SMART CONTRACTS REVIEW

zokyo

December 31st 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING VAULTCRAFT

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:
- 2 Critical issues: 1 resolved and 1 acknowledged = - 7 points deducted
- 2 High issues: 2 acknowledged = - 5 points deducted
- 7 Medium issue: 4 resolved and 3 acknowledged = - 4 points deducted
- 4 Low issues: 1 resolved and 3 acknowledged = - 2 points deducted
- 4 Informational issue: 4 acknowledged = 0 points deducted

Thus, 100 - 7 - 5 - 4 - 2 = 82

TECHNICAL SUMMARY

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

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

Contract Status

There were 2 critical issues found during the review. (See Complete Analysis)

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 VaultCraft 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 VaultCraft repository:
Repo: https://github.com/Popcorn-Limited/contracts/pull/188

Last commit - 68dfc2d8021445bf73c3964c6ceafc7e3b56f850

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

- src/vaults/multisig/phase1/AsyncVault.sol
- src/vaults/multisig/phase1/BaseControlledAsyncRedeem.sol
- src/vaults/multisig/phase1/BaseERC7540.sol
- src/vaults/multisig/phase1/OracleVault.sol
- src/peripheral/oracles/adapter/pushOracle/PushOracle.sol
- src/peripheral/oracles/OracleVaultController.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 VaultCraft 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. 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.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

VaultCraft has created permissionless DeFi infrastructure with the ability to deploy highly customizable yield bearing strategies on any EVM chain. To facilitate the needs for the ever changing landscape of DeFi, vault infrastructure was created to have the flexibility to cater to how protocols interact with such vaults while also being easy to manage by saving the need to develop new strategies with each protocol already deployed. In addition to this, vaults abstract away the complexity of interacting with protocols for users allowing them to deposit and withdraw using one interface.

Zokyo was tasked with the security review of the OracleVault, the PushOracle, the AsyncVault (and its dependencies) and the OracleVaultController. These vaults follow the ERC-7540 standard (Asynchronous ERC-4626 Tokenized Vaults) which allows users to deposit their tokens then process a withdrawal asynchronously. The OracleVault utilizes a Gnosis Safe in order to manage and hold assets while the PushOracle is used to track the value of assets locked within the Safe. The OracleVault also has the ability to set the price of the vault shares - permissioned keepers reserve the right to update such prices. There are some protections which protect user funds from significant price movements by attempting to update prices regularly however, should a significant price change occur in comparison to the previous price logged, the price will be updated but the vault will be paused immediately resulting in manipulations only being temporary.

From the user perspective, they have the ability to deposit their tokens into an AsyncVault which immediately transfers these tokens into the vault either through the mint or deposit functions. Should they wish to withdraw, they can submit a request through the requestRedeem function. From there, fulfillRedeem can be called which will eventually allow them to withdraw their tokens from the vault. Should the vault be paused by the contract authority, users will still be able to redeem their tokens. Note that the vault may charge performance fees, management fees or withdrawal incentives.

Overall the code is well written with detailed natspecs with sufficient unit testing. The issues discovered ranged between critical issues down to information which is advice given to reinforce best engineering practices. Most of these issues revolved around insufficient validation, logic issues and mathematical issues around fees.

It's recommended that the developers carefully read through the report and implement the various fixes suggested to which a fix review will take place to ensure that such fixes have been implemented correctly. In addition to this, whilst the code base has quite an extensive unit test suite through the foundry framework, its also recommended that some fuzz and integration testing is added to the repository in order to further fortify the security of the codebase.

