PUBLIC

Code Assessment

# of the UltraYield Smart Contracts

May 08, 2025

Produced for

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Ultra team,

Thank you for trusting us to help UltraYield with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of UltraYield according to Scope to support you in forming an opinion on their security risks.

Ultra implements UltraVault, an ERC-7540 compliant, managed vault with asynchronous redemption.

The most critical subjects covered in our audit are asset solvency, functional correctness, and compliance with standards. After the review, security regarding all the aforementioned subjects is high. A critical issue leading to the free minting of shares was uncovered in the first review of the codebase and was addressed in the first round of fixes, see Shares can be minted for free. In Version 2 there was an issue regarding the initialization of the system, see Order of Deployment and Initialization.

The general subjects covered are code complexity, gas efficiency, trustworthiness, documentation and specification. Security regarding all the aforementioned subjects is generally good. No specifications were provided regarding management of the funds. Note the strong assumptions made in the Trust Model.

In summary, we find that the codebase provides a good level of security.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered, and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	0
Medium-Severity Findings	4
• Code Corrected	2
• Specification Changed	2
Low-Severity Findings	16
• Code Corrected	15
• Specification Changed	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the UltraYield repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	31 March 2025	7ee804c2c894d082c0cc5397a313a9df68b59f15	Initial Version
2	23 April 2025	fda6efe9b1388d9b4eba08db3e7f681d88efcaf2	Version with Fixes
3	28 April 2025	6f8fa5a77ccf98e266fb0aa6f569aabc3c5e35f3	Additional Fixes
4	08 May 2025	4c5b2d3c302bd884216d891f3cc04a2da5d65bb7	Added events and initial balances

For the Solidity smart contracts, no compiler version has been fixed. The `pragma ^0.8.28` in some files permits any `0.8.x` version ≥`0.8.28`. After Version 2, the compiler version was fixed to `0.8.28`.

The following files are in scope of this review:

```
src:
    vaults:
        AsyncVault.sol
        BaseControlledAsyncRedeem.sol
        BaseERC7540.sol
        UltraVault.sol
    oracles:
        OracleAdmin.sol
        VaultPriceManager.sol
        UltraVaultOracle.sol
        ScaleUtils.sol
    utils:
        InitializableOwnable.sol
        Pausable.sol
```

In Version 2, the scope has been updated as follows:

Removed:

```
src/oracles/ScaleUtils.sol
```

Added:

```
src/utils/FixedPointMathLib.sol
```

(only the `mulDivDown()` and `mulDivUp()` functions)

2.1.1 Excluded from scope

Any file not listed above is out of the scope of this review, especially the functions of `FixedPointMathLib` not listed above. In particular, third-party libraries, like `solmate`, `solady`, `openzeppelin-contracts` or `openzeppelin-contracts-upgradeable`, are out of the scope of this review and are assumed to work correctly.

This review focused on the implementation of the abstract contracts (`AsyncVault, BaseControlledAsyncRedeem, BaseERC7540`) within `UltraVault`. Use of any of the abstract contract outside of `UltraVault` is out of scope and different behaviors than the ones described in System Overview are expected.

Exploits arising from incorrect oracle pricing are considered out of scope for this review.

The migration process and the computation of the number of migrated shares introduced in Version 4 are out of the scope of this review.

2.2 System Overview

This system overview describes the initially received version (**Version 1**) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Ultra has developed UltraVault, an ERC-7540 compliant vault system that includes a user-facing vault contract as well as supporting oracle and management contracts. ERC-7540 extends the ERC-4626 standard by supporting asynchronous actions. In the UltraVault only redemptions are asynchronous: users must first request a redemption, this request must be fulfilled before the withdrawal can be completed. Share pricing is determined by the external oracle contract UltraVaultOracle which relies on manually set prices updated by trusted entities. The vault's assets are managed by an external trusted fundsHolder. Oracle updates are managed through either one of two contracts: OracleAdmin and VaultPriceManager.

The contracts are not upgradeable, but the respective owner can update some of the key parameters.

2.2.1 UltraVault

UltraVault is a non-upgradable ERC-7540 vault (asynchronous ERC-4626 tokenized vault). Deposits are immediate, while redemptions are asynchronous. Funds deposited into the vault are forwarded to an external `fundsHolder` address, which is responsible for managing the assets and expected to be a multisig. The vault's share price is determined by an oracle contract. Three types of fees are charged in the vault: a performance fee, a management fee and a withdrawal fee.

Core parameters of the vault are the owner, underlying asset, token metadata, fee configuration and operational limits such as the maximum deposit amount and minimum amount for minting or redeeming. Most of these can be updated by the owner, except for the vault's underlying asset and token metadata.

The vault contract uses a layered implementation approach: UltraVault inherits from AsyncVault, which inherits from BaseControlledAsyncRedeem, which in turn extends BaseERC7540, built on top of the ERC-4626 standard.

UltraVault implements the specific logic related to the `fundsHolder` and the oracle. It overrides `totalAssets()` to fetch and calculate the total assets held by the vault based on the shares supply and its valuation. After a deposit, the contract forwards funds to the `fundsHolder`. Before a redemption is fulfilled, it pulls the necessary funds from the `fundsHolder` back into the vault. Withdrawals are not possible if this is not successful. `collectWithdrawalFee()` is overridden to transfer the fee from the `fundsHolder` to the `feeRecipient`.

The vault includes functions to update the `oracle` and `fundsHolder` using a two-step delayed process. The `owner` must first call `proposeOracle()` or `proposeFundsHolder()` to initiate a change. After a hardcoded delay of 3 days, the `owner` can finalize the change by calling `acceptProposedOracle()` / `acceptProposedFundsHolder()`. When updating the `fundsHolder`, `acceptProposedFundsHolder()` automatically pauses the vault, and it must be manually unpaused afterward. While the vault is paused, deposits are disabled. The proposals can be overwritten as long as they are running, and the timestamp is reset.

AsyncVault handles the logic for fees and operational limits. It provides wrapper functions for `deposit`, `mint`, `withdraw`, and `redeem` that operate on `msg.sender`. It implements `fulfillRedeem()` and `fulfillMultipleRedeems()` for processing redemption requests and exposes a public helper function `convertToLowBoundAssets()`. Provided the `fundsHolder` contract holds sufficient funds and the vault has the necessary allowance to pull them, anyone can fulfill a pending redemption request. In practice, this is expected to be performed by the system operator who makes the funds available e.g. by retrieving them from external investments and grants the required allowance.

The vault owner can change two key limits: the minimum amount required for a deposit or redeem operation (`minAmount`), and the `_depositLimit` which caps the total number of assets that can be deposited. Note that `totalAssets` may still increase beyond the cap due to changes in the share price determined by the oracle.

Fees are collected whenever funds move in or out of the system. A view function `accruedFees()` returns the currently outstanding performance and management fees. The following types of fees exist, the parameters can be updated by the `owner`:

- The performance fee is capped at 30% and calculated using a high water mark model based on the maximum recorded share price. The water mark remains even if the fee percentage is changed.
- The management fee is capped at 5% per year and accrues over time.
- The withdrawal fee is capped at 1%.

All fee values are defined in fixed-point format where 1e18 represents 100%.

BaseControlledAsyncRedeem implements the core asynchronous redemption flow. It keeps track of pending and claimable redemption requests and provides the main user-facing vault functions: `deposit(uint256 assets, address receiver)`, `mint(uint256 shares, address receiver)`, `redeem(uint256 shares, address receiver, address controller)` and `withdraw(uint256 assets, address receiver, address controller)`. These can be used to finalize withdrawals after a redemption request has been fulfilled.

Helper functions are provided to query redemption state: `getPendingRedeem()`, `getClaimableRedeem()`, `pendingRedeemRequest()` and `claimableRedeemRequest()`. When a user requests a redemption, they specify a controller address, which will own that redemption. The use of the term controller (instead of owner) is defined by ERC-7540. Redemptions must first be requested using `requestRedeem()` and be fulfilled before the funds can be withdrawn from the system. A per-controller timestamp (`requestTime`) is updated every time a new redemption is requested. A pending request can be cancelled by the controller or its operator by using `cancelRedeemRequest()` only if it has not yet become claimable.

The vault implements the full ERC-7540 and ERC-4626 interfaces. However, functions like `previewWithdraw()` and `previewRedeem()` are not supported due to the asynchronous nature of the flow.

The base class BaseERC7540 provides pause/unpause functionality for the `owner` and `pauser` role, as well as a role system that allows the `owner` to update roles via `updateRole()` (though only the `PauserRole` is defined). It implements operator authorization following the `IERC7540Operator` interface, which defines `setOperator()` and `isOperator()` (a public getter for the internal authorization mapping), as well as the corresponding `OperatorSet` event. In addition, BaseERC7540 extends this functionality with support for signature-based operator authorization via `authorizeOperator()`, where the signature input format is as follows: `r||s||v`. Signatures follow the EIP_712 standard and include a deadline. Operator approval statuses are stored in the `isOperator` mapping. Nonces are used to prevent replay attacks in `authorizeOperator()`, ensuring that once an operator is revoked, they cannot reuse a past signature to reauthorize themselves.

UltraVault supports ERC-165 and declares support for the following interfaces: IERC757, IERC7540Operator, IERC165, and IERC7540Redeem.

2.2.2 Oracles

UltraVault share prices are determined by the **UltraVaultOracle**, which maintains manually set prices for base (= vault shares) / quote (= vault's underlying asset) pairs. The contract is permissioned and only allows updates from the designated owner address. Prices can be set using `setPrice()` / `setPrices()` or configured to change gradually over a timespan to a target price using `scheduleLinearPriceUpdate()` / `scheduleLinearPricesUpdate()`. These scheduled updates apply linear vesting from the current price to the specified target price over a defined time period. The oracle exposes functions `getCurrentPrice()` which returns the current price based on either the stored value or the active linear update and `getQuote()` which calculates an output amount based on the current price.

Two helper contracts are used for price updates:

OracleAdmin is a simple contract that forwards calls to the UltraVaultOracle. It exposes the same external interfaces for price updates as the oracle itself. To work, it must be set as the owner in the oracle. The contract defines two roles: an owner and an admin. Both can trigger price updates. These roles are changeable — the admin can be updated directly by the owner, while ownership transfers follow a two-step process involving `claimOracleOwnership()`.

VaultPriceManager is a contract for managing share prices of multiple UltraVaultOracle instances. To work, it must be set as the owner in each UltraVaultOracle it manages and have the `PauserRole` in each vault it is linked to. The vault must be added prior to the first deposit. The contract defines two roles: an owner and admins, with admins assigned per vault. Admins are tracked via a mapping, vaults can have multiple enabled admins. Both can trigger price updates, all updates are subject to limits set by the owner. These limits enforce checks on sudden price changes (drops/increases) and a maximal drawdown check based on a high water mark that keeps track of the highest price. If one of the limits is exceeded, the oracle will pause the target vault. The owner can update these limits at any time. Price updates are triggered through the exposed functions `updatePriceInstantly()` and `updatePriceWithVesting()`, which apply the changes to the connected UltraVaultOracle.

2.2.3 Trust Model

UltraVault:

Owner: Owner of the vault. Can propose changes to fees, limits (`maxDeposit`, `minAmount`), the `fundsHolder` or the `oracle` address. Changes to the `oracle` and `fundsHolder` are subject to a 3-day time delay and must be explicitly accepted. Fully trusted. If compromised, the owner can manipulate share valuation or redirect funds.

FundsHolder: External address holding and managing the vault's assets. Expected to be a multisig. Trust level: fully trusted. In the worst case, it can withhold or steal all funds.

PauserRole: Can pause/unpause vault operations such as deposits and mints. Partially trusted.

Oracle: Smart contract responsible for providing the share price of the vault. The oracle receives off-chain updates from trusted parties (e.g. a curator), who track portfolio performance and update the price based on their estimates. Fully trusted by the UltraVault. Expected to be the UltraVaultOracle managed by either the OracleAdmin or VaultPriceManager contract. While pricing can never be perfect, it must remain within a reasonable approximation of fair value. If compromised or misconfigured, it can cause mispriced redemptions and dilute or overpay investors.

User-Related Roles:

Owner of vault shares: Arbitrary User, untrusted.

Controller: ERC-7540 terminology for the address managing a specific redemption request. When a share owner requests redemption, they assign a controller to handle it. Any address can act as a controller. Untrusted by the system.

Operator: Authorized by a controller to perform redemption-related actions on their behalf. Untrusted by the system but fully trusted by the controller who granted the authorization.

Generally, users investing in the vault must trust the off-chain fund management, which is outside the scope of this contract. The vault relies on the fundsHolder to provide liquidity for redemptions. If the fundsHolder does not cooperate, users will be unable to redeem their vault shares. The funds holder is trusted to release the funds within an acceptable timeframe and is trusted to offer a fair share price, i.e. not wait the end of a linear price decrease on purpose to release funds. The asset is expected to be a standard ERC-20 token with no fees on transfer, rebasing, blacklisting, or any other non-standard behavior that could affect operations.

OracleAdmin:

Owner: Can update the admin, transfer ownership (via a two-step process), and update prices. Admin: Can update prices.

Both the owner and the admin must be fully trusted, as the UltraVault depends on the share price provided by the oracle. If compromised, both can manipulate share valuation and potentially steal funds.

VaultPriceManager:

Owner: Can update the admin, transfer ownership, update prices, and set the limits that govern price updates. Admin: Can update prices, but only within the limits set by the owner.

The owner must be fully trusted, as he controls the limits and hence can ultimately manipulate pricing. The admin is semi-trusted, restricted by those limits.

Vault Proxy

Starting Version 2, the vault is used behind a proxy. The owner of the proxy must be fully trusted as they can change the contract's implementation.

2.2.4 Changes in Version 2

- the vault has been updated to support upgradeability and uses OpenZeppelin's ERC4626Upgradeable as a base contract
- the limits on the vault (`minAmount` and `_depositLimit`) have been removed
- when starting a linear price update, the duration of the update is passed instead of the end timestamp, the duration is required to be between 23h and 60 days
- a function `referDeposit()` has been added in the vault to allow a user to be referred by another user when depositing
- the proposals to update the funds holder and the oracle have an expiry set at 7 days
- the function `authorizeOperator()` has been removed

2.2.5 Changes in Version 3

- The proxy pattern has been changed to UUPS. UltraVault now inherits OpenZeppelin’s UUPSUpgradeable and overrides `_authorizeUpgrade()` to enforce access control.

2.2.6 Changes in Version 4

- The function `setupInitialBalances()` was added in `UltraVault` as a way for Ultra to migrate positions from an older vault. It's purpose is to allow the owner to mint arbitrary shares to arbitrary addresses. The function can be called as long as the total supply of the vault is `0`, the function can be called even if the contract is paused.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Open Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	0

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Open Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	1

- Shares Can Be Minted for Free Code Corrected

High-Severity Findings	0

Medium-Severity Findings	4

- Order of Deployment and Initialization Code Corrected
- Price Computation Can Be Simplified Code Corrected
- maxMint/maxDeposit Overestimate Available Capacity Due to Ignored Fees Specification Changed
- previewDeposit/previewMint Incorrectly Account for Deposit Limits Specification Changed

Low-Severity Findings	16

- Disable Initializer on Implementation Code Corrected
- Missing Events Code Corrected
- CEI Pattern Code Corrected
- Dead Code Code Corrected
- Discrepancy When Burning Shares on fulfillRedeem Code Corrected
- Duplicated Code Code Corrected
- Event FeesRecipientUpdated Not Emitted Code Corrected
- Events Emitted on No Update Code Corrected
- Expiry for Update Proposals Code Corrected
- Floating Pragma Version Code Corrected
- Frontrunning Fulfillment of Redemptions Specification Changed
- Missing Input Sanitization Code Corrected
- System Is Not Paused After Deployment Code Corrected
- Test Code Should Not Impact Production Code Code Corrected
- Timestamp for Linear Price Update Can Be in the Past Code Corrected
- Wrong Value for Shares in Deposit Hooks Code Corrected

Informational Findings	3

- Inconsistent Rounding Direction on Fees Code Corrected
- Malleable Signature Specification Changed
- Outdated Imports and Comments Code Corrected

6.1 Shares Can Be Minted for Free

Correctness Critical Version 1 Code Corrected

CS-EUY-001

In the function `BaseControlledAsyncRedeem.mint()`, `previewMint()` is used to compute the number of assets that must be transferred from the caller, based on an amount of shares. The implementation of `previewMint()` returns 0 if the amount of shares is below the `minAmount` or if the number of assets after the `mint` will exceed the `depositLimit`.

This can be abused by, for example, calling `mint()` with a number of shares that is below the `minAmount`. This will result in 0 assets being transferred from the caller, and the shares will be minted for free.

Code corrected:

The minimum and maximum deposit limits have been removed from the system.

6.2 Order of Deployment and Initialization

Design Medium Version 2 Code Corrected

CS-EUY-028

The current setup requires the vault to be added in the oracle before it is initialized, because `_setFees()` needs to compute the initial high water mark. This forces the deployment and initialization of the vault to be done in two steps, opening the way for a potential front running of the `initialize()` call on the vault.

Code corrected:

The calculation of the initial high water mark when initializing the contract and setting fees has been simplified to one token of the vault's asset, the initial price.

6.3 Price Computation Can Be Simplified

Design Medium Version 1 Code Corrected

CS-EUY-002

The computation done in `UltraVaultOracle._getQuote()` can be simplified, as `ScaleUtils` is mainly used to pack and unpack values. This can be avoided and simply use part of the logic of `ScaleUtils.calcOutAmount()`. This would make the code more straightforward and save gas.

Code corrected:

The `ScaleUtils` library has been removed and the price computation logic has been simplified to:

```
(inAmount * price * (10 ** (quote_decimals))) / (10 ** (base_decimals + 18))
```

6.4 maxMint/maxDeposit Overestimate Available Capacity Due to Ignored Fees

Correctness Medium Version 1 Specification Changed

CS-EUY-003

According to the EIP-4626 standard, maxMint and maxDeposit must return the maximum amount of shares that can be minted or assets that can be deposited. It explicitly states: "MUST NOT be higher than the actual maximum that would be accepted (it should underestimate if necessary)."

The UltraYield vault collects fees on minting and deposits, which are emitted as newly minted vault shares.

Since the vault enforces a `depositLimit` (based on total assets, derived from the total amount of shares and the oracle price), these fee shares consume part of the available deposit capacity.

Therefore, unless the deposit limit is paused or disabled, the values returned by `maxMint()` and `maxDeposit()` may slightly overestimate the actual available room if there are unrecognized fees yet to be realized. This breaks compliance with the ERC-4626 standard.

Specification changed:

The minimum and maximum deposit limits have been removed from the system. Without these limits, the issue described above no longer exists.

6.5 previewDeposit/previewMint Incorrectly Account for Deposit Limits

Correctness Medium Version 1 Specification Changed

CS-EUY-004

According to the EIP-4626 standard, previewDeposit and previewMint are intended to simulate the result of a deposit or mint operation **without** enforcing or considering vault-specific limits.

The spec explicitly states: "MUST NOT account for deposit limits like those returned from maxDeposit and should always act as though the deposit would be accepted, regardless of whether the user has enough tokens approved, etc."

It further states: "MUST NOT revert due to vault specific user/global limits. MAY revert due to other conditions that would also cause deposit to revert."

However, the current implementation in AsyncVault.sol factors in such limits and hence violates this specification.

Specification changed:

The minimum and maximum deposit limits have been removed from the system.

6.6 Disable Initializer on Implementation

Security Low Version 2 Code Corrected

CS-EUY-027

UltraVault has been made upgradable in Version 2. The logic to initialize the contract has therefore been moved from the constructor into initialize functions which are intended to be executed in the context of the proxy.

These initialize functions are permissionless but can only be executed once, until the initialized flag is set. Note that they can also be called on the implementation contract. While this does not have any adverse effect it is generally recommended not to leave the implementation contract uninitialized.

It is recommended to invoke `_disableInitializers()` in the constructor of the implementation to automatically lock the initialize functions in the implementation contract.

Code corrected:

The constructor of the UltraVault now invokes `_disableInitializers()` which locks the initialize functions in the implementation contract.

6.7 Missing Events

Design Low Version 2 Code Corrected

CS-EUY-026

The following important state changes are missing events:

1. In `AsyncVault.initialize()`, the `FeesRecipientUpdated` should be emitted as the fee recipient is set
2. In `UltraVault.referDeposit()`, an event should be emitted when a user is referred

---

Code corrected:

The missing events have been added.

6.8 CEI Pattern

Design Low Version 1 Code Corrected

CS-EUY-005

Some functions in the codebase do not follow the Check-Effect-Intercation pattern:

- In the function `BaseControlledAsyncRedeem._requestRedeem()`, the transfer of assets should be done after updating the `pendingRedeem` in the storage
- In the function `BaseControlledAsyncRedeem._cancelRedeemRequest()`, the transfer of assets should be done after updating the `pendingRedeem` in the storage

---

Code corrected:

In both functions the transfer of assets has been moved after updating the `pendingRedeem` in the storage.

6.9 Dead Code

Design Low Version 1 Code Corrected

CS-EUY-006

For the sake of code readability and maintainability, unused code should be removed. The following code is unused:

- the whole `ReentrancyGuard` logic
- the errors `ZeroAmount` and `InvalidFee` in `AsyncVault` are never used

Code corrected:

The unused code has been removed.

6.10 Discrepancy When Burning Shares on `fulfillRedeem`

Design Low Version 1 Code Corrected

CS-EUY-007

There is a discrepancy in the order of operations between `fulfillRedeem()` and `fulfillMultipleRedeems()`. In `fulfillRedeem()`, the shares are burned before the redeem is fulfilled, while in `fulfillMultipleRedeems()`, the shares are burned after the redeem is fulfilled. This could lead to inconsistencies in the order of emitted events.

Code corrected:

The shares are burned after the call to `_fulfillRedeem()` in `fulfillRedeem()`. This matches the order of operations in `fulfillMultipleRedeems()`.

6.11 Duplicated Code

Design Low Version 1 Code Corrected

CS-EUY-008

For the sake of code readability and maintainability, available logic should be reused instead of being duplicated. The following logic is duplicated in the codebase:

- The logic of the function `convertToLowBoundAssets()` is the same as `convertToAssets()`

Code corrected:

`convertToLowBoundAssets()` has been removed and all calls have been replaced with `convertToAssets()`.

6.12 Event FeesRecipientUpdated Not Emitted

Design Low Version 1 Code Corrected

CS-EUY-009

AsyncVault.sol defines the `FeesRecipientUpdated` event. However, this event is never emitted.

`setFeeRecipient()` updates the fee recipient but does not emit any event. The emission of `FeesRecipientUpdated` appears to be missing.

Code corrected:

The event `FeesRecipientUpdated` is emitted in `setFeeRecipient()` when the fee recipient is updated.

6.13 Events Emitted on No Update

Design | Low | Version 1 | Code Corrected

CS-EUY-010

Some events indicating a state update are emitted even when the storage is not updated. Functions emitting those events are:

- `BaseERC7540.authorizeOperator()` with the `OperatorSet` event
- `BaseERC7540.setOperator()` with the `OperatorSet` event
- `BaseERC7540.updateRole()` with the `RoleUpdated` event
- `VaultPriceManager._setLimits()` with the `LimitUpdated` event
- `VaultPriceManager.setAdmin()` with the `AdminUpdated` event
- `OracleAdmin.setAdmin()` with the `AdminUpdated` event

Code corrected:

The code has been modified to emit the events above only when the storage is updated.

6.14 Expiry for Update Proposals

Design | Low | Version 1 | Code Corrected

CS-EUY-025

The proposals for updating the oracle and the funds holder are pending until the proposal is executed. There is no expiry, proposals can remain in this pending state indefinitely while users interact with the system normally. This undermines the intention behind the two-step mechanism with a time delay, which is meant to prevent unexpected configuration changes from taking effect immediately.

Code corrected:

An expiration time has been added to the proposals, which will be set to 1 week after the proposals are created, so the proposals are executable during 4 days. If the proposal is not executed within that timeframe, it cannot be executed anymore and a new one must be created.

6.15 Floating Pragma Version

Design Low Version 1 Code Corrected

CS-EUY-011

The compiler version is not fixed in the project (`^0.8.28`). It is good practice to use a fixed compiler version in order to avoid unexpected behaviors if compiled with a different version.

Code corrected:

The compiler version has been fixed to `0.8.28`.

6.16 Frontrunning Fulfillment of Redemptions

Security Low Version 1 Specification Changed

CS-EUY-012

After a redemption is requested via `requestRedeem()`, it must be fulfilled by executing `_fulfillRedeem()` which can be done permissionlessly. This succeeds if the `beforeFulfillRedeem()` hook is able to pull the required funds from the `fundsHolder`, making them available in the contract for completing the redemption.

Successful fulfillment requires two conditions:

1. Sufficient funds held by the `fundsHolder`
2. Sufficient allowance granted to the UltraVault to transfer those funds

Our understanding of the intended mode of operation is that the vault operator makes funds available on demand by setting the necessary allowance for each redemption request. The UltraVault is not granted an unlimited allowance which would allow to use funds currently held at the address, e.g., after deposits.

Since the fulfillment process is not tied to a specific redemption request, once funds and allowance are provided, any redemption can be fulfilled. An observer could frontrun this by submitting their own `fulfillRedeem()` call for a different request (maybe one that has just been created in the very same transaction), consuming the available funds. As a result, the originally intended redemption cannot be fulfilled until the funds and/or allowance are replenished.

This creates a frontrunning vector that can disrupt orderly redemption processing.

Specification changed:

The function `_fulfillRedeem()` has been changed to require that the caller has the `OPERATOR` role, ultimately making the functions `fulfillRedeem()` and `fulfillMultipleRedeems()` permissioned.

6.17 Missing Input Sanitization

Design Low Version 1 Code Corrected

CS-EUY-013

1. The function `InitializabelOwnable.initOwner()` does not check `_newOwner` for `address(0)`
2. The constructor of `OracleAdmin` does not check `_oracle` for `address(0)`
3. The constructor of `VaultPriceManager` does not check `_oracle` for `address(0)`

4. The constructor of `AsyncVault` does not check `_feeRecipient` for `address(0)`
5. The function `VaultPriceManager._setLimits` does not enforce a lower bound on the limits
6. The length of the `signature` in `BaseERC7450.authorizeOperator` is not checked to be `65` bytes long

After Version 2:

1. The address `_asset` is not checked for `address(0)` in `BaseERC7540.initiallize()`

Code corrected:

1. FIXED
2. NOT FIXED
3. FIXED
4. FIXED
5. OK: Ultra does not exclude the limits to be 0
6. FIXED: code removed

After Version 2:

1. FIXED

6.18 System Is Not Paused After Deployment

Design Low Version 1 Code Corrected

CS-EUY-014

When the oracle or the funds holder are changed in the UltraVault, the system is paused for manual inspection by operators. This should also happen after deployment/initialization of the new contract. The system should be paused until the operators have confirmed that everything is working as expected.

Code corrected:

The vault is now set to paused during initialization.

6.19 Test Code Should Not Impact Production Code

Design Low Version 1 Code Corrected

CS-EUY-015

The function `VaultPriceManager.addVault()` is requiring the total supply of the vault to be `<= 1` instead of `< 1` as a hack for the tests to work. As a good practice, production code should not be tailored to make tests pass, the tests should be adapted instead.

Code corrected:

The function `VaultPriceManager.addVault()` has been updated to require the vault's total supply being `0`.

6.20 Timestamp for Linear Price Update Can Be in the Past

Design | Low | Version 1 | Code Corrected

CS-EUY-016

Using a linear price update allows to have a linear transition from a start price to a target price. The function `UltraVaultOracle._scheduleLinearPriceUpdate()` allows for prices to be in the past (`< block.timestamp`), applying such a timestamp boils down to applying the target price instantly, which goes against the purpose of the linear price update.

Code corrected:

The function `UltraVaultOracle._scheduleLinearPriceUpdate()` has been updated to receive the duration of the linear price update instead of the end timestamp. The duration is enforced to be between 23 hours and 60 days and is added to `block.timestamp` to calculate the end timestamp. This forces the end timestamp to be in the future.

6.21 Wrong Value for Shares in Deposit Hooks

Correctness | Low | Version 1 | Code Corrected

CS-EUY-017

In the functions `BaseControlledAsyncRedeem.mint()` and `BaseControlledAsyncRedeem.deposit()`, the hook `beforeDeposit(assets, shares)` is called before `assets` in the former case and `shares` in the latter are computed. If the hook was actually using the `assets` and `shares` values, one of them would be zero.

Code corrected:

The call to the hook has been moved after the computation of `assets` and `shares`.

6.22 Inconsistent Rounding Direction on Fees

Informational | Version 1 | Code Corrected

CS-EUY-024

When the performance fee is calculated, it is rounded up, but management and withdrawal fees are rounded down.

Code corrected:

The performance fee calculation now rounds down making the rounding direction consistent across all fees.

6.23 Malleable Signature

Informational | Version 1 | Specification Changed

CS-EUY-020

The signature accepted by `BaseERC7540.authorizeOperator()` is malleable as `s` is not checked for being in a particular half of the curve. The signature cannot be replayed since it contains a nonce, but it is usually good practice to follow industry standard, which enforces that `s` is in the lower half of the curve, see https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/utils/cryptography/ECDSA.sol#L143.

Specification changed:

The function `BaseERC7540.authorizeOperator()` has been removed. The only way to set an operator is through the `setOperator()` function.

6.24 Outdated Imports and Comments

Informational | Version 1 | Code Corrected

CS-EUY-021

`BaseERC740.sol` contains leftover elements from earlier development iterations that are no longer relevant in the current version of the code:

- A comment referencing "EIP-7441 LOGIC" appears above a section unrelated to that EIP. These remnants appear to be informational leftovers from intermediate development stages.

`UltraVaultOracle.sol` contains leftover elements from earlier development iterations that are no longer relevant in the current version of the code:

- The `Math` library is imported but never used

Code correct:

- The comment mentioning "EIP-7441 LOGIC" was removed along with the associated code.
- The `Math` library import was removed.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Fee Collection Increases Vault's Total Assets

Informational **Version 1** **Risk Accepted**

CS-EUY-018

In UltraVault, fees are collected by minting additional shares unlike in similar vaults, where minting fees is expected to dilute the share value. Since `UltraVault` calculates `totalAssets()` based on the total number of shares and a manually set share price from an oracle, collecting fees increases `totalAssets()` . UltraVaults are centrally managed, and the share price oracle is manually controlled. The share price must reflect upcoming fee dilution. If fees are collected without the share price being adjusted accordingly, the vault may become insolvent, i.e., `totalAssets()` exceeds the actual assets held. Consider the following example to illustrate:

1. Vault has a 5% management fee and is initially empty.
2. Alice deposits 1000(e6) USDC and receives 1,000,000,000 shares.
3. One day later, share price is still 1:1. `collectFees()` is called.
4. Vault now has 1,000,136,986 shares and still only 1000(e6) USDC.
5. `totalAssets()` reports 1,000,136,986 USDC, but real assets are only 1000(e6) USDC.

Risk accepted:

Client states:

```
It’s part of the design to simplify calculations.
```

7.2 Gas Optimisations

Informational **Version 1** **Code Partially Corrected**

CS-EUY-019

1. Some of the storage variables can be made `immutable`. Examples are:

   1. `OracleAdmin.oracle`
   2. `VaultPriceManager.oracle`
   3. `BaseERC7540.share`

2. In the callpaths of `AsyncVault.previewDeposit()` and `AsyncVault.previewMint()`, the contract is checked to be paused twice, once in `AsyncVault` and a second time in `BaseControlledAsyncRedeem` (same functions). Therefore, the "happy path" consumes more gas than necessary.

3. In the function `AsyncVault._setFees()`, the branch `fees.highwaterMark == 0` is taken only when the contract is deployed, and then `totalSupply = 0`. In this case, the call to `convertToAssets()` is a no-op on its input and will always return `10**decimals`.
4. The function `UltraVaultOracle._getDecimals()` makes a static call to the token to check whether it implements `decimals()`. In the edge case where the token does not implement `decimals()` and has a state modifying fallback function, the static call will fail and consume all the gas it was given (63/64). Any callpath using `_getDecimals()` would become extremely expensive and could DOS the system.
5. The array received as parameters to `OracleAdmin.setPrices` and `OracleAdmin.scheduleLinearPricesUpdate` can stay in `calldata` as they are forwarded to the oracle right away.
6. The functions `withdraw` and `redeem` make a call to `_collectFees()`. This call is not necessary as no shares are minted nor burned during the callpaths.
7. The hook `afterDeposit()` is calling `_collectFees()`, this will almost always be a no-op as fees are already collected during the `beforeDeposit` hook. In the case where it is not a no-op, fee collection can be deferred to the next deposit or `fulfillRedeem` call. All in all, fee collection is not needed in the `afterDeposit` hook.
8. In the function `UltraVaultOracle._setPrice()`, the `targetPrice` can be set to `0`.

After Version 2

1. The `else` branch in `BaseERC7540.setOperator()` is not needed as `success` is initialized to `false` by default.

Code partially corrected:

1. FIXED: The two oracle addresses can now be updated hence they can no longer be made immutable. Because UltraVault is now upgradable in Version 2, `share()` must return the proxy's address. This means the storage variable initialized with `address(this)` cannot simply be declared as immutable as this would store the implementation contract address. The storage variable `share` has been replaced by a function `share` which returns `address(this)`, the address of the currently executing account. Invoked via the proxy, `share()` returns the proxy's address which is the address of the share token.
2. FIXED: The functions in `AsyncVault` have been removed.
3. NOT FIXED: even after the change to OpenZeppelin's implementation, this computation can still be simplified as the total supply will still be 0.
4. ACK: Ultra acknowledged the potential issue and states "Acknowledged, this corner case shouldn't affect the system".
5. FIXED: the parameters are now passed as `calldata`.
6. ACK: Ultra acknowledged the potential optimization and states "Acknowledged, this is helpful for our accounting purposes so this extra gas spend is justified"
7. FIXED: the call to `_collectFees()` has been removed.
8. FIXED: the `targetPrice` is now set to `0`.

After Version 2

1. FIXED: The `else` branch in `BaseERC7540.setOperator()` has been removed.

7.3 Idle Liquidity

Informational Version 1 Risk Accepted

CS-EUY-023

When depositing liquidity, a user sees their shares minted right away, but their liquidity will be idle until the `fundsHolder` actually invests the funds. During this time, the user is still earning a yield on their non-working liquidity. This could lead to actors depositing liquidity and then requesting a redemption shortly after, effectively stealing the yield of the working liquidity provider.

The profitability of such a scheme of course depends on the yield of the vault, the fees, and the frequency at which the funds holder invests the funds in a strategy.

Risk accepted:

Client states:

```
It is expected behaviour that the funds might not be allocated straight away.
We expect the vault curators to frequently monitor the deposits and timely allocate
the funds so decided not to add complexity to the smart contracts in this regard
```

7.4 `VaultPriceManager` and Vault's High Water Mark Can Be Out of Sync

Informational Version 1 Acknowledged

CS-EUY-022

When the price of a vault's share is updated by the `VaultPriceManager`, the vault's high water mark is not updated. This can lead to a situation where the vault's high water mark is not the same as the one from the vault.

Examples of situations where the marks are out of sync.

- When an upward linear price update is submitted, the high water mark in the `VaultPriceManager` is set to be the target price of the update. This can lead to a situation where the vault's high water mark is lower than the one from the `VaultPriceManager` until it reaches maturity, which is expected.
- If a price that exceeds the `jump` value is submitted, the high water mark in the `VaultPriceManager` is not updated. Now if `fulfillRedeem` (which collects the fees) is called while the system is paused, or the update was actually legit and the vault is unpaused, the vault's high water mark will be updated to the new price, but the `VaultPriceManager`'s high water mark will not be updated. This can lead to a situation where the vault's high water mark is higher than the one from the `VaultPriceManager`.

Acknowledged:

Ultra states:

```
This shouldn’t affect overall operations of the vault.
```

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Exchange Rate From ClaimableRedeem

Note **Version 1**

The exchange rate that can be computed from ClaimableRedeem is not necessarily the exchange rate that was applied during `_fulfillRedeem()`. It is effectively the weighted average of all the non-fully claimed fulfilled redemption requests for a particular `controller`.