability and readability by providing clearer, context-specific error information.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation

It is recommended to replace hard-coded revert strings in `require` statements for custom errors, which can be done following the logic below.
1. Standard require statement (to be replaced):

```
require(condition, "Condition not met");
```

2. Declare the error definition to state

```
error ConditionNotMet();
```

3. As currently is not possible to use custom errors in combination with `require` statements, the standard syntax is:

```
if (!condition) revert ConditionNotMet();
```

More information about this topic in Official Solidity Documentation.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.13 (HAL-06) WRONG ORACLE FEED ADDRESSES

// INFORMATIONAL

Description

The `SPCTPriceOracle` and `USDzPriceOracle` contracts contained the hardcoded address `0xAed0c38402a5d19df6E4c03F4E2DceD6e29c1ee9` as the `SPCT/USD` and `USDz/USD` oracle addresses respectively. However, this was found to be the `DAI/USD` Chainlink oracle on Ethereum mainnet.
This finding was only rated as informational risk because it is believed that the codebase is still not in production and will be updated before deploying. However, be aware that going live with the current configuration could lead to disastrous results.

BVSS

AO:S/AC:L/AX:L/C:N/I:L/A:N/D:N/Y:N/R:N/S:U (0.5)

Recommendation

Make sure the oracle addresses in `SPCTPriceOracle` and `USDzPriceOracle` contracts are correct to avoid potential catastrophic outcomes.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.14 (HAL-07) LACK OF EVENT EMISSION

// INFORMATIONAL

Description

It has been observed that important functionalities are missing emitting events.
Events are a method of informing the transaction initiator about the actions taken by the called function. It logs its emitted parameters in a specific log history, which can be accessed outside of the contract using some filter parameters. Events help non-contract tools to track changes, and events prevent users from being surprised by changes.

Score

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

Recommendation

All functions updating important parameters should emit events.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.15 (HAL-08) EVENTS MISSING INDEXED FIELDS

// INFORMATIONAL

Description

Indexed event fields make the data more quickly accessible to off-chain tools that parse events, and add them to a special data structure known as "topics" instead of the data part of the log. A topic can only hold a single word (32 bytes) so if you use a reference type for an indexed argument, the Keccak-256 hash of the value is stored as a topic instead.
Each event can use up to three indexed fields. If there are fewer than three fields, all the fields can be indexed. It is important to note that each index field costs extra gas during emission, so it's not necessarily best to index the maximum allowed fields per event (three indexed fields).
This is specially recommended when gas usage is not particularly of concern for the emission of the events in question, and the benefits of querying those fields in an easier and straight-forward manner surpasses the downsides of gas usage increase.
Notice the following events not using any indexed field:

- `SPCTPool.sol`:

```
event Execute(uint256 amount, uint256 timestamp);
event Repay(uint256 amount, uint256 timestamp);

event mintFeeRateChanged(uint256 newFeeRate, uint256 timestamp);
event redeemFeeRateChanged(uint256 newFeeRate, uint256 timestamp);
event treasuryChanged(address newTreasury, uint256 timestamp);
```

- `sUSDz.sol`:

```
event YieldReceived(uint256 amount, uint256 timestamp);
event CDPeriodChanged(uint256 newCDPeriod, uint256 timestamp);
event VestingPeriodChanged(uint256 newCDPeriod, uint256 timestamp);
```

- `USDz.sol`:

```
event mintFeeRateChanged(uint256 newFeeRate, uint256 timestamp);
event redeemFeeRateChanged(uint256 newFeeRate, uint256 timestamp);
event treasuryChanged(address newTreasury, uint256 timestamp);
event oracleChanged(address newOracle, uint256 timestamp);
```

- `vault.sol`:

```
event IncreaseCap(uint256 amount, uint256 timestamp);
event Execute(uint256 amount, uint256 timestamp);
```

Finally, it was noted that, as shown above, most events used `timestamp` as an argument. Notice that `block.timestamp` and `block.number` are added to event information by default, so adding them manually wastes gas.