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 VaultCraft team and the VaultCraft 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	Attacker Can Grief The Management Fee To Be Lesser Than Expected	Critical	Resolved
2	convertToAssets is receiving asset's decimals instead of share's decimals, possibly leading to a miscalculation	Critical	Acknowledged
3	A vault can be added to OracleVaultController after receiving deposits leading to incorrect price	High	Acknowledged
4	Insufficient Validation On Oracle Prices Could Lead to Stale Outputs	High	Acknowledged
5	Misconfiguration of Bounds Leading to Underpayment During Redeem Fulfillment	Medium	Resolved
6	No slippage protection can lead to loss of funds	Medium	Acknowledged
7	Fee On Transfer Tokens May Break Virtual Accounting When Withdrawing And Depositing	Medium	Acknowledged
8	The Redeem Function In BaseControlledAsyncRedeem Does Not Check That msg.sender Can Spend Owner Funds Using Allowance	Medium	Acknowledged
9	User's funds can get locked due to 'limits' having no maximum limit.	Medium	Resolved
10	Missing valid range check for signature	Medium	Resolved
11	Incorrect calculation of managementFee	Medium	Resolved
12	Use of Floating Solidity Version	Low	Acknowledged

#	Title	Risk	Status
13	Unbounded Gas Usage in Batch Updates	Low	Acknowledged
14	Potential Misconfiguration in setFees Function	Low	Acknowledged
15	Misconfiguration of Fees Leading to Excessive Charges	Low	Resolved
16	Redundant code and assumptions in tests can introduce bugs in the future	Informational	Acknowledged
17	Prices by PushOracle come from a centralized authority	Informational	Acknowledged
18	Functions Across The Targeted Codebase Performs Operations Directly On State Variables Which May Result In Costly Transactions	Informational	Acknowledged
19	Internal Functions Should Be Prefixed With _	Informational	Acknowledged

Attacker Can Grief The Management Fee To Be Lesser Than Expected

Fee is accrued when the share value increases / accrues value , then anyone can call the takeFees function which will accrue performance and management fee and send it to the fee recipient.

But an attacker can do the following →

1. Call takeFees even when the share value has not increased, therefore no fee shares would be minted to the fee recipient but the feesUpdatedAt parameter would be updated anyhow.

```
if (shareValue > fees_.highWaterMark) fees.highWaterMark = shareValue;

        if (fee > 0) _mint(fees_.feeRecipient, convertToShares(fee));

        fees.feesUpdatedAt = uint64(block.timestamp);
```

2. Now when the share actually accrues value and admin calls the takeFees function the management fee would be calculated as follows →

```
return
            managementFee > 0
                ? managementFee.mulDivDown(
                    totalAssets() * (block.timestamp -
fees_.feesUpdatedAt),
                    365.25 days // seconds per year
                ) / 1e18
                : 0;
```

We can see it is dependent on the difference between the current block timestamp and the feesUpdatedAt parameter which would be smaller than expected due to the attacker calling the takeFees function even when share value was same , if done perfectly the management fee can even be 0 if the attacker performs the takeFees call just before the share value would have increased (after a large deposit for example).

Recommendation:

Don't update the lastUpdatedAt if the share accrued no value.

convertToAssets is receiving asset's decimals instead of share's decimals, possibly leading to a miscalculation

When calculating how much worth a share is in the protocol (for example when calculating fees) we perform the following →

```
function _accruedPerformanceFee(
        Fees memory fees_
    ) internal view returns (uint256) {
        uint256 shareValue = convertToAssets(10 ** asset.decimals());
```

But the function convertToAssets is being passed one asset instead of one share , although this might not have a difference if both shares and asset have the same decimals but if the asset is for example USDC/UDST then the calculation would be incorrect , since we would be calculating of much 1e6 of a share is worth instead of 1e18 .

Recommendation:

Pass in one share instead of one asset to convertToAssets()

A vault can be added to OracleVaultController after receiving deposits leading to incorrect price

The function `addVault()` within the `OracleVaultController.sol` smart contract is used to add a vault to the controller to be able to update its price. The function always initialize the price to 1e18 (1:1) -- This is to prevent pausing the vault on the first update, so the `addVault()` function should be called before the vault has received any deposits. However, there not exist any check to ensure that there were no previous deposits when adding a new vault:

```
function addVault(address vault) external onlyOwner {
        highWaterMarks[vault] = 1e18;

        oracle.setPrice(vault, address(ERC4626(vault).asset()), 1e18, 1e18);


        emit VaultAdded(vault);
    }
```

As a result, a new vault with already assets deposited can be added, leading to recording and incorrect price.

