SMART CONTRACTS REVIEW

zokyo

August 8th 2025 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contract passed a security audit.

ZOKYO AUDIT SCORING UWI

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: 0 points

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 1 High issue: 1 resolved = 0 points deducted
- 3 Medium issues: 3 resolved = 0 points deducted
- 3 Low issues: 3 resolved = 0 points deducted
- 3 Informational issues: 3 resolved = 0 points deducted

Thus, the score is 100

TECHNICAL SUMMARY

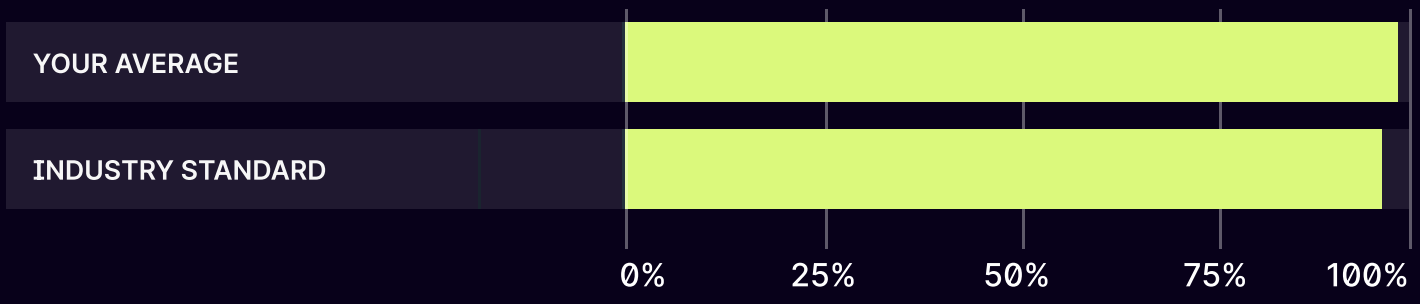
This document outlines the overall security of the UWI smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the UWI smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

Testable Code



98% of the code is testable, which is above the industry standard of 95%.

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the UWI team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the UWI repository:
Repo: https://github.com/uwihomes/musdvault

Last commit -411cf50d5c289bf3d6da70d0bdda18698dbf87d7

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- MUSDVault.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of UWI smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. Part of this work includes writing a test suite using the Foundry testing framework. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.	03	Testing contract/s logic against common and uncommon attack vectors.
02	Cross-comparison with other, similar smart contract/s by industry leaders.	04	Thorough manual review of the codebase line by line.

Executive Summary

The UWI codebase for audit consist of the contract MUSDVault.sol that allows users to deposit MUSD to the vault in return for vault shares. The vault has non-transferrable shares, whitelisting functionality (batch-controlled for gas limit issue) and push-based yield distribution, using access-based control. Admin can deploy the vault's asset for yield with full-tracking and loss is calculated upon funds return. The vault implements safety mechanisms using pause/unpause operation for critical functionalities.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the UWI team and the UWI team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Users might not be added to the holders list and receive no yields	High	Resolved
2	Vault fails for any asset with decimals other than 18	Medium	Resolved
3	Vault inflation griefing attack	Medium	Resolved
4	Centralization in the vault	Medium	Resolved
5	Incorrect event data for injectRecoveryCapital() method	Low	Resolved
6	Incorrect event data for yieldDistribution() method	Low	Resolved
7	Method injectRecoveryCapital can be front-run	Low	Resolved
8	Remove legacy variable no longer needed	Informational	Resolved
9	Missing clearer error messages	Informational	Resolved
10	Missing zero value check for assets in depositFor() function	Informational	Resolved

Users might not be added to the holders list and receive no yields

In Contract MUSDVault.sol, the method cleanupInactiveHolders() removes any users from the holders list whose share balance is zero. However, it does not set the _hasHeld mapping for that user to false, which results in the user not being added to the holders list when they deposit again in the future.