Score

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

Recommendation

It is recommended to add the `indexed` keyword when declaring events, considering the following example:

```
event treasuryChanged(address indexed newTreasury, uint256 timestamp);
```

Finally, consider removing the `timestamp` argument from all the events. So, an even better approach would be:

```
event TreasuryChanged(address indexed oldTreasury, address indexed
newTreasury);
```

Remediation Plan

PARTIALLY SOLVED: The **Anzen team** solved the issue by adding indexed attributes to most of the events and removing the timestamp redundant value. Notice that this finding is not marked as solved because some events are still missing the mentioned attribute and the `src/interfaces/IsUSDz.sol` file is still referencing events with timestamps.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/c4b57e285f7fabcf5c70628b9b56b828b30d0543

7.16 (HAL-10) INTERFACES DO NOT MATCH THE SOURCE CODE

// INFORMATIONAL

Description

Within the Anzen protocol v2, a suite of interfaces is provided in the `/src/interfaces/` directory to define contract specifications. However, it was discovered that none of the contract implementations are inheriting from those interfaces. This oversight poses a risk as it lacks a crucial mechanism for ensuring consistency between interface declarations and their implementations. Inheritance of interfaces guarantees the alignment of function signatures, thereby preventing inadvertent errors resulting from incorrect implementations.
During testing, efforts were made to establish inheritance between the contract implementations and their corresponding interfaces. Unfortunately, these attempts were unsuccessful, leading to errors such as `Missing implementation` and `Function overload clash`.

Score

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

Recommendation

It is recommended to fix this issue by enforcing inheritance between contract implementations and their respective interfaces. This step is crucial for maintaining code integrity and preventing discrepancies between interface specifications and their implementations. By adhering to this practice, the risk of introducing errors due to mismatched function signatures can be mitigated effectively.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.17 (HAL-11) UNUSED IMPORTS

// INFORMATIONAL

Description

In the `USDz.sol`, `childUSDz.sol` and `Canto_childUSDz.sol` files, some of the imported contracts were found to be unused. More specifically:
On `USDz.sol`:

```
import "forge-std/Test.sol";
```

On `childUSDz.sol` and `Canto_childUSDz.sol`:

```
import {
    SendParam,
    OFTReceipt,
    MessagingReceipt,
    MessagingFee
} from "@layerzerolabs/lz-evm-oapp-v2/contracts/oft/interfaces/IOFT.sol";
```

Score

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

Recommendation

Unused imports should be cleaned up from the code if they have no purpose. Clearing these imports can reduce gas usage during the deployment of contracts, and also increases the readability of the code.

Remediation Plan

SOLVED: The **Anzen team** solved the issue by removing the unused imports.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/5911483d1bee3a8a69724da57cfc897d697a37de

7.18 (HAL-12) DEPRECATED LIBRARIES IN USE

// INFORMATIONAL

Description

During the security review, it was noted that the code of the file `src/utils/SafeMath.sol` was extracted from an older version of the popular OpenZeppelin library.

Score

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

Recommendation

As a general best-security practice, it is highly advisable to consistently review all third-party code being utilized and meticulously assess the accompanying security audits.
For this particular case, we suggest removing the use of `SafeMath` altogether, as it does not need anymore since Solidity version `0.8.0`. Notice, the version of OpenZeppelin in use by the project (v5) does not have such library anymore.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.19 (HAL-13) FLOATING PRAGMA IN SOLIDITY CONTRACTS

// INFORMATIONAL

Description

The files in scope currently use floating pragma version `^0.8.21`, which means that the code can be compiled by any compiler version that is greater than or equal to 0.8.21, and less than 0.9.0.
It is recommended that contracts should be deployed with the same compiler version and flags used during development and testing. Locking the pragma helps to ensure that contracts do not accidentally get deployed using another pragma. For example, an outdated pragma version might introduce bugs that affect the contract system negatively.
Moreover, in the `foundry.toml` configuration file, the project is declaring `version = "0.8.24"` instead of `solc_version = "0.8.24"`.

Score

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

Recommendation

Lock the pragma version to the same version used during development and testing.
For the `foundry.toml` configuration file, consider changing the declaration `version = "0.8.24"` to `solc_version = "0.8.24"`.

Remediation Plan

PARTIALLY SOLVED: The **Anzen team** fixed the (out-of-scope) `foundry.toml` configuration file. However, the source code files in scope are still using a floating pragma (`pragma solidity ^0.8.21`).

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/012a9b1e8296adbfe0149da620b9a9f411dd703c

7.20 (HAL-14) GLOBAL STATE VARIABLE CAN BE A CONSTANT

// INFORMATIONAL

Description

State variables that are not updated following deployment should be declared constant to save gas.

```
uint256 public collateralRate = 1;
```

Score

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

Recommendation

Consider adding the `constant` attribute to state variables that never change.

Remediation Plan

SOLVED: The **Anzen team** solved the issue by adding the constant attribute to the `collateralRate` state variable.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/1f4c9be633a6a2733431d999132b548117a10011

7.21 (HAL-15) CONSTANTS VISIBILITY NOT SET

// INFORMATIONAL

Description

It is best practice to set the visibility of state variables and constants explicitly. In the `SPCTPriceOracle.sol` and `USDzPriceOracle.sol` contracts, the `heartbeat` and `SPCT_USD_ORACLE_ADDRESS` constants did not have the visibility set:

```
uint256 constant heartbeat = 2 hours;
address constant SPCT_USD_ORACLE_ADDRESS =
0xAed0c38402a5d19df6E4c03F4E2DceD6e29c1ee9;
```

Score

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

Recommendation

Consider explicitly setting the visibility of all state variables and constants.

Remediation Plan

SOLVED: The **Anzen team** solved the issue by explicitly setting the visibility to the mentioned constants.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/1940320d4da410833fbcbbf01f410ecaf0fb2fb1

7.22 (HAL-16) MISLEADING OR OUTDATED COMMENTS

// INFORMATIONAL

Description

Several comments in the `sUSDz.sol` file appear to be wrong or outdated:

1. `* @notice Add Yield(USDe) to this contract.`
2. `* @notice Total vested USDe in this contract.`
3. `* @notice Used to claim USDe after CD has finished.`

These comments reference to a token named `USDe`, which is not utilized or mentioned anywhere else in the project's codebase. It seems that these comments were extracted from an older version of the codebase and do not accurately reflect the current implementation.

Score

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

Recommendation

To address these issues:

1. **Review and Update Comments**: Conduct a thorough review of all comments in the `sUSDz.sol` file. Identify any comments that reference unused or outdated features, such as `USDe`.
2. **Remove or Update Outdated Comments**: For comments referencing features that are no longer used in the project, consider removing them entirely to avoid confusion. Alternatively, if the comments are still relevant but inaccurately describe the current implementation, update them to reflect the correct information.
3. **Maintain Documentation Consistency**: Ensure that all comments accurately describe the functionality of the codebase and align with the current implementation. Regularly review and update comments as necessary to maintain consistency and clarity in the codebase documentation.

By implementing these remediation steps, you can improve the accuracy and clarity of the codebase comments, making it easier for developers to understand and maintain the project.

Remediation Plan

SOLVED: The **Anzen team** solved the issue by removing the outdated comments.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/5911483d1bee3a8a69724da57cfc897d697a37de

7.23 (HAL-17) MULTIPLE TYPOS IN COMMENTS AND ERROR MESSAGES

// INFORMATIONAL

Description

A few typographical errors were found in error messages and comments:

- On `sUSDz.sol`:

```
require(_assets <= maxWithdraw(msg.sender), "WITHDRAW_AMOUNT_EXECEED");
```

`EXECEED` should be `EXCEEDED`.

```
require(_shares <= maxRedeem(msg.sender), "WITHDRAW_AMOUNT_EXECEED");
```

`EXECEED` should be `EXCEEDED`

- On `vault.sol`:

```
// If is USDC, check total excuted amount first.
```

`excuted` should be `executed`

- On `SPCTPool.sol`:

```
* @dev SPCT only avaliable for KYC Users.
```

`avaliable` should be `available`

Score

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

Recommendation

To maintain clarity and trustworthiness, it is essential to rectify any typographical errors present within the contracts. Correcting such errors minimizes the likelihood of confusion and reinforces confidence in the accuracy and integrity of the documentation.

Remediation Plan

SOLVED: The **Anzen team** solved the issue by correcting the typographical errors found.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/d47e109c053b0980cd9e787a7e9436093b0914d4

7.24 (HAL-18) NAMING CONVENTION NOT FOLLOWED

// INFORMATIONAL

Description

During the code review, several instances of non-compliant naming conventions were identified in the Solidity codebase. Specifically:

- Some contract names were not in CapWords:
  - `childUSDz` and `sUSDz`
- Some constants were not in uppercase:
  - `uint256 constant heartbeat = 2 hours;`
  - `uint256 public constant maxMintFeeRate = FEE_COEFFICIENT / 100;`
  - `uint256 public constant maxRedeemFeeRate = FEE_COEFFICIENT / 100;`
- Some events are not in CapWords:
  - `event mintFeeRateChanged(uint256 newFeeRate, uint256 timestamp);`
  - `event redeemFeeRateChanged(uint256 newFeeRate, uint256 timestamp);`
  - `event treasuryChanged(address newTreasury, uint256 timestamp);`
  - `event mintFeeRateChanged(uint256 newFeeRate, uint256 timestamp);`
  - `event redeemFeeRateChanged(uint256 newFeeRate, uint256 timestamp);`
  - `event treasuryChanged(address newTreasury, uint256 timestamp);`
  - `event oracleChanged(address newOracle, uint256 timestamp);`
- The `getUnvestedAmount()` function of the `sUSDz` contract has a name implyting that it only "gets" a value (of `view` type), but it is actually updating some variables.

Score

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

Recommendation

To ensure consistency and improve code readability, it is recommended to adhere strictly to the naming conventions specified by the Solidity language. Contract names should be in CapWords, constants should be entirely capitalized, and event names should follow the CapWords convention. Adopting these standard naming practices will enhance the clarity and maintainability of the codebase, making it easier for developers to understand and maintain the code over time.
Additionally, enforcing these conventions through code reviews and automated linting tools can help maintain consistency across the project and prevent future deviations from the standard naming conventions.

Remediation Plan

PARTIALLY SOLVED: The **Anzen team** solved the issue by following naming convention more accurately. It is important to note that there are still some room for improvement like the `collateralRate`, `maxMintFeeRate`, `maxRedeemFeeRate` and `heartbeat` constants (should be in `UPPER_CASE`), the `vault.sol` and `childUSDz.sol` file names (could be renamed to `Vault.sol` and `ChildUSDz.sol` respectively) and `childUSDz` contract inside the file `Canto_childUSDz.sol` (could be renamed to `CantoChildUSDz` or `ChildUSDz`).

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/6e543ae8fdaa44a4816ebe1af8c072a559e7cf8c

7.25 (HAL-19) TAUTOLOGY AND BOOLEAN EQUALITIES

// INFORMATIONAL

Description

In the contract `SPCTPool`, a tautology expression was detected because it is checking an unsigned integer (`uint256`) is greater than 0. Such expressions are of no use since they always evaluate to true regardless of the parameters they receive.

```
function repay(uint256 _amount) external onlyRole(POOL_MANAGER_ROLE) {
  require(_amount > 0, "REPAY_AMOUNT_IS_ZERO");
  require(executedUSD.sub(_amount) >= 0,
"REPAY_AMOUNT_EXCEED_EXECUTED_SHARES");
  ...
```