Recommendation:

Implement a check to ensure that there are no assets within the vault (no previous deposits). If there are, revert.

Insufficient Validation On Oracle Prices Could Lead to Stale Outputs

The `PushOracle` contract allows for the setting of prices for base / quote pairs by permissioned entities via the `setPrice` and `setPrices` functions however, there is insufficient validation on the freshness of these prices which may lead to stale and/or invalid quotes when attempting to rely on the `PushOracle` as a price feed. Should there be a significant movement in price for the tokens utilising the price feed, such prices may be considered outdated which may result in the loss or theft of tokens for contracts integrating with the `PushOracle`.

Recommendation:

It's recommended that the prices set introduces a timestamp at which the price was updated alongside its price. In addition to this, sufficient checks should be made to ensure that these prices have been updated within a reasonable time frame which could be configurable by these permissioned entities.

Misconfiguration of Bounds Leading to Underpayment During Redeem Fulfillment

Description

The `AsyncVault` contract allows the owner to set upper and lower bounds using the `setBounds` function. These bounds are used in the `convertToLowBoundAssets` function to adjust asset calculations during redeem fulfillment. If the `bounds.lower` parameter is set to a value close to `1e18` (which represents 100% in the contract's context), the calculation can significantly reduce the amount of assets users receive when their redeem requests are fulfilled.

Scenario

1. Misconfiguration by Owner:
- The owner sets `bounds.lower` to a high value, such as `0.99e18` (99%).
- This setting implies that the vault should hold back 99% of the assets during redeem fulfillment.
2. User Redeem Request:
- A user requests to redeem 100 shares.
- Under normal circumstances, these shares might correspond to 100 assets.
3. Redeem Fulfillment:

The `convertToLowBoundAssets` function calculates the assets to return:

```
assets = totalAssets().mulDivDown(1e18 - bounds.lower, 1e18);
```

-

With `bounds.lower = 0.99e18`, the calculation becomes:

```
assets = totalAssets().mulDivDown(0.01e18, 1e18);\
```

- This results in only 1% of the expected assets being available for redemption.
4. User Receives Significantly Less Assets:
- The user receives only 1 asset instead of the expected 100 assets.
- The remaining 99 assets are effectively withheld due to the misconfigured bound.

Recommendation:

Restrict `bounds.lower` to a reasonable maximum value, such as `0.1e18` (10%), to prevent excessive withholding of assets.

Ensure that `bounds.lower` is less than or equal to a safe threshold that aligns with the vault's strategy and user expectations

MEDIUM-2 | ACKNOWLEDGED

No slippage protection can lead to loss of funds

Description

The `deposit()` function within the `BaseControlledAsyncRedeem.sol` smart contract is used for users to deposit assets in exchange for shares. The amount of shares minted to the users are calculated in the following way:

```
function convertToShares(uint256 assets) public view virtual returns (uint256) {
      uint256 supply = totalSupply; // Saves an extra SLOAD if totalSupply is
non-zero.

      return supply == 0 ? assets : assets.mulDivDown(supply, totalAssets());
  }
```

It can be seen that if the total number of assets increases, for the same supply, the amount of shares will decrease. This allows an attacker to front-run a deposit transaction in order to send assets to the contract so that the amount of received shares by the user is decreased in comparison to the expected amount.

Recommendation:

Add slippage control in order to avoid front-running issues. To implement an effective access control add a `minAmountOut` parameter and a `deadline` one.

Fee On Transfer Tokens May Break Virtual Accounting When Withdrawing And Depositing

Description

In DeFi tokenized vaults allow users to deposit their tokens and earn yield on those tokens deposited. These instances of ERC7540 make use of virtual accounting when attempting to calculate how many tokens are owed to the user when processing an asynchronous withdrawal. This is denoted in the internal `_fulfillRedeem` function within the `BaseControlledAsyncRedeem` contract when modifying claimable assets and shares as well as pending shares. Vault deposit and withdrawal work similarly to the ERC4626 standard however, commonly used tokens such as USDC and USDT have functionalities which enable those token owners to enable fees at any time which may cause breakages in the vault contract accounting.

Recommendation:

It is recommended that the balance of the deposited token is taken before and after critical function executions such as depositing, withdrawing, redeeming and minting to assert that the correct amount has in fact been transferred to the contract in addition to fees which may or may not exist. This could be done by using a modifier to make it widely accessible to multiple functions.

The Redeem Function In BaseControlledAsyncRedeem Does Not Check That msg.sender Can Spend Owner Funds Using Allowance

Description

The `redeem` function in the `BaseControlledAsyncRedeem` function allows users to claim their tokens by redeeming vault shares. The EIP-4626 (derived) standard enforces that when msg.sender creates a redeem transaction, there should be a check that funds can be spent by allowance. Currently, there is a check against `isOperator` to ensure that msg.sender has approval; however, there exists an assumption that the full balance for a user is approved which does not cater for cases where a user only wishes to approve half their balance.

```
function redeem(
    uint256 shares,
    address receiver,
    address controller
) public virtual override returns (uint256 assets) {
    require(
        controller == msg.sender || isOperator[controller][msg.sender],
        "ERC7540Vault/invalid-caller"
    );
    // ============= SNIP ============
```

Recommendation:

It's recommended that `isOperator` is replaced with an allowance which utilises a `uint256` as opposed to a `bool` to allow users to approve specific amounts to other users which removes the assumption that the whole balance has been approved forever.

User's funds can get locked due to 'limits' having no maximum limit.

The ´Limits´ struct within the `AsyncVault.sol` contract contains 2 variables:

1. `depositLimit`: Maximum amount of assets that can be deposited into the vault
2. `minAmount`: Minimum amount of shares that can be minted / redeemed from the vault

The owner can execute the `setLimits()` function to modify these limit variables.

```
function setLimits(Limits memory limits_) external onlyOwner {
    _setLimits(limits_);
}

/// @dev Internal function to set the limits
function _setLimits(Limits memory limits_) internal {
    emit LimitsUpdated(limits, limits_);

    limits = limits_;
}
```

The problem is that there are not any maximum upper or lower limits for these variables, meaning that they can be set to 0 or to 99999999999999999... leading to users not being able to deposit or withdraw their funds, therefore getting them locked in the contract.

Recommendation:

Define 2 immutable variables representing the maximum and minimum upper limits and check that the new limits set within the `setLimits()` function are between the mentioned limits variables.

Missing valid range check for signature

There is a missing check for a valid range of s in the authorizeOperator() function of the **BaseERC7540** contract

```
bytes32 r;
bytes32 s;
uint8 v;
assembly {
    r := mload(add(signature, 0x20))
    s := mload(add(signature, 0x40))
    v := byte(0, mload(add(signature, 0x60)))
}
```

The valid range for s is in 0 < s < secp256k1n ÷ 2 + 1
This means that the valid range for s must be in the "lower half" of the secp256k1 curve's order (n) to prevent signature malleability attacks. In other words:
0 < s < secp256k1n/2 + 1
Where
secp256k1n =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141

Recommendation:

Signature validation should include code :

```
if (uint256(s) > 0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF5D576E7357A4501DDFE92F46681B20A0)
 revert("Invalid s value");
}
```

For example refer:
https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol#L143

Incorrect calculation of managementFee

Description

The `_accruedManagementFee()` function within the `AsyncVault.sol` smart contract is used to calculate the managementFee. One of the parameters used for this calculator is the total amount of seconds per year:

```
function _accruedManagementFee(
        Fees memory fees_
    ) internal view returns (uint256) {
        uint256 managementFee = uint256(fees_.managementFee);

        return
            managementFee > 0
                ? managementFee.mulDivDown(
                    totalAssets() * (block.timestamp - fees_.feesUpdatedAt),
                    365.25 days // seconds per year
                ) / 1e18
                : 0;
    }
```

It can be observed that the amount of second per year is incorrect as it has been defined as '365.25 days' instead of '365 days'.

Recommendation:

Fix the amount of seconds per year using the exact correct amount in seconds: 31536000.

LOW-1 | ACKNOWLEDGED

Use of Floating Solidity Version

The contracts specify an unlocked (floating) Solidity version using pragma solidity ^0.8.25;. While this allows the contracts to compile with future Solidity versions, it also introduces potential risks. Any newly released compiler version could include breaking changes, vulnerabilities, or differences in behavior that may negatively affect the contract's functionality.

Locking the Solidity version to a specific compiler version ensures consistency and protects the contract from unanticipated changes or issues introduced in newer versions of the compiler. For instance, upgrades in Solidity may impact security features or gas optimizations.

Recommendation:
It is recommended to use a fixed Solidity version (e.g., pragma solidity 0.8.25;) to maintain stability and minimize unforeseen issues caused by compiler updates.

LOW-2 | ACKNOWLEDGED

Unbounded Gas Usage in Batch Updates

Description:
The `updatePrices` and `setLimits` functions process arrays in a loop, with no upper limit on the number of items. A large input could cause excessive gas usage, leading to out-of-gas errors or DoS.

Scenario:
A keeper or owner submits a batch update with an excessively large array of price updates. The transaction fails due to exceeding the gas limit, leaving other legitimate updates unprocessed.

Recommendation:
Set a reasonable maximum limit on the number of updates that can be processed in a single batch.

LOW-3 | ACKNOWLEDGED

Potential Misconfiguration in setFees Function

Description:
The setFees function allows the owner to set new fee rates. While there are checks to prevent setting fees above certain thresholds, there is a possibility of misconfiguration, such as setting the feeRecipient to the zero address.

Exploit Scenario:
The owner accidentally sets the feeRecipient to the zero address. As a result, when fees are taken, they might be sent to an unintended address or cause the transaction to fail.

Recommendation:
Enhance validation in the setFees function to prevent setting invalid fee parameters.

LOW-4 | RESOLVED

Misconfiguration of Fees Leading to Excessive Charges

Description:
The setFees function allows the owner to set fee rates for performance fees, management fees, and withdrawal incentives. While there are checks to prevent setting these fees above certain thresholds (20% for performance fee, 5% for management fee, and 5% for withdrawal incentive), these thresholds themselves may be too high and can lead to excessive fees being charged to users. Additionally, there is no mechanism to prevent frequent changes to the fees, which could be exploited to charge users unexpectedly.

Scenario

1. **Owner Sets Maximum Fees:**
    - The owner sets the performance fee to 20% (2e17), management fee to 5% (5e16), and withdrawal incentive to 5% (5e16).
    - These high fees significantly reduce user returns and may not align with user expectations or industry standards.

2. **Frequent Fee Changes:**
   - The owner repeatedly changes the fee structure without notice.
   - Users cannot predict or calculate their expected returns accurately.

Recommendation:

Reduce the maximum allowable fees to more reasonable levels (e.g., 2% management fee, 10% performance fee).

Introduce a timelock mechanism that delays fee changes, giving users time to react.

Redundant code and assumptions in tests can introduce bugs in the future

The following lines are redundant in the unit tests and fuzz tests and can be removed:

A) Test Contract Name: BaseControlledAsyncRedeemTest

1) Function Name: testFuzz_WithdrawWithOperator()

```
asset.mint(owner, redeemAmount);
asset.approve(address(baseVault), redeemAmount);
```

2) Function Name: testWithdrawWithOperator()

```
asset.mint(owner, redeemAmount);
asset.approve(address(baseVault), redeemAmount);
```

3) Function Name: testPartialFulfillRedeem()

```
asset.mint(owner, partialAmount);
asset.approve(address(baseVault), partialAmount);
```

4) Function Name: testRedeemWithOperator()

```
asset.mint(owner, redeemAmount);
asset.approve(address(baseVault), redeemAmount);
```

5) Function Name: testFuzz_RedeemWithOperator()

```
asset.mint(owner, redeemAmount);
asset.approve(address(baseVault), redeemAmount);
```

6) Function Name: testClaimableRedeemRequest()

```
asset.mint(owner, redeemAmount);
asset.approve(address(baseVault), redeemAmount);
```

7) Function Name: testMaxWithdraw()

```
asset.mint(owner, redeemAmount);
asset.approve(address(baseVault), redeemAmount);
```

8) Function Name: testMaxRedeem()

```
asset.mint(owner, redeemAmount);
asset.approve(address(baseVault), redeemAmount);
```

9) Function Name: testFulfillRedeem()

```
asset.mint(owner, redeemAmount);
asset.approve(address(baseVault), redeemAmount);
```

10) Function Name: testPartialFulfillRedeem()

```
asset.mint(owner, partialAmount);
asset.approve(address(baseVault), partialAmount);
```

B) Test Contract Name: AsyncVaultTest

1) Function Name: testFulfillMultipleRedeems()

```
asset.approve(address(asyncVault), redeemAmount * 2);
```

Recommendation:

It is advised to remove the above lines from the said functions as these might be based on assumptions and can introduce unnecessary assumptions or vulnerabilities in future.

INFORMATIONAL-2 | ACKNOWLEDGED

Prices by PushOracle come from a centralized authority

The `setPrice()` function within the `PushOracle.sol` smart contract is used only by the owner to set and define the prices for base/quote assets. This means that the prices which are obtained and handled by the `PushOracle.sol` contract may come from a centralized authority or from an 'unknown' party within the scope of this contract.

```
function setPrice(
    address base,
    address quote,
    uint256 bqPrice,
    uint256 qbPrice
) external onlyOwner {
    _setPrice(base, quote, bqPrice, qbPrice);
}
```

Recommendation:

It is recommended to not used centralized oracles but instead use a decentralized one like Chainlink Price Feeds.

INFORMATIONAL-3 | ACKNOWLEDGED

Functions Across The Targeted Codebase Performs Operations Directly On State Variables Which May Result In Costly Transactions

Description

There are various functions such as `_fulfillRedeem` in the `BaseControlledAsyncRedeem` contract which performs operations directly on `currentBalance` , a storage variable. Each storage variable read (opcode `sload`) can become quite gas intensive when operated on multiple times. When a storage variable is read for the first time (cold access), this will cost 2,100 gas units to execute the transaction. Each time after that (warm access) this will cost an additional 100 gas units thereafter. This was identified in the `BaseControlledAsyncRedeem` contract mainly around the usage of `currentBalance` being defined as a storage variable within certain functions.

Recommendation:

It's recommended that the desired storage variables are cached into a memory variable and updated once all operations are complete as memory variables only cost 3 cast units for each read.

INFORMATIONAL-4 | ACKNOWLEDGED

Internal Functions Should Be Prefixed With _

Recommendation:

Internal and private functions for example `handleWithdrawalIncentive` and `beforeFulfillRedeem` which should be prefixed with an underscore (_). It is recommended that such functions be modified in order to improve the readability of the codebase.

	• **src/vaults/multisig/phase1/AsyncVault.sol** • **src/vaults/multisig/phase1/BaseControlledAsyncRedeem.sol** • **src/vaults/multisig/phase1/BaseERC7540.sol** • **src/vaults/multisig/phase1/OracleVault.sol** • **src/peripheral/oracles/adapter/pushOracle/PushOracle.sol** • **src/peripheral/oracles/OracleVaultController.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

We are grateful for the opportunity to work with the VaultCraft 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 VaultCraft team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo