Security Review Report
NM-0273 SWELL BTC LRT

NETHERMIND
SECURITY

(Aug 9, 2024)

Contents

1 Executive Summary

This document presents the security review performed by Nethermind Security for Swell smart contracts. The main purpose of the protocol is to enable the restaking of WBTC. The Swell team has extended the YearnV3 vault by integrating it with `AeraVaultV2` to achieve this. They developed a custom strategy for the vault and a `DelayedWithdraw` contract to manage withdrawals from the Yearn vault. Additionally, a `WithdrawLimitModule` was introduced to ensure all withdrawals are processed exclusively through the `DelayedWithdraw` contract. This architecture allows users to deposit WBTC into the Yearn vault, with assets subsequently restaked via the custom strategy into various restaking protocols, including `Symbiotic` and `EigenLayer`.

The security review focused on custom and modified contracts: `AeraStrategy`, `DelayedWithdraw`, `WithdrawLimitModule`, and changes to `RoleManager`. The Nethermind Security team also examined potential integration issues among these components, as well as with the `VaultV3` and `AeraVaultV2` contracts, assessing the integration of these components within the overall system and highlighted general weaknesses in the current design of Swell protocol. Note that the integration between `AeraVaultV2` and various restaking protocols will be addressed by the `Swell` team in a future development cycle and will require an additional security review.

The audit was performed using (a) manual analysis of the codebase, (b) automated analysis tools, (c) simulation of the smart contract, and (d) creation of test cases. **Along this document, we report** fifteen points of attention, where one is classified as `High`, three are classified as `Medium`, one is classified as `Low`, and ten are classified as `Informational` or `Best Practice`. The issues are summarized in Fig. 1.

This document is organized as follows. Section 2 presents the files in the scope. Section 3 summarizes the issues. Section 4 presents the system overview. Section 5 discusses the risk rating methodology. Section 6 details the issues. Section 7 discusses the documentation provided by the client for this audit. Section 8 presents the compilation, tests, and automated tests. Section 9 concludes the document.

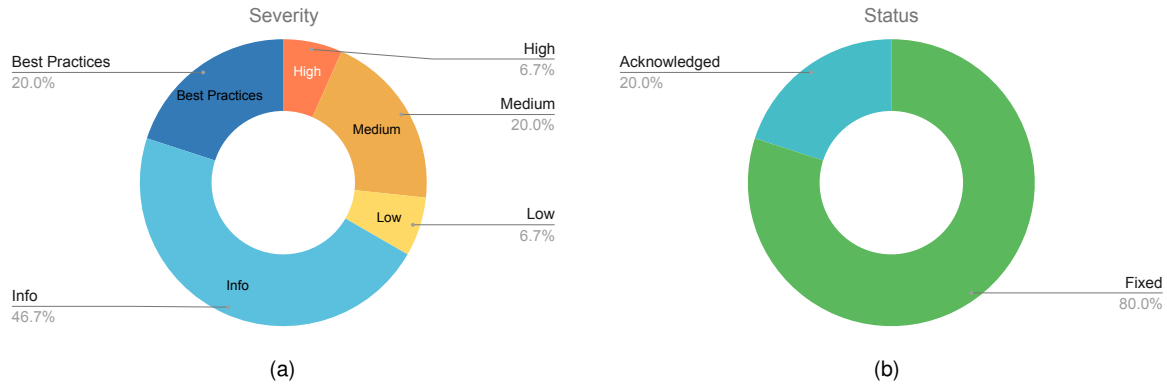

Fig. 1: Distribution of issues: Critical (0), **High** (1), **Medium** (3), **Low** (1), **Undetermined** (0), **Informational** (7), **Best Practices** (3). **Distribution of status: Fixed** (12), **Acknowledged** (3), **Mitigated** (0), **Unresolved** (0)

Summary of the Audit

Audit Type	Security Review
Initial Report	Aug 9, 2024
Response from Client	Regular responses during audit engagement
Final Report	Aug 9, 2024
Repository	btc-lrt
Commit (Audit)	92f90a91b08b215785e28c4de61bc6f575834146
Commit (Final)	0fda842c42fb9abc13d47c59a5dc7fdb40028ba3
Documentation Assessment	Medium
Test Suite Assessment	Low

2 Audited Files

	Contract	LoC	Comments	Ratio	Blank	Total
1	yearnVault/RoleManager.sol	351	349	99.4%	125	825
2	yearnVault/WithdrawLimitModule.sol	102	24	23.5%	23	149
3	yearnVault/DelayedWithdraw.sol	246	136	55.3%	86	468
4	tokenisedStrategy/AeraStrategy.sol	118	6	5.1%	30	154
	Total	**817**	**515**	**63.0%**	**264**	**1596**

3 Summary of Issues

	Finding	Severity	Update
1	Frontrunning of Symbiotic `claim(...)` function with `report()` can decrease strategy debt	High	Acknowledged
2	Remaining profits from shares should be transferred back to the `AeraStrategy`	Medium	Fixed
3	`AeraVaultV2` is not compatible with EigenLayer and Karak	Medium	Acknowledged
4	`assetsAtTimeRequest` does not consider previous withdrawal requests	Medium	Fixed
5	Protocol fees cannot be completely redeemed	Low	Fixed
6	Duplicated `_assessShareOfUnrealisedLosses()` function in the `WithdrawLimitModule` contract	Info	Fixed
7	Inconsistent check for the strategies length	Info	Fixed
8	Inconsistent multi-asset handling in `DelayedWithdraw` contract	Info	Fixed
9	Incorrect check inside `withdrawAeraVault(...)` function	Info	Fixed
10	Reset `AeraStrategy` allowance of `AeraVaultV2` when the ownership is transferred	Info	Fixed
11	The `value()` function of `AeraVaultV2` has to ensure that it returns total assets rather than their value	Info	Acknowledged
12	`setVaultAera(...)` does not update token allowance for `AeraVaultV2`	Info	Fixed
13	Missing Natspec for the `transferDustToFeeRecipient(...)` function	Best Practices	Fixed
14	Pending owner is not reset after accepting the ownership transfer	Best Practices	Fixed
15	Unnecessary `payable` keyword in `executeOnAera(...)` function	Best Practices	Fixed