1. User deposits ⇒ added to holders list and _hasHeld set true, marked active
2. User redeems all shares ⇒ marked inactive
3. Admin runs cleanupInactiveHolders ⇒ user is removed from the holders list
4. User deposits again ⇒ since _hasHeld is still true, user won't be added to the holders list but will be marked active
5. Admin distributes yield ⇒ since user not in holders list ⇒ received no yield

Test:

```
it.only('Should add user to holders list when deposit again after clean up', async function () {
    const depositAmount = ethers.parseEther('1000');
// user1 deposited tokens
    await vault
      .connect(user1)
      .deposit(depositAmount, user1.address);
    const withdrawAmount = ethers.parseEther('1000');
    const musdBefore = await musd.balanceOf(user1.address);
// user1 redeems all shares
    await expect(
      vault
        .connect(user1)
        .withdraw(withdrawAmount, user1.address, user1.address)
    ).to.emit(vault, 'Withdraw');
    const musdAfter = await musd.balanceOf(user1.address);
    expect(musdAfter - musdBefore).to.equal(withdrawAmount);
    console.log(
      'user share balance after complete withdrawal',
      await vault.balanceOf(user1)
    );
```

```
  let holdersList = await vault.holdersList();
      console.log('holders list before admin cleans inactive holders', holdersList);
// admin cleans inactive holders
      await vault.cleanupInactiveHolders(10);
      holdersList = await vault.holdersList();
      console.log('holders list after admin cleans inactive holder', holdersList);
// user1 deposits again,
await vault
        .connect(user1)
        .deposit(depositAmount, user1.address);
      holdersList = await vault.holdersList();
      console.log('holders list after user deposits again', holdersList);
    });
```

Impact:
If a user is not added to the holders list while having the shares, the user will not receive and yield in the yieldDistribution method because this method iterates through all the holders in the holders list and transfers yields to their address. Since this user is not in the holder list, it won't receive any yield. This is a fund loss for users as they have deposited, hold shares and still receive no yield.

Recommendation:
Update the method cleanupInactiveHolders() to set _hasHeld as false for the users who are removed from the holders list.

Vault fails for any asset with decimals other than 18

The readme file mentions the vault can work successfully with decimals 6 (like USDC), decimals 18 (most stable coins), etc. While it behaves correctly with MUSD, which has 18 decimals, the vault will fail to function properly if an asset with any other decimal is used.

The reason for the vault failing lies in these methods:

1. SharePrice calculation: the method sharePrice() returns share price as 10**decimals if supply == 0. If the asset has decimals 6(for eg, USDZ), it will return the share price as 1e18 when supply = = meaning one share is worth 10^12 USDC. This will impact the shares calculation for users and the vault.
2. isUnderwater calculation: the method isUnderwater() checks if the share price is less than 1 asset token or not. But it checks it by comparing sharePrice() (in asset decimals) with 10**decimals (i.e. 1e18). If the asset is USDC (6 decimals), the sharePrice will always be less than 1e18, resulting in the vault always being underwater.

Impact:
Since the vault will return the wrong share price when supply == 0 and will always be underwater for an asset with decimals 6, and the same would follow for any asset with decimals less than 18, share management and yield distribution will fail totally.

Recommendation:
Update the isUnderwater() method to check if sharePrice is less than 10**asset_decimals(). Similarly, the sharePrice() method should return 10**asset_decimals when supply == 0.

Vault inflation griefing attack

In contract MUSDVault.sol, although Openzeppelin's ERC4626 version of the contract has been inherited, it is still vulnerable to Vault inflation attack. This is because the _decimalOffset() of the ERC4626 contract has default value of zero which does not completely prevent the inflation attack from happening by the first depositor to the vault.

```
  function decimals() public view virtual override(IERC20Metadata, ERC20) returns
(uint8) {
      return _underlyingDecimals + _decimalsOffset();
  }
```

```
  function _decimalsOffset() internal view virtual returns (uint8) {
      return 0;
  }
```

For example, here is the attack flow:
- Alice deposits 1 wei → Gets 1 share
- Alice directly transfers 100,000 MUSD to vault (asset inflation)

After Alice's inflation, Bob deposits 50,000 MUSD:
shares = assets * totalSupply / totalAssets+1
shares = 50,000 MUSD * 1 / 100,000,000,000,000,000,000,001
shares = 0.5 (rounds down to 0!)

Thus Bob gets only 0 shares for 50,000 MUSD!

Now when Alice withdraws her share, she is able to withdraw not all but 75000 MUSD. This leads to Alice losing 25% of her assets overall, but Bob loses 100% of his assets.

But if the decimal offset is set to a higher value such as 6, Bob's loss for the same attack reduces to about 1% only.