Furthermore, two boolean equalities were found in the `USDz` contract. This means comparing to a constant (`true` or `false`), which is a bit more gas expensive than directly checking the boolean value.

```
require(mode == false, "PLEASE_MIGRATE_TO_NEW_VERSION");
```

Score

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

Recommendation

For the detected tautology, if the intention is to make sure `amount` is not greater than `executedUSD`, consider the following change:

```
if(amount > executedUSD) {
  revert("REPAYAMOUNT_EXCEED_EXECUTED_SHARES");
}
```

For the boolean equalities, consider using `if(!mode)` instead of `require(mode == false)`.

Remediation Plan

PARTIALLY SOLVED: The **Anzen team** partially solved the issue by fixing most of the require statements. However, for the boolean equality, only the one present in the `deposit()` was fixed, but there is one remaining in the `depositBySPCT()` function. Finally, the require to check if `repay()` was called with an `_amount` of 0 was removed, but it is recommended to add it back to prevent the function from emitting events with 0 value.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/9b375186c98265bdce0eebdd3a50118c9317bb4c

7.26 (HAL-20) INTERNAL FUNCTIONS CALLED ONLY ONCE CAN BE INLINED

// INFORMATIONAL

Description

During the security assessment, it was observed that both the internal function `_checkCollateralRate()` within the `USDz` contract and the `_updateVestingAmount()` internal function within the `sUSDz` contract were only invoked once.
While modular design often enhances code readability and maintainability, in instances where internal functions are utilized singularly, it can introduce unnecessary complexity and hinder code comprehension. This redundancy not only obscures the logic flow but also increases the cognitive load for developers maintaining or auditing the codebase.

Score

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

Recommendation

Instead of separating the logic into a separate function, consider inlining the logic into the calling function or modifier. This can reduce the number of function calls and improve readability.

Remediation Plan

ACKNOWLEDGED: The **Anzen team** acknowledged the issue.

7.27 (HAL-22) UNUSED EVENTS

// INFORMATIONAL

Description

The following events are defined but never emitted.

```
event Mint(address indexed user, uint256 amount, uint256 timestamp);
event Burn(address indexed user, uint256 amount, uint256 timestamp);
event Repay(uint256 amount, uint256 timestamp);
```

Score

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

Recommendation

Unused events can be removed to make the code cleaner.

Remediation Plan

SOLVED: The **Anzen team** solved the issue by either start using or removing the unused events.

Remediation Hash

https://github.com/Anzen-Finance/protocol-v2/commit/a3e40a70e6ea0a7180442b5360de8398d85c39a3

STATIC ANALYSIS REPORT

Description

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

Output

```
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
USDz.deposit(uint256) (src/USDz.sol#143-200) ignores return value by usdc.transferFrom(msg.sender,address(this),_amount) (src/USDz.sol#148)
USDz.depositBySPCT(uint256) (src/USDz.sol#208-230) ignores return value by spct.transferFrom(msg.sender,address(this),_amount) (src/USDz.sol#213)
USDz.redeem(uint256) (src/USDz.sol#239-300) ignores return value by usdc.transfer(msg.sender,convertToUSDC) (src/USDz.sol#256)
USDz.redeem(uint256) (src/USDz.sol#239-300) ignores return value by usdc.transfer(msg.sender,convertToUSDC) (src/USDz.sol#265)
USDz.redeem(uint256) (src/USDz.sol#239-300) ignores return value by usdc.transfer(msg.sender,convertToUSDC) (src/USDz.sol#280)
USDz.redeem(uint256) (src/USDz.sol#239-300) ignores return value by usdc.transfer(msg.sender,convertToUSDC) (src/USDz.sol#295)
USDz.redeemBackSPCT(uint256) (src/USDz.sol#308-328) ignores return value by spct.transfer(msg.sender,_amount) (src/USDz.sol#325)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
OFTCore._removeDust(uint256) (lib/LayerZero-v2/oapp/contracts/oft/OFTCore.sol#302-304) performs a multiplication on the result of a division:
        - (_amountLD / decimalConversionRate) * decimalConversionRate (lib/LayerZero-v2/oapp/contracts/oft/OFTCore.sol#303)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
OFTCore._buildMsgAndOptions(SendParam,uint256) (lib/LayerZero-v2/oapp/contracts/oft/OFTCore.sol#191-212) ignores return value by IOAppMsgInspector(msgInspector).inspect(message,options) (lib/LayerZero-v2/oapp/contracts/oft/OFTCore.sol#211)
USDz.deposit(uint256) (src/USDz.sol#143-200) ignores return value by usdc.approve(address(spct),_amount) (src/USDz.sol#149)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

```
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
USDzFlat.withdraw(address,uint256) (src/USDzFlat.sol#26-29) ignores return value by usdz.transfer(_to,_amount) (src/USDzFlat.sol#27)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
sUSDz._updateVestingAmount(uint256) (src/sUSDz.sol#279-284) uses a dangerous strict equality:
        - require(bool,string)(getUnvestedAmount() == 0,UNVESTING_IS_NOT_ZERO) (src/sUSDz.sol#280)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dangerous-strict-equalities


INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation

INFO:Detectors:
SPCTPool.deposit(uint256) (src/SPCTPool.sol#85-116) ignores return value by usdc.transferFrom(msg.sender,address(this),_amount) (src/SPCTPool.sol#89)
SPCTPool.redeem(uint256) (src/SPCTPool.sol#137-164) ignores return value by usdc.transfer(msg.sender,convertToUSDC) (src/SPCTPool.sol#159)
SPCTPool.execute(uint256) (src/SPCTPool.sol#185-193) ignores return value by usdc.transfer(msg.sender,_amount) (src/SPCTPool.sol#190)
SPCTPool.repay(uint256) (src/SPCTPool.sol#201-209) ignores return value by usdc.transferFrom(msg.sender,address(this),_amount) (src/SPCTPool.sol#208)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply

INFO:Detectors:
SPCTPool.repay(uint256) (src/SPCTPool.sol#201-209) contains a tautology or contradiction:
        - require(bool,string)(executedUSD.sub(_amount) >= 0,REPAY_AMOUNT_EXCEED_EXECUTED_SHARES) (src/SPCTPool.sol#203)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#tautology-or-contradiction
```

```
INFO:Detectors:
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) has bitwise-xor operator ^ instead of the exponentiation operator **:
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-exponentiation
INFO:Detectors:
OFTCore._removeDust(uint256) (lib/LayerZero-v2/oapp/contracts/oft/OFTCore.sol#302-304) performs a multiplication on the result of a division:
        - (_amountLD / decimalConversionRate) * decimalConversionRate (lib/LayerZero-v2/oapp/contracts/oft/OFTCore.sol#303)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse = (3 * denominator) ^ 2 (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#184)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#188)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#189)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#190)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#191)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#192)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - denominator = denominator / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#169)
        - inverse *= 2 - denominator * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#193)
Math.mulDiv(uint256,uint256,uint256) (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#123-202) performs a multiplication on the result of a division:
        - prod0 = prod0 / twos (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#172)
        - result = prod0 * inverse (lib/openzeppelin-contracts/contracts/utils/math/Math.sol#199)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#divide-before-multiply
INFO:Detectors:
OFTCore._buildMsgAndOptions(SendParam,uint256) (lib/LayerZero-v2/oapp/contracts/oft/OFTCore.sol#191-212) ignores return value by IOAppMsgInspector(msgInspector).inspect(message,options) (lib/LayerZero-v2/oapp/contracts/oft/OFTCore.sol#211)
childUSDz.constructor(address,address,uint256) (src/Canto_childUSDz.sol#28-36) ignores return value by turnstile.assign(csrId) (src/Canto_childUSDz.sol#35)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return
```

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.