4 System Overview

The `Swell` protocol introduces a new set of contracts designed to enable the restaking of WBTC across various protocols, including `Symbiotic` and `EigenLayer`. Users deposit WBTC into the `Yearn Vault V3`, which is managed by `Swell`. The protocol features a custom strategy called `AeraStrategy`, allowing the Yearn vault to allocate assets into `AeraVaultV2`. Guardians manage this vault, who then stake these assets into RST protocols.

The overall architecture of the protocol, including contracts and their main functionality, is illustrated in the following diagram:

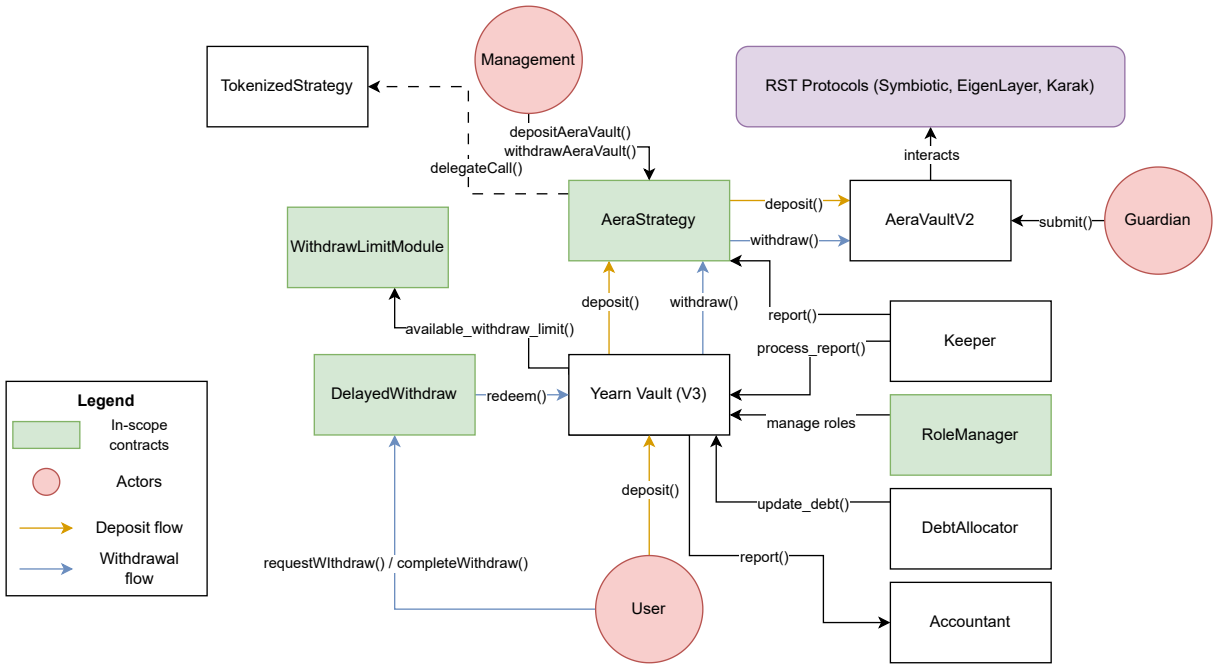

Fig. 2: Swell BTC LRT architecture

The main functionalities of the protocol are implemented within the following contracts:

- `VaultV3` - The Swell Vault contract manages user deposits and withdrawals. It is integrated with a single `AeraStrategy` contract that generates restaking rewards for the deposited assets.
- `AeraVaultV2` - This vault manages the assets of the `AeraStrategy` by depositing them into RST protocols.
- `AeraStrategy` - The main strategy connected to `VaultV3`. Assets deposited into this strategy are transferred to `AeraVaultV2`, where guardians manage the restaking process.
- `DelayedWithdraw` - This contract implements a delayed withdrawal mechanism for the `VaultV3` contract. Due to delays inherent in restaking protocols, users cannot withdraw directly from the vault and must use this contract to request and complete their withdrawals.
- `WithdrawLimitModule` - An additional module linked to the Swell vault determining the number of withdrawable assets. `Swell` uses this contract to ensure that withdrawals are processed exclusively through the `DelayedWithdraw` contract; for any other user, it returns zero assets.

Although the `AeraVaultV2` contract and `VaultV3` contract are crucial to the overall architecture, they were not directly in scope. The `Nethermind Security team` only considered possible integration issues with `AeraStrategy` and the rest of the system.

4.1 Deposit Flow and Restaking

Users deposit WBTC into the `VaultV3` contract and receive an equivalent amount of vault shares. Initially, these assets are marked as `idle`, meaning they are not generating rewards.

The Yearn Vault integrates with various protocols through a system of strategies designed for yield generation. Assets are deposited into the custom `AeraStrategy` using the `update_debt(...)` function, which then issues shares to the `VaultV3`.

```
def update_debt(strategy: address, target_debt: uint256, max_loss: uint256 = MAX_BPS) -> uint256
```

In this setup, `AeraStrategy` should have only one owner of the shares: `VaultV3`. `AeraStrategy` subsequently deposits these assets into `AeraVaultV2`, which serves as a treasury management contract. Guardians manage this contract and perform actions by calling the `submit(...)` function.

```
function submit(Operation[] calldata operations) external override nonReentrant onlyGuardian whenHooksSet
↪  whenNotFinalized whenNotPaused reserveFees checkReservedFees
```

Additionally, guardians are responsible for staking WBTC into RST protocols such as `Symbiotic`. Once the assets are restaked, the strategy begins to generate profits or losses based on the events in the RST protocol.

4.2 Withdrawal Flow

To initiate a withdrawal, a user must interact with the `DelayedWithdraw` contract by submitting a request via the `requestWithdraw(...)` function.

```
function requestWithdraw(ERC20 asset, uint96 shares, uint16 maxLoss, bool allowThirdPartyToComplete) external
↪  requiresAuth nonReentrant
```

This function transfers the specified `shares` from the user to the `DelayedWithdraw` contract. It then creates a withdrawal request and records the number of assets the user would receive based on the current share value. The request is subject to a maturation period known as `maturity`.

Once the withdrawal request reaches maturity, the user can complete the withdrawal using the `completeWithdraw(...)` function.

```
function completeWithdraw(ERC20 asset, address account) external requiresAuth nonReentrant returns (uint256 assetsOut)
```

This function recalculates the number of assets the user can receive. The function will revert if the share-to-asset ratio has changed significantly and exceeds the defined `maxLoss`. A portion of the shares is taken as a fee, and the remaining shares are redeemed for assets, which are then sent to the user. Users should note that they may receive fewer assets if the ratio changes between the request and completion time.

To ensure that withdrawal requests are covered, protocol admins are responsible for initiating withdrawals from RST protocols when `idle` assets in `VaultV3` are insufficient. The maturity period in the `DelayedWithdraw` contract is designed to align with similar delays in RST protocol withdrawals. Initially, guardians of `AeraVaultV2` initiate these withdrawals, transferring assets to `AeraVaultV2`. The `AeraStrategy` management then withdraws assets from `AeraVaultV2` to the strategy, making them available for redemption requests from `VaultV3`.

4.3 Reporting profits and losses

Reporting is a key part of the Yearn vault architecture. Anyone can use the permissionless `Keeper` contract to update the strategy's profits and losses and propagate the strategy's newest state to the vault.

`Keeper` allows anyone to call the `report()` function on the `AeraStrategy`. The strategy will update its state based on the losses or profits. It implements the internal function `_harvestAndReport()`:

```
function _harvestAndReport() internal view override returns (uint256 _totalAssets) {
_totalAssets = asset.balanceOf(address(this)) + IAeraVaultV2(vaultAera).value();
}
```

This function returns the total assets managed by the strategy, obtained as the current balance of the contract plus the result of the `value()` function. The `value()` function has to return the total number of assets managed by `AeraVaultV2`, including the balance of the vault, together with all assets staked in the RST protocols.

After the report is completed, the `Keeper` can call multi-signature wallets or smart wallets with multi-factor authentication is strongly recommended are vested over ten days to prevent flash-loan attacks; however, losses immediately affect all shareholders.

4.4 Centralization risks

Due to its overall architecture, the protocol carries some centralization risks. The primary risk arises if the protocol admin keys are compromised. To mitigate these risks, it is strongly recommended to utilize multi-signature wallets or smart wallets with multi-factor authentication. Below, we outline the key areas vulnerable to centralization risks:

- The `ADD_STRATEGY_MANAGER` role has the authority to add any strategy contract. This poses a risk as it could potentially direct assets to a malicious strategy.
- The `AeraStrategy` contract contains an `execute(...)` function that performs low-level calls to arbitrary targets with arbitrary data. This functionality could lead to unintended or harmful executions. Additionally, `AeraVaultV2` has a similar function, which, for example, allows the transfer of all assets to any arbitrary address.
- All assets managed by the `AeraStrategy` are handled by `AeraVaultV2`, which is managed by guardians. This introduces a centralization risk as guardians can transfer assets to any address using the `submit(...)` function.

5 Risk Rating Methodology

The risk rating methodology used by Nethermind Security follows the principles established by the OWASP Foundation. The severity of each finding is determined by two factors: **Likelihood** and **Impact**.

Likelihood measures how likely the finding is to be uncovered and exploited by an attacker. This factor will be one of the following values:

a) **High**: The issue is trivial to exploit and has no specific conditions that need to be met;

b) **Medium**: The issue is moderately complex and may have some conditions that need to be met;

c) **Low**: The issue is very complex and requires very specific conditions to be met.

When defining the likelihood of a finding, other factors are also considered. These can include but are not limited to motive, opportunity, exploit accessibility, ease of discovery, and ease of exploit.

Impact is a measure of the damage that may be caused if an attacker exploits the finding. This factor will be one of the following values:

a) **High**: The issue can cause significant damage, such as loss of funds or the protocol entering an unrecoverable state;

b) **Medium**: The issue can cause moderate damage, such as impacts that only affect a small group of users or only a particular part of the protocol;

c) **Low**: The issue can cause little to no damage, such as bugs that are easily recoverable or cause unexpected interactions that cause minor inconveniences.

When defining the impact of a finding, other factors are also considered. These can include but are not limited to Data/state integrity, loss of availability, financial loss, and reputation damage. After defining the likelihood and impact of an issue, the severity can be determined according to the table below.

		Severity Risk		
Impact	**High**	Medium	High	Critical
	Medium	Low	Medium	High
	Low	Info/Best Practices	Low	Medium
	Undetermined	Undetermined	Undetermined	Undetermined
		Low	**Medium**	**High**
		Likelihood		

To address issues that do not fit a High/Medium/Low severity, Nethermind Security also uses three more finding severities: **Informational**, **Best Practices**, and **Undetermined**.

a) **Informational** findings do not pose any risk to the application, but they carry some information that the audit team intends to pass to the client formally;

b) **Best Practice** findings are used when some piece of code does not conform with smart contract development best practices;

c) **Undetermined** findings are used when we cannot predict the impact or likelihood of the issue.

6 Issues

6.1 [High] Frontrunning of Symbiotic `claim(...)` function with `report()` can decrease strategy debt

File(s): `Vault.sol`

Description: The Symbiotic RST protocol allows users to re-stake their assets and obtain equivalent shares. Users can determine the number of assets deposited in Symbiotic's vault using the `balanceOf(...)` function:

```
function balanceOf(address account) external view returns (uint256) {
    uint256 epoch = currentEpoch();
    return activeBalanceOf(account) + withdrawalsOf(epoch, account) + withdrawalsOf(epoch + 1, account);
}
```

- `activeBalanceOf(account)` returns the number of deposited assets;
- `withdrawalsOf(epoch, account)` returns the number of assets available for withdrawal after the current epoch;
- `withdrawalsOf(epoch + 1, account)` returns the number of assets the user requested in the current epoch;

The epoch refers to specific periods within the Symbiotic ecosystem. The `balanceOf(...)` method, based on the current epoch, provides different results. This method also indicates assets that can be slashed by Symbiotic.

Withdrawals in Symbiotic occur in two steps:

1. The user initiates a withdrawal via `withdraw(...)` in epoch `X`;
2. In epoch `X+2`, the user can call `claim(uint epoch)`. This delay occurs because assets are added to `withdrawalsOf(epoch+1)`, but are claimable only when the next epoch finishes;

In epoch `X+2`, the `balanceOf(...)` function will exclude these claimable assets. An attacker can exploit this timing to cause a loss in `VaultV3`.

During epoch `X+2`, a guardian of `AeraVaultV2` must call the `claim(...)` function to transfer assets into the vault. However, an attacker could front-run this function, calling `report()` on the strategy and `process_report(...)` on `VaultV3` via `Keeper`, which allows permissionless reporting.

If this happens, `AeraStrategy` will update its assets to inform VaultV3 of its profit or loss status. Before the `claim(...)` function is executed in Symbiotic, the `AeraVaultV2` will not account for these claimable assets, causing `AeraStrategy` and `VaultV3` to register a loss. Shares will be burned, or outstanding profits will decrease. Once the `claim(...)` function is called and reporting is done again, the strategy will report a profit, causing the "withdrawable" assets to be vested as profit.

Recommendation(s): The `value()` function in `AeraVaultV2` should include all claimable assets from Symbiotic.

Status: Acknowledged

Update from the client: outside the current scope for a future audit.

6.2 [Medium] Remaining profits from shares should be transferred back to the `AeraStrategy`

File(s): `DelayedWithdraw.sol`

Description: When completing withdrawals at `DelayedWithdraw`, the user always receives fewer assets. However, its shares still accrue profits/losses when locked in a `DelayedWithdraw` contract and waiting for their maturity date when the user can withdraw.

If the shares are earning profits, these profits are left in the `DelayedWithdraw` contract and can be transferred to `feeAddress` by the `transferDustToFeeRecipient(...)` function. However, these remaining profits should be distributed to the shareholders, as users' shares should stop accruing rewards when they request withdrawal. The rest of the profits should go to the remaining shareholders.

Therefore, the remaining assets should be sent to AeraStrategy, as they cannot be deposited directly into `VaultV3`. In this case, the shareholders should get profits, not the protocol, to maintain a fair rewarding model.

Recommendation(s): Consider sending the remaining share profits to the users rather than the protocol.

Status: Fixed

Update from the client: Code Change - db278c8127fbc1855cfb29edad4c471df6fabb2e

6.3 [Medium] `AeraVaultV2` is not compatible with EigenLayer and Karak

File(s): `AeraVaultV2.sol`, `AeraStrategy.sol`

Description: The `AeraVaultV2` is designed to hold assets deposited into the `AeraStrategy`, which then restakes these assets into various restaking protocols. The `Swell` protocol has indicated that RST will include Symbiotic, Karak, and EigenLayer.

The Yearn vault `VaultV3` relies on knowing the total assets deposited into the `AeraStrategy` to determine if its strategy is profitable or incurring losses. `AeraStrategy` reports this information through the `report()` function, which invokes the internal `_harvestAndReport()` function. This function calculates the total assets held by the strategy as follows:

```
function _harvestAndReport() internal view override returns (uint256 _totalAssets) {
  _totalAssets = asset.balanceOf(address(this)) + IAeraVaultV2(vaultAera).value();
}
```

The `value()` function retrieves the value of balances deposited in the vault and the total assets of ERC4626 shares, which should be linked to various yield strategies. The issue arises because this approach only works if the RST protocol is ERC4626 compliant, which is not the case with EigenLayer. When the protocol stakes in EigenLayer, the `AeraStrategy` will not be able to report the assets present in EigenLayer as `AeraVaultV2` is not able to obtain this value, and the `value()` function is likely to revert.

Furthermore, Karak and EigenLayer implement a two-step withdrawal model, which first removes/transfers shares from the user, and the user can redeem his shares after a specific period. The `AeraVaultV2` should also account for these queued shares, which is not currently possible.

Recommendation(s):

- Use only ERC4626-compliant RST protocols;
- Implement custom logic to handle non-ERC4626 compliant protocols like EigenLayer;
- Implement a mechanism that allows the tracking of outstanding withdrawals;

Status: Acknowledged

Update from the client: Outside of the current scope, it's for a future Audit.

6.4 [Medium] `assetsAtTimeRequest` does not consider previous withdrawal requests

File(s): `DelayedWithdraw.sol`

Description: The `DelayedWithdraw` contract enforces a two-step withdrawal process for VaultV3 shareholders. Shareholders initiate a withdrawal by calling `requestWithdraw(...)`, and after a specified delay period, they can call `completeWithdraw` to receive their assets.

When a user requests a withdrawal, various values are stored, including `req.assetsAtTimeOfRequest`, which represents the number of assets at the time of the request. However, this value does not consider all shares of the request, leading to inconsistencies. Users can make multiple withdrawal requests, adding up the total shares to be redeemed from the vault during `completeWithdraw`.

The `req.assetsAtTimeOfRequest` is calculated as follows:

```
function requestWithdraw(...) external requiresAuth nonReentrant {
// ...
  req.shares += shares;
  uint40 maturity = uint40(block.timestamp + withdrawAsset.withdrawDelay);
  req.maturity = maturity;
  // @audit not considering all shares
  req.assetsAtTimeOfRequest = lrtVault.previewRedeem(shares);
  req.maxLoss = maxLoss;
  req.allowThirdPartyToComplete = allowThirdPartyToComplete;
// ...
}
```

Due to this calculation, `completeWithdraw` is likely to revert because of the `maxLoss` check, as there will be a significant difference between `minAssetToWithdraw` and `maxAssetToWithdraw`:

```
uint256 minAssetToWithdraw = req.assetsAtTimeOfRequest < currentAssetToWithdraw
      ? req.assetsAtTimeOfRequest
      : currentAssetToWithdraw;
uint256 maxAssetToWithdraw = req.assetsAtTimeOfRequest < currentAssetToWithdraw
      ? currentAssetToWithdraw
      : req.assetsAtTimeOfRequest;
```

In the case of multiple withdrawal requests, `minAssetToWithdraw` will always consider `req.assetsAtTimeOfRequest`, while `maxAssetToWithdraw` will store `currentAssetToWithdraw`, leading to a significant discrepancy and causing a revert.

Recommendation(s): Update `req.assetsAtTimeOfRequest` to account for all shares in the withdrawal request.

Status: Fixed

Update from the client: Code Change - 4bf46743042a2bca0663550410b328d9926cdc36

6.5 [Low] Protocol fees cannot be completely redeemed

File(s): `DelayedWithdraw.sol`

Description: After a user completes a withdrawal, a specified `withdrawFee` percentage is deducted as a protocol fee, collected in vault shares, as illustrated below:

```
function _completeWithdraw(...) internal returns (uint256 assetsOut) {
  // ...
  if (withdrawAsset.withdrawFee > 0) {
        // Handle withdraw fee.
        uint256 fee = uint256(shares).mulDivDown(withdrawAsset.withdrawFee, 1e4);
        shares -= fee;

        // Transfer fee to feeAddress.
        IERC20Metadata(lrtVault).safeTransfer(feeAddress, fee);
      }
  // ...
}
```

However, an issue arises because the `feeAddress` also needs to withdraw its shares through the same `_completeWithdraw(...)` function, resulting in additional fees being applied.

Recommendation(s): To mitigate this issue, prevent the vault from taking fees from the already claimed protocol fees. This can be achieved by either claiming the fees as assets or implementing a separate mechanism for the `feeAddress` to redeem its shares without incurring additional fees.

Status: Fixed

Update from the client: Code Change - b730ce08c7f4d5a10b07896cdbface6b96f50cda We also changed the way max loss is computed: The new approach computes the accepted amount from `maxAmount`, not from `minAmount`, cause `1/(1+maxLoss) != 1-maxLoss`.

6.6 [Info] Duplicated `_assessShareOfUnrealisedLosses()` function in the `WithdrawLimitModule` contract

File(s): `WithdrawLimitModule.sol`

Description: The `WithdrawLimitModule` contract defines the internal `_assessShareOfUnrealisedLosses(...)` function to assess the shares of unrealised losses. However, the logic of this function is identical to the Swell vault's `assess_share_of_unrealised_losses(...)` external function. Resulting in redundant logic definition.

Recommendation(s): Reuse the existing `assess_share_of_unrealised_losses(...)` function from Swell vault.

Status: Fixed

Update from the client:Code Change - 425b04dfafdf0f259d1f94907f2ec23564fb93a4

6.7 [Info] Inconsistent check for the strategies length

File(s): `WithdrawLimitModule.sol`

Description: The `available_withdraw_limit(...)` function determines the maximum withdrawal amount during the redemption process of Swell's vault. It accepts a list of strategies and enforces that the list contains either 0 or 1 strategy addresses.

```
function available_withdraw_limit(
    address _owner,
    uint256 maxLoss,
    address[] calldata strategies //@audit parameter provided during the call to `redeem`
) external view returns (uint256) {
    if (_owner != withdrawManager) return 0;
    uint256 lenStrateg = strategies.length;
    //@audit `strategies.length` should always be 0
    require(lenStrateg <= 1, ">1Strat");
    // ...
}
```

The `strategies` array is provided by the caller of the `redeem(...)` function, which is invoked within the `_completeWithdraw()` function of the `DelayedWithdraw` contract, as shown below:

```
lrtVault.redeem(shares, address(this), address(this), maxLoss);
```

Since the `strategies` parameter is not explicitly passed, it defaults to an empty array. Therefore, the `available_withdraw_limit()` function should not consider a length of 1 as valid input.

Recommendation(s): Modify the check in the `available_withdraw_limit(...)` function to account for the fact that the `strategies` array will always be empty, by only accepting an array length of 0.

Status: Fixed

Update from the client: Code Change - ea78c0e5f83716da318d4544a9a83696a2a1723e

6.8 [Info] Inconsistent multi-asset handling in `DelayedWithdraw` contract

File(s): `DelayedWithdraw.sol`

Description: The `DelayedWithdraw` contract currently supports multiple assets through the `withdrawAssets` mapping, which stores the withdrawal settings for each asset.

```
/**
 * @notice Mapping from assets to their respective withdrawal settings.
 */
mapping(ERC20 => WithdrawAsset) public withdrawAssets;
```

Despite this multi-asset functionality, the contract interacts with a single `lrtVault` for asset redemptions. This mismatch can lead to potential configuration issues. Specifically, in functions such as `_cancelWithdraw(...)` and `_completeWithdraw(...)`, the `asset` parameter might differ from the underlying asset of the `lrtVault`.

Recommendation(s): Address this inconsistency by either updating the contract to use a mapping for `lrtVault` to support multiple assets or by implementing checks to ensure that the `asset` parameter aligns with the underlying asset of the `lrtVault`.

Status: Fixed

Update from the client: Code Change : 0b121f353c3efdfc7d57b5f4843c43898b957e04

6.9 [Info] Incorrect check inside `withdrawAeraVault(...)` function

File(s): AeraStrategy.sol

Description: The `withdrawAeraVault(...)` function currently checks if the `AeraVaultV2` contains enough assets to be withdrawn using the `value()` function.

```
function withdrawAeraVault(uint256 amount) external onlyQuickManagement {
  require(IAeraVaultV2(vaultAera).value() >= amount, "AmountTooHigh");
  _freeFunds(amount);
}
```

However, this function returns the total number of all assets, including those in RST protocols. Instead, it should verify if the current balance of `AeraVault2` can fulfil the requested withdrawal amount.

Recommendation(s): Adjust the check to use the balance of the vault rather than the total value.

Status: Fixed

Update from the client: Code Change - 239928e971a2df03b675efb28bd026cd1cffca38

6.10 [Info] Reset `AeraStrategy` allowance of `AeraVaultV2` when the ownership is transferred

File(s): AeraStrategy.sol

Description: The `AeraStrategy` implements `transferOwnership(...)` which allows to transfer ownership of `AeraVaultV2`. This function does not reset the token allowance to `AeraVaultV2` to zero.

This should be implemented based on AeraVaults' Spearbit audit report. Currently, any malicious guardian can transfer all tokens in `AeraStrategy` to any arbitrary address through the `submit(...)` function.

Recommendation(s): Reset the token allowance to zero.

Status: Fixed

Update from the client: Code Change - f3787099ce3053c5806b75179525a78b246265cf

6.11 [Info] The `value()` function of `AeraVaultV2` has to ensure that it returns total assets rather than their value

File(s): AeraStrategy.sol

Description: The `AeraStrategy` reports its assets using the `_harvestAndReport()` function, which calculates total assets with the following formula:

```
_totalAssets = asset.balanceOf(address(this)) + IAeraVaultV2(vaultAera).value();
```

The problem could arise from the `value()` function in `AeraVaultV2`, which returns the value of the assets rather than their actual number. This can cause an inconsistency when combined with `asset.balanceOf(address(this))`, as the value is not equivalent to the balance. This difference can lead to inaccurate reporting of the total assets held by the strategy.

While the `value()` implementation can be customized to return the number of assets by having a custom oracle that will return a 1:1 rate, the protocol has to ensure that these oracles are used.

Recommendation(s): Consider customizing the used oracle to return the total number of assets and not their value.

Status: Acknowledged

Update from the client: Outside the current scope, it's for a future audit

6.12 [Info] `setVaultAera(...)` does not update token allowance for `AeraVaultV2`

File(s): `AeraStrategy.sol`

Description: The `AeraStrategy` contract includes a function, `setVaultAera(...)`, which allows updating the `AeraVaultV2` address. However, this function does not increase the token allowance for the new vault, which can prevent subsequent deposits. Furthermore, the allowance for the old vault is not reduced.

```
function setVaultAera(address _vaultAera) external onlyManagement {
  require(IAeraVaultV2(vaultAera).value() == 0, "!Empty");
  //@audit  Allowance is not updated for the new vault
  vaultAera = _vaultAera;
  emit UpdateVaultAera(_vaultAera);
}
```

While the contract provides an `execute(...)` function for low-level contract interactions, this is not an ideal solution for managing routine operations like updating allowances.

Recommendation(s): When setting a new `AeraVaultV2`, update the token allowance accordingly. Additionally, consider removing the allowance for the previous vault.

Status: Fixed

Update from the client: Code Change - 41ce0081e6b5ac2103e11810f1d570ce6bd640ee

6.13 [Best Practices] Missing Natspec for the `transferDustToFeeRecipient(...)` function

File(s): `DelayedWithdraw.sol`

Description: The `transferDustToFeeRecipient(...)` function is designed to transfer the contract's balance of a specified `asset` to the protocol's `feeAddress`. Although this function uses the `requiresAuth` modifier, it lacks documentation indicating its intended visibility. This omission could lead to misconfiguration of the Authentication module.

```
// @audit missing comments specifying the function's visibility
function transferDustToFeeRecipient(ERC20 asset) public requiresAuth {
  if (address(asset) == address(lrtVault)) revert DelayedWithdraw__transferNotAllowed();

  uint256 balance = asset.balanceOf(address(this));
  if (balance > 0) {
    asset.safeTransfer(feeAddress, balance);
  }
}
```

Recommendation(s): Add NatSpec documentation to the function to clearly specify its intended visibility.

Status: Fixed

Update from the client: Code Change - db278c8127fbc1855cfb29edad4c471df6fabb2e

6.14 [Best Practices] Pending owner is not reset after accepting the ownership transfer

File(s): `WithdrawLimitModule.sol`

Description: The `WithdrawLimitModule` contract implements a 2-steps ownership transfer mechanism. First, the current owner calls the `transferOwnership(...)` function, providing the address of the new owner, storing it in the `pendingOwner` variable. Subsequently, the pending owner must call the `acceptOwnershipt()` function, which will update the owner of the contract to the pending owner.

However, the `pendingOwner` variable is not reset upon setting the `owner`. Consequently, the same address remains designated as both the current and pending owner, allowing the current owner to initiate unnecessary ownership transfers repeatedly.

```
function acceptOwnershipt() external {
  address _pendingOwner = pendingOwner;
  require(msg.sender == pendingOwner, "!PendingOwner");

  owner = _pendingOwner;
  emit OwnerSet(_pendingOwner);
}
}
```

Recommendation(s): Consider resetting the `pendingOwner` to `address(0)` when the ownership transfer is successfully done.

Status: Fixed

Update from the client: Code Change - c1b81184932df7718a386f49b6ee5446c31a6e0b

6.15 [Best Practices] Unnecessary `payable` keyword in `executeOnAera(...)` function

File(s): `AeraStrategy.sol`

Description: The `AeraStrategy` contract's `executeOnAera(...)` function is marked as payable but does not utilize any Ether provided in `msg.value`, which should be zero.

```
  function executeOnAera(Operation calldata operation) external payable onlyManagement {
    IAeraVaultV2(vaultAera).execute(operation);
  }
```

Recommendation(s): Consider removing payable from the `executeOnAera(...)` function.

Status: Fixed

Update from the client: Code Change - 329b35d8dce6e71db196ca7365da60128a43401a

7 Documentation Evaluation

Software documentation refers to the written or visual information that describes the functionality, architecture, design, and implementation of software. It provides a comprehensive overview of the software system and helps users, developers, and stakeholders understand how the software works, how to use it, and how to maintain it. Software documentation can take different forms, such as user manuals, system manuals, technical specifications, requirements documents, design documents, and code comments. Software documentation is critical in software development, enabling effective communication between developers, testers, users, and other stakeholders. It helps to ensure that everyone involved in the development process has a shared understanding of the software system and its functionality. Moreover, software documentation can improve software maintenance by providing a clear and complete understanding of the software system, making it easier for developers to maintain, modify, and update the software over time. Smart contracts can use various types of software documentation. Some of the most common types include:

- Technical whitepaper: A technical whitepaper is a comprehensive document describing the smart contract's design and technical details. It includes information about the purpose of the contract, its architecture, its components, and how they interact with each other;
- User manual: A user manual is a document that provides information about how to use the smart contract. It includes step-by-step instructions on how to perform various tasks and explains the different features and functionalities of the contract;
- Code documentation: Code documentation is a document that provides details about the code of the smart contract. It includes information about the functions, variables, and classes used in the code, as well as explanations of how they work;
- API documentation: API documentation is a document that provides information about the API (Application Programming Interface) of the smart contract. It includes details about the methods, parameters, and responses that can be used to interact with the contract;
- Testing documentation: Testing documentation is a document that provides information about how the smart contract was tested. It includes details about the test cases that were used, the results of the tests, and any issues that were identified during testing;
- Audit documentation: Audit documentation includes reports, notes, and other materials related to the security audit of the smart contract. This type of documentation is critical in ensuring that the smart contract is secure and free from vulnerabilities.

These types of documentation are essential for smart contract development and maintenance. They help ensure that the contract is properly designed, implemented, and tested, and they provide a reference for developers who need to modify or maintain the contract in the future.

Remarks about the Swell documentation

The documentation for the Swell protocol was provided through notion page with high-level diagrams. Moreover, the Swell team addressed all questions and concerns raised by the Nethermind Security team, providing valuable insights and a comprehensive understanding of the project's technical aspects.

8 Test Suite Evaluation

8.1 Compilation Output

```
> forge build
[] Compiling...
[] Compiling 1 files with Solc 0.8.25
[] Solc 0.8.25 finished in 4.61s
Compiler run successful!
```

8.2 Tests Output

```
> forge test

Ran 7 tests for test/VaultV3.t.sol:VaultV3Test
[PASS] testDeposit() (gas: 159663)
[PASS] testIfAccountantIsSet() (gas: 15310)
[PASS] testRedeemRevert() (gas: 173511)
[PASS] testReportGains() (gas: 1069084)
[PASS] testUpdateDebt() (gas: 405391)
[PASS] testWithdraw() (gas: 306580)
[PASS] testWithdrawRevert() (gas: 176664)
Suite result: ok. 7 passed; 0 failed; 0 skipped; finished in 7.04s (11.04ms CPU time)
```

Remarks about Swell test suite

The Swell team has developed a testing suite that covers the protocol's basic flows, including deposits, withdrawals, and simulating the strategy's profits. However, the test suite lacks various integration tests together with an exploration of different scenarios which could influence whole protocol.

9 About Nethermind

Nethermind is a Blockchain Research and Software Engineering company. Our work touches every part of the web3 ecosystem - from layer 1 and layer 2 engineering, cryptography research, and security to application-layer protocol development. We offer strategic support to our institutional and enterprise partners across the blockchain, digital assets, and DeFi sectors, guiding them through all stages of the research and development process, from initial concepts to successful implementation.

We offer security audits of projects built on EVM-compatible chains and Starknet. We are active builders of the Starknet ecosystem, delivering a node implementation, a block explorer, a Solidity-to-Cairo transpiler, and formal verification tooling. Nethermind also provides strategic support to our institutional and enterprise partners in blockchain, digital assets, and decentralized finance (DeFi). In the next paragraphs, we introduce the company in more detail.

Blockchain Security: At Nethermind, we believe security is vital to the health and longevity of the entire Web3 ecosystem. We provide security services related to Smart Contract Audits, Formal Verification, and Real-Time Monitoring. Our Security Team comprises blockchain security experts in each field, often collaborating to produce comprehensive and robust security solutions. The team has a strong academic background, can apply state-of-the-art techniques, and is experienced in analyzing cutting-edge Solidity and Cairo smart contracts, such as ArgentX and StarkGate (the bridge connecting Ethereum and StarkNet). Most team members hold a Ph.D. degree and actively participate in the research community, accounting for 240+ articles published and 1,450+ citations in Google Scholar. The security team adopts customer-oriented and interactive processes where clients are involved in all stages of the work.

Blockchain Core Development: Our core engineering team, consisting of over 20 developers, maintains, improves, and upgrades our flagship product - the Nethermind Ethereum Execution Client. The client has been successfully operating for several years, supporting both the Ethereum Mainnet and its testnets, and now accounts for nearly a quarter of all synced Mainnet nodes. Our unwavering commitment to Ethereum's growth and stability extends to sidechains and layer 2 solutions. Notably, we were the sole execution layer client to facilitate Gnosis Chain's Merge, transitioning from Aura to Proof of Stake (PoS), and we are actively developing a full-node client to bolster Starknet's decentralization efforts. Our core team equips partners with tools for seamless node set-up, using generated docker-compose scripts tailored to their chosen execution client and preferred configurations for various network types.

DevOps and Infrastructure Management: Our infrastructure team ensures our partners' systems operate securely, reliably, and efficiently. We provide infrastructure design, deployment, monitoring, maintenance, and troubleshooting support, allowing you to focus on your core business operations. Boasting extensive expertise in Blockchain as a Service, private blockchain implementations, and node management, our infrastructure and DevOps engineers are proficient with major cloud solution providers and can host applications in-house or on clients' premises. Our global in-house SRE teams offer 24/7 monitoring and alerts for both infrastructure and application levels. We manage over 5,000 public and private validators and maintain nodes on major public blockchains such as Polygon, Gnosis, Solana, Cosmos, Near, Avalanche, Polkadot, Aptos, and StarkWare L2. Sedge is an open-source tool developed by our infrastructure experts, designed to simplify the complex process of setting up a proof-of-stake (PoS) network or chain validator. Sedge generates docker-compose scripts for the entire validator set-up based on the chosen client, making the process easier and quicker while following best practices to avoid downtime and being slashed.

Cryptography Research: At Nethermind, our Cryptography Research team is dedicated to continuous internal research while fostering close collaboration with external partners. The team has expertise across a wide range of domains, including cryptography protocols, consensus design, decentralized identity, verifiable credentials, Sybil resistance, oracles, and credentials, distributed validator technology (DVT), and Zero-knowledge proofs. This diverse skill set, combined with strong collaboration between our engineering teams, enables us to deliver cutting-edge solutions to our partners and clients.

Smart Contract Development & DeFi Research: Our smart contract development and DeFi research team comprises 40+ world-class engineers who collaborate closely with partners to identify needs and work on value-adding projects. The team specializes in Solidity and Cairo development, architecture design, and DeFi solutions, including DEXs, AMMs, structured products, derivatives, and money market protocols, as well as ERC20, 721, and 1155 token design. Our research and data analytics focuses on three key areas: technical due diligence, market research, and DeFi research. Utilizing a data-driven approach, we offer in-depth insights and outlooks on various industry themes.

Our suite of L2 tooling: Warp is Starknet's approach to EVM compatibility. It allows developers to take their Solidity smart contracts and transpile them to Cairo, Starknet's smart contract language. In the short time since its inception, the project has accomplished many achievements, including successfully transpiling Uniswap v3 onto Starknet using Warp.

- **Voyager** is a user-friendly Starknet block explorer that offers comprehensive insights into the Starknet network. With its intuitive interface and powerful features, Voyager allows users to easily search for and examine transactions, addresses, and contract details. As an essential tool for navigating the Starknet ecosystem, Voyager is the go-to solution for users seeking in-depth information and analysis;
- **Horus** is an open-source formal verification tool for StarkNet smart contracts. It simplifies the process of formally verifying Starknet smart contracts, allowing developers to express various assertions about the behavior of their code using a simple assertion language;
- **Juno** is a full-node client implementation for Starknet, drawing on the expertise gained from developing the Nethermind Client. Written in Golang and open-sourced from the outset, Juno verifies the validity of the data received from Starknet by comparing it to proofs retrieved from Ethereum, thus maintaining the integrity and security of the entire ecosystem.

Learn more about us at nethermind.io.

General Advisory to Clients

As auditors, we recommend that any changes or updates made to the audited codebase undergo a re-audit or security review to address potential vulnerabilities or risks introduced by the modifications. By conducting a re-audit or security review of the modified codebase, you can significantly enhance the overall security of your system and reduce the likelihood of exploitation. However, we do not possess the authority or right to impose obligations or restrictions on our clients regarding codebase updates, modifications, or subsequent audits. Accordingly, the decision to seek a re-audit or security review lies solely with you.

Disclaimer

This report is based on the scope of materials and documentation provided by you to Nethermind in order that Nethermind could conduct the security review outlined in **1. Executive Summary** and **2. Audited Files**. The results set out in this report may not be complete nor inclusive of all vulnerabilities. Nethermind has provided the review and this report on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your sole risk. Blockchain technology remains under development and is subject to unknown risks and flaws. The review does not extend to the compiler layer, or any other areas beyond the programming language, or other programming aspects that could present security risks. This report does not indicate the endorsement of any particular project or team, nor guarantee its security. No third party should rely on this report in any way, including for the purpose of making any decisions to buy or sell a product, service or any other asset. To the fullest extent permitted by law, Nethermind disclaims any liability in connection with this report, its content, and any related services and products and your use thereof, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. Nethermind does not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by a third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or accessible through the report, its content, and the related services and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising, and Nethermind will not be a party to or in any way be responsible for monitoring any transaction between you and any third-party providers of products or services. As with the purchase or use of a product or service through any medium or in any environment, you should use your best judgment and exercise caution where appropriate. FOR AVOIDANCE OF DOUBT, THE REPORT, ITS CONTENT, ACCESS, AND/OR USAGE THEREOF, INCLUDING ANY ASSOCIATED SERVICES OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, INVESTMENT, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.