Proof of Concept (PoC): https://gist.github.com/shanzson/27c68eb0f5b0a3bd3eae6832f5750bd7

Impact:
The inflation attack by the first depositor can lead to the subsequent user still losing their assets significantly, but the first depositor will lose some assets too. This is also known as a griefing attack in which the attacker loses some funds in order to carry out an attack. Additionally as the vault has a whitelist functionality, this reduces the risk of such attacks from happening immediately.

Recommendation:
It is advised to vault admin to deposit first into the vault via a non-trivial amount. For example, for the amount for vaults with assets like USDC (6 decimals), depositing 1 full unit (1 USDC), 10 USDC, or an amount that is not trivially small is common and safe practice. This would significantly reduce the risk of inflation attack and then the subsequent users could be whitelisted for depositing into the vault.

MEDIUM-3 | RESOLVED

Centralization in the vault

In contract MUSDVault.sol, the function deployCapital() can be used to deploy the entire vault funds to any external address by the admin. Moreover, the function returnCapital() which is primarily used to update the state regarding the total capital deployed from the vault and track the total losses after returning funds to the vault, does not check if the funds were actually added to the vault.

Impact:
This could lead to a malicious admin to deploy capital to any external wallet address and not return the deployed funds. Also it is possible to update and reduce the totalosses via the returnCapital() function without actually returning funds to the vault.

Recommendation:
Although the comments in the code say that “The admin must transfer funds to the vault before calling this(here returnCapital()) function”, it would be a best practice to carry out returning of funds via the same function to ensure that totalLosses and total capital deployed can only be updated when the funds have been returned to the vault. Additionally it is advised that the team uses a multisig for the admin and whitelist admin roles with a configuration of 5/9 or 6/11 to safeguard against accidental private key losses and any compromise of admin roles.

Incorrect event data for injectRecoveryCapital() method

In contract MUSDVault.sol, the method injectRecoveryCapital() allows the admin to donate assets to the vault to recover the loss and emits an event `RecoveryCapitalInjected` as follows:

```
// Calculate share price before external call
        uint256 priceAfterInjection = sharePrice(); //@audit no injection
yet

        // Emit event before external call (best practice)
        emit RecoveryCapitalInjected(amount, priceAfterInjection);

        // External call last
        token.safeTransferFrom(msg.sender, address(this), amount);
```

Here, the event is emitted before the asset is transferred from the admin to the vault.

Impact:
The event is using sharePrice() as one of its data; the sharePrice() here will be the stale value, as sharePrice() is calculated using totalAssets(). Since tokens are transferred before the sharePrice() is calculated and the event being emitted, the token transferred will not be used for sharePrice() calculation, resulting in a stale sharePrice(), which will impact any source dependent on this event's value.

Recommendation:
Update the logic as follows:

```
// External call
        token.safeTransferFrom(msg.sender, address(this), amount);

        // Calculate share price before external call
        uint256 priceAfterInjection = sharePrice();

        // Emit event before external call (best practice)
        emit RecoveryCapitalInjected(amount, priceAfterInjection);
```

Incorrect event data for yieldDistribution() method

In the contract MUSDVault.sol, the method _performDistribution() emits the event yieldEpochComplete with totalYield and actuallyDistributed, which have the same values for an epoch. That might be incorrect in case there are multiple distribution transactions, as the yield amount will be different for each txn.
For eg, 1st distribution with 300 USD for 200 users and 2nd with 300 USD for 200 users (last 200 users, let's say), so total yield for epoch should be 600 USD and distributed will be 600 USD as well. But as per the current logic, the event will be emitted with data totalYield as 300 and actuallyDistributed as 300.

Impact:
Any off-chain source relying on the data of these events would get incorrect data, showing less value for total yield in an epoch.

Recommendation:
Update the _performDistribution() method to correctly emit the event data or transfer all yields in one transaction.

LOW-3 | RESOLVED

Method injectRecoveryCapital can be front-run

In the contract MUSDVault.sol, the method injectRecoveryCapital() allows the admin to donate assets to the vault to recover from losses. While this raises the share price for the vault, it can be front-run as well.

1. Admin sends the txn, which calls injectRecoveryCapital() to raise the share price
2. A user checks this txn in the mempool and front-runs it to mint shares at the current low price
3. Once the admin txn is confirmed, the user redeems those shares at a better share price.

Impact:
This will impact all users who have vault shares, as part of the recovery capital might be stolen due to front-running.

Recommendation:
As it is in the method yieldDistributionWithRetention, this method can be called only when the deposit is disabled; similarly, a check can be added to the injectRecoveryCapital method to disable the deposit.

INFORMATIONAL-1 | RESOLVED

Remove legacy variable no longer needed

In the contract MUSDVault.sol, there is a variable “lossesAcknowledged” which is declared and used in the method sharePrice() but never really updated, meaning its value always remains `0`.
Since the variable is a legacy variable and is no longer needed, it can be removed.

Recommendation:
Remove the legacy variable “lossesAcknowledged”, which is not used properly.

Missing clearer error messages

The function withdrawFor() function does not check for maxWithdraw value of an owner. Although without this check, the transaction would fail later in the execution when trying to burn more shares than the owner has, but with a less clear error message. By checking early with maxWithdraw, it would be still good to add a more robust and clearer way to handle this failing of transaction with a require statement and error instead.

The same is the case with depositFor(), mintFor() and redeemFor() function.

Recommendation:
It is advised to add the corresponding require statements for each function respectively as a best practice

```
require(assets <= maxWithdraw(owner), "MUSDVault: withdraw more than max");
```

```
require(assets <= maxDeposit(receiver), "MUSDVault: deposit more than max");
require(shares <= maxMint(receiver), "MUSDVault: mint more than max");
```

```
require(shares <= maxRedeem(owner), "MUSDVault: redeem more than max");
```

Missing zero value check for assets in depositFor() function

The depositFor() function allows assets with `0` amount to be passed into the function and the function passes. While this does not pose any security issue as of now, in future upgrades to the contract, the team must be careful not to allow this case to do any unwanted changes in state to the other variables in the contract.

```
function depositFor(uint256 assets, address receiver, uint256 minShares)
    public
    nonReentrant
    whenNotPaused
    returns (uint256 shares)
{
    if (!depositsEnabled) revert DepositsDisabled();
    if (whitelistEnabled && !whitelist[receiver]) revert NotWhitelisted();
    if (receiver == address(0)) revert ZeroAddress();
    ...
    }
```

Recommendation:

It is advised to add a revert statement or a require statement to prevent zero assets being passed for the execution of the function.

```
    if (assets == 0) revert ZeroAssets();
```

	MUSDVault.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

CODE COVERAGE AND TEST RESULTS FOR ALL FILES

Tests written by Zokyo Security

As a part of our work assisting UWI in verifying the correctness of their contract code, our team was responsible for writing integration tests using the Foundry testing framework.

The tests were based on the functionality of the code, as well as a review of the UWI contract requirements for details about issuance amounts and how the system handles these.

Ran 120 tests for test/foundry/Zokyo_tests.t.sol:Zokyo_tests
[PASS] test_AcknowledgeLossAlwaysReverts() (gas: 16314)
[PASS] test_AddToDeploymentWhitelist() (gas: 45691)
[PASS] test_AddToDeploymentWhitelistSelf() (gas: 18587)
[PASS] test_AddToDeploymentWhitelistZeroAddress() (gas: 18589)
[PASS] test_AddToWhitelist() (gas: 45613)
[PASS] test_AdminDepositSuccess() (gas: 241781)
[PASS] test_AdminDepositUnauthorized() (gas: 50165)
[PASS] test_AdminDepositZeroAddress() (gas: 25824)
[PASS] test_ApproveAlwaysReverts() (gas: 18413)
[PASS] test_AuthorizeUpgrade() (gas: 4426212)
[PASS] test_AuthorizeUpgradeUnauthorized() (gas: 4448589)
[PASS] test_BatchDistributionAcrossMultipleEpochs() (gas: 1114377)
[PASS] test_BatchUpdateDeploymentWhitelist() (gas: 76601)
[PASS] test_BatchUpdateDeploymentWhitelistSelf() (gas: 45280)
[PASS] test_BatchUpdateDeploymentWhitelistTooLarge() (gas: 39030)
[PASS] test_BatchUpdateDeploymentWhitelistZeroAddress() (gas: 44863)
[PASS] test_BatchUpdateWhitelist() (gas: 101651)
[PASS] test_BatchUpdateWhitelistTooLarge() (gas: 58731)
[PASS] test_CannotInitializeTwice() (gas: 19276)
[PASS] test_CleanupHoldersArraySwapLogic() (gas: 450773)
[PASS] test_CleanupHoldersArraySwapLogicSpecific() (gas: 468547)
[PASS] test_CleanupInactiveHolders() (gas: 448983)
[PASS] test_CleanupInactiveHoldersComplexScenario() (gas: 875232)
[PASS] test_CleanupInactiveHoldersInvalidBatchSize() (gas: 27460)
[PASS] test_ComplexScenarioWithMultipleHolders() (gas: 802362)
[PASS] test_ConstructorDisablesInitializers() (gas: 14299)
[PASS] test_DeployCapitalAmountTooLarge() (gas: 21316)
[PASS] test_DeployCapitalNoPurpose() (gas: 21095)
[PASS] test_DeployCapitalStrategyNotWhitelisted() (gas: 241697)

[PASS] test_DeployCapitalSuccess() (gas: 437090)
[PASS] test_DeployCapitalToSelf() (gas: 19052)
[PASS] test_DeployCapitalWhitelistDisabled() (gas: 403187)
[PASS] test_DeployCapitalZeroAddress() (gas: 19055)
[PASS] test_DeployCapitalZeroAmount() (gas: 21124)
[PASS] test_DepositForSlippageExceeded() (gas: 44140)
[PASS] test_DepositForWithSlippageProtection() (gas: 233045)
[PASS] test_DepositSuccess() (gas: 240815)
[PASS] test_DepositToExistingHolder() (gas: 248122)
[PASS] test_DepositToZeroAddress() (gas: 25737)
[PASS] test_DepositTracksNewHolders() (gas: 234388)
[PASS] test_DepositWhenDepositsDisabled() (gas: 35948)
[PASS] test_DepositWhenPaused() (gas: 51072)
[PASS] test_DepositWhenWhitelistEnabled() (gas: 283374)
[PASS] test_DistributeYieldBatchTooLarge() (gas: 386090)
[PASS] test_DistributeYieldDepositsNotDisabled() (gas: 234775)
[PASS] test_DistributeYieldInsufficientAllowance() (gas: 250924)
[PASS] test_DistributeYieldInvalidRange() (gas: 241234)
[PASS] test_DistributeYieldNoActiveShares() (gas: 211189)
[PASS] test_DistributeYieldOutOfBounds() (gas: 241334)
[PASS] test_DistributeYieldSuccess() (gas: 434011)
[PASS] test_DistributeYieldWithRetention() (gas: 524930)
[PASS] test_DistributeYieldWithRetentionInvalidPercentage() (gas: 463941)
[PASS] test_DistributeYieldWithRetentionNotUnderwater() (gas: 250842)
[PASS] test_DistributeYieldWithZeroRetention() (gas: 517664)
[PASS] test_DistributeYieldZeroAmount() (gas: 241143)
[PASS] test_DistributionLastHolderRoundingEdgeCase() (gas: 445676)
[PASS] test_EdgeCaseZeroSharesAfterDeposit() (gas: 199505)
[PASS] test_EventEmissions() (gas: 473374)
[PASS] test_FullCoverageEdgeCases() (gas: 629005)
[PASS] test_GetActiveDeployedCapital() (gas: 511371)
[PASS] test_GetActiveHoldersCount() (gas: 198752)
[PASS] test_GetDeploymentHistory() (gas: 511464)
[PASS] test_GetHoldersCount() (gas: 368131)
[PASS] test_HolderCleanupEdgeCases() (gas: 484180)
[PASS] test_HolderReactivationEdgeCase() (gas: 302910)
[PASS] test_Initialize() (gas: 66755)
[PASS] test_InjectRecoveryCapital() (gas: 479492)
[PASS] test_InjectRecoveryCapitalInsufficientAllowance() (gas: 460167)
[PASS] test_InjectRecoveryCapitalNotUnderwater() (gas: 241750)
[PASS] test_InjectRecoveryCapitalZeroAmount() (gas: 18410)

[PASS] test_IsUnderwater() (gas: 460990)
[PASS] test_MintForSlippageExceeded() (gas: 44168)
[PASS] test_MintForWithSlippageProtection() (gas: 233115)
[PASS] test_MintForWithTypeMaxSlippageEdgeCase() (gas: 229720)
[PASS] test_MintSuccess() (gas: 236316)
[PASS] test_MintToZeroAddress() (gas: 25740)
[PASS] test_MintWhenDepositsDisabled() (gas: 35951)
[PASS] test_MintZeroShares() (gas: 25868)
[PASS] test_PauseUnpause() (gas: 34097)
[PASS] test_RecoveryScenarioWithRetainedYield() (gas: 551028)
[PASS] test_RedeemAllSharesMarksInactive() (gas: 199270)
[PASS] test_RedeemForExactZeroBalanceEdgeCase() (gas: 198161)
[PASS] test_RedeemForInactiveHolderZeroBalance() (gas: 321996)
[PASS] test_RedeemForReturnPath() (gas: 250003)
[PASS] test_RedeemForSlippageExceeded() (gas: 239439)
[PASS] test_RedeemForWithSlippageProtection() (gas: 246515)
[PASS] test_RedeemForWithZeroMinAssetsEdgeCase() (gas: 245043)
[PASS] test_RedeemSuccess() (gas: 251635)
[PASS] test_RedeemToZeroAddress() (gas: 233898)
[PASS] test_RedeemWhenRedeemsDisabled() (gas: 244172)
[PASS] test_RemoveFromDeploymentWhitelist() (gas: 35726)
[PASS] test_RemoveFromWhitelist() (gas: 35576)
[PASS] test_ReturnCapitalDeploymentNotActive() (gas: 26023)
[PASS] test_ReturnCapitalSuccess() (gas: 405758)
[PASS] test_ReturnCapitalTooMuch() (gas: 434431)
[PASS] test_ReturnCapitalWithLoss() (gas: 449446)
[PASS] test_RoleBasedAccessControl() (gas: 251373)
[PASS] test_SetDeploymentWhitelistEnabled() (gas: 27783)
[PASS] test_SetDepositsEnabled() (gas: 27866)
[PASS] test_SetMaxHoldersPerDistribution() (gas: 25847)
[PASS] test_SetMaxHoldersPerDistributionInvalidMax() (gas: 27532)
[PASS] test_SetRedeemsEnabled() (gas: 27765)
[PASS] test_SetWhitelistEnabled() (gas: 28126)
[PASS] test_SetWhitelistEnabledUnauthorized() (gas: 47969)
[PASS] test_SharePrice() (gas: 458560)
[PASS] test_SharePriceCalculationWithLargeValues() (gas: 592961)
[PASS] test_SharePriceNoSupply() (gas: 15300)
[PASS] test_TotalAssets() (gas: 435513)
[PASS] test_TransferAlwaysReverts() (gas: 232418)
[PASS] test_TransferFromAlwaysReverts() (gas: 232486)
[PASS] test_WithdrawAllSharesMarksInactive() (gas: 201941)

[PASS] test_WithdrawForExactZeroBalanceEdgeCase() (gas: 198180)
[PASS] test_WithdrawForInactiveHolderZeroBalance() (gas: 392263)
[PASS] test_WithdrawForReturnPath() (gas: 250071)
[PASS] test_WithdrawForSlippageExceeded() (gas: 239531)
[PASS] test_WithdrawForWithSlippageProtection() (gas: 246625)
[PASS] test_WithdrawForWithTypeMaxSlippageEdgeCase() (gas: 246655)
[PASS] test_WithdrawSuccess() (gas: 251805)
[PASS] test_WithdrawToZeroAddress() (gas: 233964)
[PASS] test_WithdrawWhenRedeemsDisabled() (gas: 244198)
Suite result: ok. 120 passed; 0 failed; 0 skipped; finished in 797.62ms (73.40ms CPU time)

Ran 41 tests for test/foundry/Zokyo_Fuzz_tests.t.sol:Zokyo_Fuzz_tests
[PASS] testFuzz_AccessControl(uint256) (runs: 256, μ: 91210, ~: 62581)
[PASS] testFuzz_AcknowledgeLoss(uint256) (runs: 256, μ: 22963, ~: 23076)
[PASS] testFuzz_AdminDeposit(uint256) (runs: 256, μ: 253796, ~: 253909)
[PASS] testFuzz_CleanupInactiveHolders(uint256) (runs: 256, μ: 5207431, ~: 5319145)
[PASS] testFuzz_ComplexScenario(uint256) (runs: 256, μ: 619465, ~: 600753)
[PASS] testFuzz_DeployCapital(uint256,bool) (runs: 256, μ: 452206, ~: 434434)
[PASS] testFuzz_DeploymentWhitelistOperations(uint256) (runs: 256, μ: 704246, ~: 719159
[PASS] testFuzz_Deposit(uint256,bool,bool) (runs: 256, μ: 176050, ~: 79523)
[PASS] testFuzz_DepositFor(uint256,uint256) (runs: 256, μ: 171183, ~: 244589)
[PASS] testFuzz_DistributeYield(uint256,uint256) (runs: 256, μ: 471712, ~: 481583)
[PASS] testFuzz_DistributeYieldWithRetention(uint256,uint256) (runs: 256, μ: 724022, ~: 7
[PASS] testFuzz_GetActiveDeployedCapital() (gas: 548172)
[PASS] testFuzz_GetDeploymentHistory(uint256) (runs: 256, μ: 562759, ~: 536334)
[PASS] testFuzz_InjectRecoveryCapital(uint256,bool) (runs: 256, μ: 381081, ~: 255708)
[PASS] testFuzz_Mint(uint256,bool) (runs: 256, μ: 143996, ~: 44145)
[PASS] testFuzz_MintFor(uint256,uint256) (runs: 256, μ: 161252, ~: 248824)
[PASS] testFuzz_PauseOperations(uint256,bool) (runs: 256, μ: 165411, ~: 239191)
[PASS] testFuzz_Redeem(uint256,uint256) (runs: 256, μ: 273679, ~: 273826)
[PASS] testFuzz_RedeemFor(uint256,uint256,uint256) (runs: 256, μ: 276893, ~: 282636)
[PASS] testFuzz_RemoveFromWhitelist(uint256) (runs: 256, μ: 617827, ~: 630839)
[PASS] testFuzz_ReturnCapital(uint256,uint256,bool) (runs: 256, μ: 458553, ~: 480869)
[PASS] testFuzz_SetMaxHoldersPerDistribution(uint256) (runs: 256, μ: 30728, ~: 30819)
[PASS] testFuzz_TransferRestrictions(uint256) (runs: 256, μ: 249667, ~: 249780)
[PASS] testFuzz_ViewFunctions(uint256,uint256,uint256) (runs: 256, μ: 503976, ~: 506010)
[PASS] testFuzz_WhitelistOperations(uint256) (runs: 256, μ: 918970, ~: 541221)
[PASS] testFuzz_Withdraw(uint256,uint256,bool) (runs: 256, μ: 274328, ~: 287796)
[PASS] testFuzz_WithdrawFor(uint256,uint256,uint256) (runs: 256, μ: 268371, ~: 262480)
[PASS] testFuzz_ZeroAddressRejection(uint256) (runs: 256, μ: 96551, ~: 96664)
[PASS] test_ActiveHolderConsistency() (gas: 16927)

```
[PASS] test_DeploymentWhitelistEdgeCases() (gas: 413293)
[PASS] test_EdgeCasesFor100PercentCoverage() (gas: 654951)
[PASS] test_ExtremeEdgeCasesForCoverage() (gas: 1078593)
[PASS] test_FinalCoverageAttempt() (gas: 685633)
[PASS] test_HolderTrackingEdgeCases() (gas: 312512)
[PASS] test_SharePriceEdgeCases() (gas: 467919)
[PASS] test_SharePricePositive() (gas: 16050)
[PASS] test_SlippageProtectionExactBoundaries() (gas: 385672)
[PASS] test_SmokeTest() (gas: 259521)
[PASS] test_TotalAssetsConsistency() (gas: 27319)
[PASS] test_TotalSupplyConsistency() (gas: 16972)
[PASS] test_YieldDistributionCompleteEdgeCases() (gas: 679414)
Suite result: ok. 41 passed; 0 failed; 0 skipped; finished in 2.92s (16.05s CPU time)

Ran 2 test suites in 1.62s (2.41s CPU time): 161 tests passed, 0 failed, 0
skipped (161 total tests)
```

The resulting code coverage (i.e., the ratio of tests-to-code) is as follows:

FILE	% STMTS	% BRANCH	% FUNCS	% LINES
MUSDVault.sol	98.02%	95.83%	100%	98.23%
All Files	**98.02%**	**95.83%**	**100%**	**98.23%**

We are grateful for the opportunity to work with the UWI team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the UWI team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo