PUBLIC

Code Assessment

of the swBTC Smart Contracts

August 9, 2024

Produced for

by

Contents

1 Executive Summary

Dear Swell team,

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

Swell implements a BTC LRT aiming to be compatible with any restaking protocol to allocate to the best AVS's across multiple platforms. The system is built off YearnV3's robust Vault Codebase.

The most critical subjects covered in our audit are the functional correctness of the delayed withdrawal module and the correct integration of Yearn's VaultV3 and tokenized strategy systems. Security regarding all the aforementioned subjects is extensive.

Other general subjects covered are access control and interactions with the Aera vault. Security regarding all the aforementioned subjects is high.

The test coverage is minimal and should be improved to ensure that all code paths and features are tested.

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	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	2
Low-Severity Findings	5
• Code Corrected	5

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 swBTC 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	18 July 2024	92f90a91b08b215785e28c4de61bc6f575834146	Initial Version
2	30 July 2024	7efa6269de87eb07e1b9f93acb3264781ecef567	Second Version
3	8 August 2024	239928e971a2df03b675efb28bd026cd1cffca38	Third Version
4	8 August 2024	0fda842c42fb9abc13d47c59a5dc7fdb40028ba3	Fourth Version

For the Solidity smart contracts, the compiler version `0.8.25` was chosen, for the Vyper smart contracts, the compiler version `0.3.10` was chosen. Both compilers were used to compile to EVM version `Shanghai`.

The following files were in scope for the assessment:

- `src/tokenisedStrategy/AeraStrategy.sol`
- `src/yearnVault/DelayedWithdraw.sol`
- `src/yearnVault/WithdrawLimitModule.sol`

The following files were also in scope, as the code was forked from the Yearn codebase, the assessment was limited to a review of the changes made compared to the Yearn system if any, together with a general review regarding their overall integration within the system:

Tokenized strategy:

- `src/tokenisedStrategy/BaseStrategy.sol`
- `src/tokenisedStrategy/TokenizedStrategy.sol`

Tokenized strategy periphery (added in Version 2):

- `src/tokenisedStrategy/BaseHooks.sol`
- `src/tokenisedStrategy/Hooks.sol`

Vault periphery:

- `src/yearnVault/Accountant.sol`
- `src/yearnVault/DebtAllocator.sol`
- `src/yearnVault/DebtAllocatorFactory.sol`
- `src/yearnVault/Keeper.sol`
- `src/yearnVault/Registry.sol`
- `src/yearnVault/RegistryFactory.sol`
- `src/yearnVault/ReleaseRegistry.sol`
- `src/yearnVault/RoleManager.sol`

Yearn Vaults v3:

- `src/yearnVault/VaultFactory.vy`
- `src/yearnVault/VaultV3.vy`
- `src/yearnVault/interfaces`

The following yearn commits were used for assessing the changes made to the yearn codebase:

- Tokenized strategy: cf791a6f2d360e5c33866c9f0de10e83085920e9
- Tokenized strategy periphery: 6ce8d29b1e107a89754dd9f17337582734989b4d
- Vault periphery: e6489628373925a722b65e7d36e1d177738d4b37
- Yearn Vaults v3: 9fbc614bbce9d7cbad42e284a15f0f43cf1a673f

2.1.1 Excluded from scope

Anything not mentioned above is considered out of scope. This includes and is not limited to `solmate`, `openzeppelin` libraries, tests and mock contracts.

2.2 System Overview

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

At the end of this report section, we have added subsections for each of the changes according to the versions.

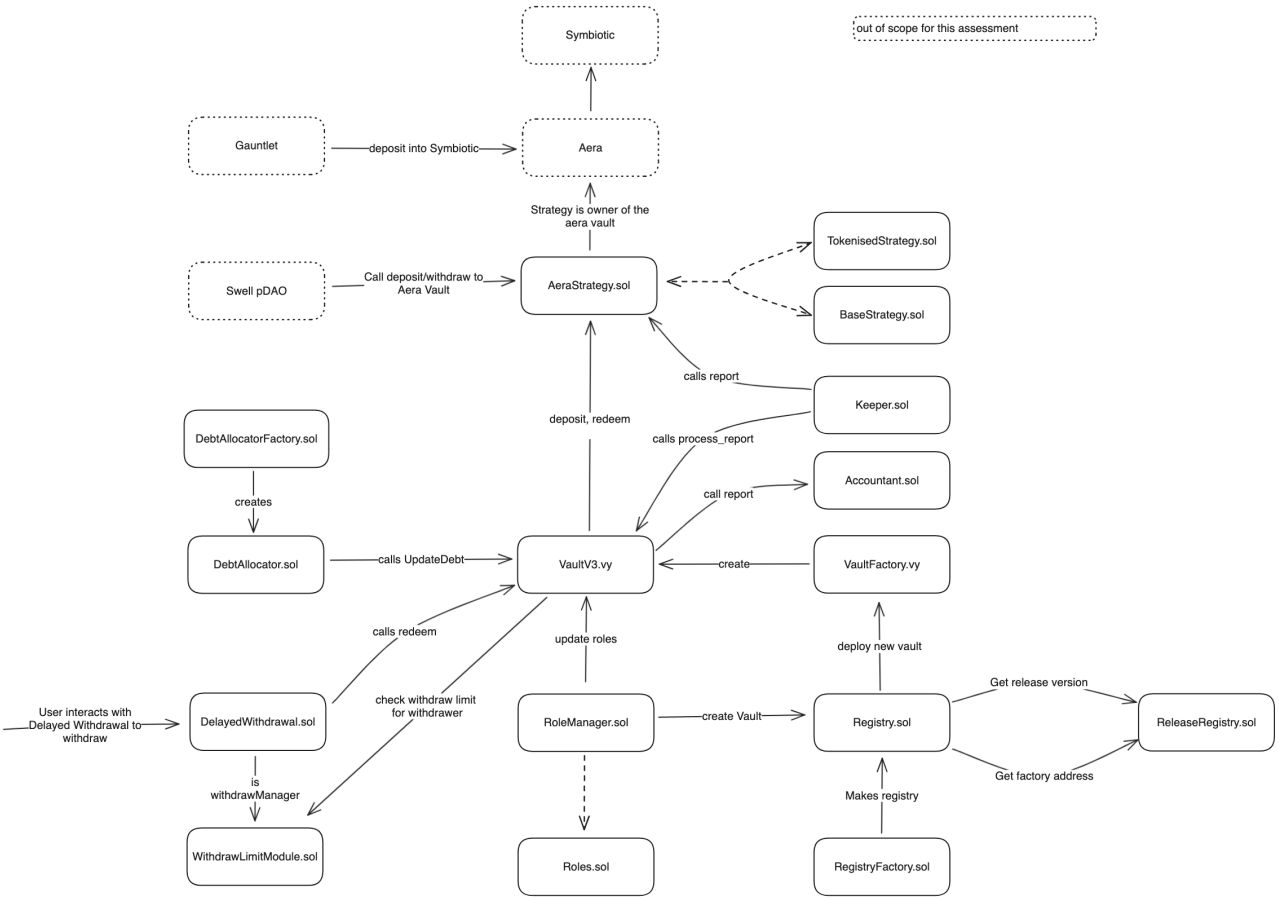
Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

Swell offers a BTC LRT aiming to be compatible with any restaking protocol.

The system can be divided into two components:

1. An Allocator Vault, where users can deposit wBTC and receive a corresponding share of the (ERC4626) vault. The vault is built off the Yearn vaultV3 codebase and inherits its logic and roles.
2. A tokenized strategy, which deposits the supplied wBTC into an Aera vault. The Aera vault is out of scope for this review, but it is assumed that it would then effectively allocate to the best AVS across multiple restaking protocols.

The following diagram shows the system components and their interactions:



2.2.1 The Allocator Vault

The following details the allocator vault and its components. As most of the contracts are identical or extremely similar to the Yearn codebase, only the `DelayedWithdrawal` and the `WithdrawLimitModule` are described in detail.

2.2.1.1 Registry Factory

The Registry Factory is responsible for creating new Registry contracts. The function `createNewRegistry` is permissionless and will create a new registry. If an address is provided, it will be used as governance otherwise the message sender is used. The `releaseRegistry` address holds the implementation of the release registry to be used by the created registries.

2.2.1.2 Release Registry

The Release Registry is a permissioned contract that holds the address of the factory for each version of the vault. New releases can be made by providing the new factory address and given a release version, the corresponding factory address can be retrieved.

2.2.1.3 Registry

The registry serves as an on-chain registry to track Swell vaults and strategies. The Registry holds the addresses of deployed vaults/strategies, together with the assets they manage and several other metadata. It can be used to:

- Deploy and endorse a new vault using `newEndorsedVault`. The governance or an `endorser` can deploy a vault for some asset directly from the registry. The version to be used can be specified to fetch the correct factory address.
- Endorse an existing vault. The governance or an `endorser` can endorse a vault that was deployed outside the registry.
- Tag a vault with some custom string. This feature is available to the governance or a `tagger`.

2.2.1.4 Vault Factory

The Vault Factory is responsible for creating new vaults. It holds an immutable variable `VAULT_ORIGINAL` which is the address of the implementation of the version of the vault the factory is supposed to create. The function `deploy_new_vault` is unpermissioned and can be used to deploy a new vault with the given asset, name, symbol, role manager and profit max unlock time. The provided role manager is used to later set the roles of the vaults.

Additionally, the factory's governance can also:

- Change the fee recipient address used by all vaults created by the factory.
- Change the protocol fee of a specific vault.
- Shut down the factory, preventing the creation of new vaults.

2.2.1.5 Role Manager

The Role Manager can manage the roles of one or multiple vaults. The contracts define several positions, and each position can hold a subset of the Vaults roles. The initial configuration set in the constructor is as follows:

- `Daddy`: All roles.
- `Brain`: `REPORTING_MANAGER`, `DEBT_MANAGER`, `QUEUE_MANAGER`, `DEPOSIT_LIMIT_MANAGER`, `DEBT_PURCHASER`.
- `Security`: `MAX_DEBT_MANAGER`.
- `Keeper`: `REPORTING_MANAGER`.
- `Strategy Manager`: `ADD_STRATEGY_MANAGER`, `REVOKE_STRATEGY_MANAGER`.
- `Debt Allocator`: `REPORTING_MANAGER`, `DEBT_MANAGER`.

Additionally, the role manager can:

- Start managing an existing vault whose `role_manager` is set to the Role Manager's address. This can be done by the `Daddy` by calling `addNewVault`. If the Vault is not yet endorsed by the registry known to the Role Manager, it will endorse it first.
- Deploy a new vault, endorse it to the registry and start managing it. This can be done by the `Daddy` by calling `newVault`.
- Remove a vault, can be done by the `Daddy` by calling `removeVault`. In this case, the `chad` (initially set to `Daddy`) will be set as the new role manager of the given vault.
- Update the `DebtAllocator` of a vault. This can be done by the `Brain` by calling `updateDebtAllocator`.
- Update the Keepers of a vault. This can be done by the `Brain` by calling `updateKeeper`.

2.2.1.6 VaultV3

The following is an adapted description of the vault V3 from the Yearn documentation.

The Yearn VaultV3 is designed to distribute funds of depositors for a specific `asset` into different strategies and robustly manage accounting. In the case of Swell's allocator vault, the asset used is `wBTC` and the only known strategy is the `AeraStrategy`, a tokenized strategy that deposits the funds into an Aera vault so they can be used to restake across multiple protocols.

`wBTC` depositors receive shares proportional to their deposit amount. Vault tokens are yield-bearing and can be redeemed at any time to get back the deposit plus any yield generated. Given that the strategy involves restaking, redeeming is not instant. Users can only withdraw their funds using the `DelayedWithdraw`, which will delay the withdrawal for a certain amount of time.

Different permissioned roles can be attributed using the `role_manager`. They manage the vault together and can allocate funds as they best see fit to different strategies and adjust the strategies and allocations as needed, as well as reporting realized profits or losses. In the case of Swell, an important role is the `WITHDRAW_LIMIT_MANAGER`, except to be attributed to the `WithdrawLimitModule`, and essential to the delayed withdrawal mechanism.

Strategies are any ERC-4626 compliant contracts that use the same underlying `asset` as the vault. The vault provides no assurances as to the safety of any strategy and it is the responsibility of those who hold the corresponding roles to choose and fund strategies that best fit their desired specifications.

Those holding vault tokens can redeem the tokens for the corresponding amount of underlying assets based on any reported profits or losses since their initial deposit.

The vault is built to be customized by the management to be able to fit their specific desired needs. Including the customization of strategies, accountants, ownership etc.

2.2.1.7 Debt Allocator Factory

The Debt Allocator Factory is responsible for creating new `DebtAllocator` contracts. The Debt Allocator implementation is provided at deployment time, and upon calling the unpermissioned function `newDebtAllocator`, a new Debt Allocator is created with the given implementation using the EIP-1167 minimal proxy pattern. The factory's keepers can update the `baseFeeProvided` and `maxAcceptableBaseFee`, read by the deployed Debt Allocators using `isCurrentBaseFeeAcceptable()`. Additionally, the governance promotes or demotes keepers using `setKeeper()`.

2.2.1.8 Debt Allocator

This Debt Allocator is meant to be used alongside a Yearn V3 vault to provide the needed triggers for a keeper to perform automated debt updates for the vault's strategies. The allocator should be attributed the `DEBT_MANAGER` role in the vault. The governance and managers can add or remove strategies to be managed using `setStrategyDebtRatio()`, and `removeStrategy()`. Keepers can call `update_debt()` to let the vault re-balance the debt vs target debt for some strategy. This will effectively compare the current debt with the target debt and will take funds or deposit new funds into the strategy.

The `keeper` and `governance` roles are inherited from the Debt Allocator Factory, while managers can be added or removed by the governance using `setManager()`.

2.2.1.9 Keeper

The keeper is a contract that can be used to allow for permissionless reporting on V3 vaults and strategies. It is expected that the contract is given the keeper role in some strategies and the allocator vault as it exposes 3 unpermissioned functions:

- `report()` which calls the given strategy `report()` function.
- `tend()` which calls the given strategy `tend()` function.
- `process_report()` which calls the given vault `process_report()` function.

2.2.1.10 Accountant

The accountant is a contract that can be used to allow for permissionless reporting on V3 vaults. The Vault calls the accountant as part of `VaultV3._process_report()`, the accountant is in charge of handling fees, refunds and running health checks on the reported profits and losses.

The privileged roles of the contract accountant are the `feeManager`, `feeRecipient` and `vaultManager`.

2.2.1.11 Delayed Withdraw

The `DelayedWithdraw` contract allows users to withdraw their funds from the vault after a certain delay.

Users can perform the following actions:

- Request a withdrawal by calling `requestWithdraw()`. This will create a new withdrawal request with an asset and the given amount of shares to be redeemed. The user also provides `maxLoss` which is the maximum loss the user is willing to accept, and `allowThirdPartyToComplete` which allows third parties to complete the withdrawal. The contract will transfer the shares to itself and increment the user's request for the given asset, overwriting the previous maturity time and max loss. Note that the shares are not redeemed yet.
- Cancel a withdrawal request by calling `cancelWithdraw()`. This will send back the shares to the user and remove the user's request for the given asset.
- Complete a withdrawal by calling `completeWithdraw()`. If the given withdrawal request is matured, the completion window has not passed, and the loss is not greater than `maxLoss`, the contract will redeem the share for the user and transfer the asset to the user.
- Allow third parties to complete withdrawal for a given asset by calling `setAllowThirdPartyToComplete()`. This means that anyone can call the `completeWithdraw()` function on behalf of the user after the withdrawal matured.

The following functionalities are available to privileged users (the `owner` and privileged users defined by the `authority`):

- Pause or unpause the contract by calling `pause()` or `unpause()`. If the contract is paused, users can not open new withdraw requests and withdraw requests can not be completed.
- Transfer ownership to some new address with `transferOwnership()`.
- Set a new authority using `setAuthority()` (There is no authority set by default).
- Stopping withdrawals for a given asset by calling `stopWithdrawalsInAsset()`.
- Adding new assets to the list of assets that can be withdrawn by calling `setupWithdrawAsset()`.
- Changing the withdrawal delay for a given asset by calling `changeWithdrawDelay()`.
- Changing the completion window for a given asset by calling `changeCompletionWindow()`.
- Changing the withdraw fee for a given asset by calling `changeWithdrawFee()`.
- Changing the max loss for a given asset by calling `changeMaxLoss()`.
- Updating the fee address using `setFeeAddress()`.
- Cancelling any user withdrawal with `cancelUserWithdraw()`.
- Completing a user withdrawal even if it is outside the completion window with `completeUserWithdraw()`.

2.2.1.12 Withdraw Limit Module

The Withdraw Limit Module is a contract used to prevent anyone except for the `DelayedWithdraw` from withdrawing funds from the allocator vault. This effectively enforces the delayed withdrawal mechanism as otherwise users could withdraw their funds instantly by calling the vault directly. The contract's principal function is `available_withdraw_limit()`, which is used by the vault to check if a withdrawal request can be fulfilled. The function enforces that the `_owner` of the shares being burned is the `withdrawManager` which should be the `DelayedWithdraw`'s address. The rest of the function's logic essentially replicates the vault's `_max_withdraw()` logic, with the exception that at most one strategy can be redeemed from to fulfill the withdrawal request.

The contract's owner can change the `withdrawManager` address through `setWithdrawManager` and also transfer ownership of the contract through `transferOwnership`.

2.2.2 The Aera Strategy

Swell implements a strategy to deposit funds into an Aera Vault. The strategy inherits from the `BaseStrategy` from Yearn's tokenized strategy template and is hence an ERC-4626 compliant contract. For a detailed description of the global strategy structure, refer to the Yearn Tokenized Strategy Documentation.

As all stateful user-facing functions are already implemented in the `BaseStrategy` and the `TokenizedStrategy`, the `AeraStrategy` simply overrides the following functions:

- `availableWithdrawLimit()` is called before any withdrawal or redemption to enforce any limits desired, it returns the IDLE assets that are sitting in the strategy contract. This means that the strategy does not allow for withdrawals to withdraw funds from the Aera vault.
- `availableDepositLimit()` is called before any deposit or mints to enforce any limits desired. The function returns 0 except if the `owner` wanting to deposit is the allocator vault, in which case it returns the maximum value of `uint256`. This effectively restricts the strategy to only accept deposits from the allocator vault.
- `_deployFunds()` is called at the end of a `deposit` or `mint` and call the `vaultAera.deposit()` function to deposit the funds into the Aera vault.
- `_freeFunds()` is called at the end of a withdrawal or redemption and calls the `vaultAera.withdraw()` function to deposit the funds into the Aera vault.
- `_harvestAndReport()` returns the sum of both idle assets and the Aera vault's value. The function does not perform any redeployment of funds.

The privileged roles of the strategy are:

- The `management` role, which can execute arbitrary calls on behalf of the strategy with `execute()`, manages the Aera Vault including (un)pausing it, manages the strategy including managing privileged roles, changing the performance fee and its recipient as well as shutting down the strategy or performing emergency withdrawal.
- The `quickManagement` role, which can call both `_deployFunds()` and `_freeFunds()`. The latter is important as `_freeFunds()` can not be called by strategy users given the `availableWithdrawLimit()` restriction.
- The keepers that can call `report()` to harvest and record all realized profits and losses.

2.2.3 Trust Model

Privileged roles of the system have powerful capabilities, it should be noted that they can perform actions that could be harmful to the system, such as withdrawing all funds from the system or locking the system in many ways. It is assumed that all precautions are taken to ensure that these roles are not compromised and act honestly.

The Vault underlying token should be ERC-20 compliant without weird behaviors such as double entry points, rebase mechanisms, transfer fees, irrational return values, high decimals, unusually large supply, etc

- The following roles in the system are fully trusted:
    - The `DelayedWithdraw`'s `owner`.
    - The `WithdrawLimitModule`'s `owner`.
    - The `VaultFactory`, `RoleManager`, `RegistryFactory`, `Registry`, `ReleaseRegistry`, `DebtAllocatorFactory`'s `governance`.
    - All `VaultV3` roles defined in `Roles`.
    - The `keepers` of the `DebtAllocatorFactory`.

- The managers of the DebtAllocator.
- The vaultManager and feeManager of the Accountant.
- The AeraStrategy's management, emergencyAdmin, quickManagement and keepers.

- The DelayedWithdraw's authority is assumed to be set to a trusted contract that implements access control and prevents regular users from calling any of the DelayedWithdraw's functions except for the one annotated as public: setAllowThirdPartyToComplete(), requestWithdraw(), cancelWithdraw() and completeWithdraw(), which should be callable by anyone. Privileged parties allowed by the authority to call the other functions are assumed to be trusted.
- The vaultAera set in the AeraStrategy is trusted to manage the funds sent to it effectively.
- BaseStrategy's tokenizedStrategyAddress is assumed to be the address of the deployed implementation of TokenizedStrategy on the same chain the BaseStrategy is to be deployed.
- In WithdrawLimitModule, withdrawManager is assumed to be set to the address of the DelayedWithdraw.
- The AeraVaultV2 owned by the AeraStrategy is assumed to have wBTC set as the numeraire asset.

2.2.4 Changes in versions

All changes made in Version 2 are implementations of fixes for the issues raised in this report. The most significant change is the addition of the BaseHooks to the AearaStrategy to ensure that only the allocator vault can deposit into the strategy. The BaseHooks allows for various hooks to be overridden by the strategist to enforce custom logic upon deposit, withdrawal, or transfer. In the case of the AeraStrategy, only the _preDepositHook is overridden to ensure that the caller of deposit() and mint() is the allocator vault.

Version 2 introduces setForceVaultAera() in AeraStrategy and safeLockedShares() in DelayedWithdraw, two new functions that should only be called by some trusted privileged roles.

- setForceVaultAera() allows for the management to change the VaultAera address even in the case the old vault was not empty and should hence only be used carefully in emergencies.
- safeLockedShares() allows the DelayedWithdraw to transfer locked shares out of the contract. This function can in theory also be used to transfer shares of ongoing users' withdrawals, hence it should be used with caution. This highlights that the DelayedWithdraw's owner and authority should be fully trusted by users.

Version 3 fixes further issues and slightly improves the codebase with the following changes:

- Only the correct asset can be set up as a withdraw asset in the DelayedWithdraw contract. This is enforced in the contract even though it still has the structure supporting multiple ones.
- The loss computation done in the DelayedWithdraw._completeWithdraw() was slightly improved to match better the intended behavior.
- If the feeAddress is the one calling the DelayedWithdraw._completeWithdraw() function, the fee is voided.
- The function DelayedWithdraw.transferDustToFeeRecipient() was replaced by DelayedWithdraw.transferDustToStrategy() sending the dust to the strategy instead of the fee recipient.
- The function DelayedWithdraw.safeLockedShares() now verifies that it does not transfer shares of ongoing users' withdrawals.

- `WithdrawLimitModule._assessShareOfUnrealisedLosses()` was removed to simplify the codebase.
- The function `WithdrawLimitModule.available_withdraw_limit()` is now using the `default_queue` and ignore the `calldata`.
- `pendingOwner` is reset to the zero address after the ownership transfer in `DelayedWithdraw.acceptOwnership()`.
- `AeraStrategy.execute()` now checks that the `msg.value` is the same as the `value` parameter and `executeOnAera` was made `payable`.
- `AeraStrategy.withdrawAeraVault()` verifies the balance of the vault before withdrawing.
- `AeraStrategy` resets its allowance to the `AeraVaultV2` when transferring its ownership.

Version 4 reverts the change regarding `AeraStrategy.execute()` made in **Version 3** given it allowed for locking ether into the `AeraStrategy`.

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 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:

- 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 Findings section.

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	2

- Incorrect Withdrawal Amount When Opening Multiple Requests Code Corrected
- Withdrawals Can Be DOS by a Malicious User Code Corrected

Low-Severity Findings	5

- AeraStrategy May Have Locked Funds Code Corrected
- Allowances Not Updated When Changing Vault Code Corrected
- Locked Shares Code Corrected
- Setting a Wrong Address as the vaultAera Can DOS Code Corrected
- Vyper Version Code Corrected

Informational Findings	3

- Duplicated Code Code Corrected
- Non-indexed Events Code Corrected
- Typo in Function Name Code Corrected

6.1 Incorrect Withdrawal Amount When Opening Multiple Requests

Correctness Medium Version 1 Code Corrected

CS-SWELL-SWBTC-001

In `DelayedWithdraw`, `requestWithdraw()` allows users to request a new withdrawal request even if they have an existing pending withdrawal request. In such cases

- The request's `shares` are incremented with the new shares to redeem
- The old `maxLoss`, `maturity` date and `allowThirdPartyToComplete` are respectively overwritten with the given `maxLoss`, the newly computed maturity date, and the given `allowThirdPartyToComplete`.

However, the previous `assetsAtTimeOfRequest` is also overwritten with `lrtVault.previewRedeem(shares)`. This means that in the case a user calls multiple times `requestWithdraw()`, `assetsAtTimeOfRequest` will be overwritten with the shares provided in the last call.

- If the user has a small `maxLoss`, this will most likely prevent them from completing the withdrawal request due to a loss check performed between `assetsAtTimeOfRequest` and `currentAssetToWithdraw`.

- If the user has a large `maxLoss`, the amount of asset withdrawn by the user might be significantly smaller than it should be given that the user will only be given `assetsAtTimeOfRequest`, which only corresponds to the shares of the last withdrawal request.

Code corrected

`assetsAtTimeOfRequest` is now computed using the whole request shares to account for the share sent in each call to `requestWithdraw()`.

6.2 Withdrawals Can Be DOS by a Malicious User

Design | Medium | Version 1 | Code Corrected

CS-SWELL-SWBTC-002

When a deposit is triggered for a strategy, the entire balance is deployed through `deployFunds` in `TokenizedStrategy._deposit()` which results in emptying all the idle funds from the strategy.

Since the available balance to withdraw from the strategy - defined in `AreaStrategy.availableWithdrawLimit()` - returns the contract's balance no funds are left and this will return 0.

The consequence is that if there are no more or not enough idle funds in the vault V3, a user will not be able to withdraw his funds if the strategy balance is zero. This state of a zero balance can be maintained by a malicious user who can keep depositing, making sure that the strategy balance is always zero. This can be done by calling `TokenizedStrategy.deposit()` with the allocating Vault as the `receiver`. During this time, no user will be able to withdraw some funds. Even if the management decides to free funds from the strategy, the malicious actor can backrun the management's transaction and deposit again to empty the strategy balance.

Note that in this hypothetical scenario, the malicious actor is not able to steal funds, and instead, keeps making donations to the system to keep the strategy balance at zero.

Code corrected

The Yearn tokenized strategy periphery `BaseHooks` contract is now used to allow for the `AeraStrategy` to override the `_preDepositHook` and ensure that when calling `deposit` or `mint`, the message sender must be the receiver. Combined with `availableDepositLimit()`, this ensures that only the allocator vault can deposit into the strategy and no one else can do it on behalf of the allocator vault.

6.3 `AeraStrategy` May Have Locked Funds

Design | Low | Version 3 | Code Corrected

CS-SWELL-SWBTC-018

`AeraStrategy.execute` requires that the `msg.value` is the same as the value parameter being passed to the low-level call. This means that if this low-level call reverts, the funds will be locked in the contract.

Code corrected

The check has been removed from the function.

6.4 Allowances Not Updated When Changing Vault

Design | Low | Version 1 | Code Corrected

CS-SWELL-SWBTC-003

In `AeraStrategy`, `setVaultAera()` can be used to update the Aera vault to use. However, when doing so:

1. The allowance that was given to the previous vault is not removed.
2. The new vault is not given an allowance, and an extra action from the management will be needed to do so using `execute()`.

Code corrected

When changing the vault, the allowance given to the previous one is now set to 0 and the new one's to `type(uint256).max`.

6.5 Locked Shares

Design | Low | Version 1 | Code Corrected

CS-SWELL-SWBTC-004

The function `transferDustToFeeRecipient()` does not allow for the owner to transfer vault shares out of the contract given that it could be shares belonging to a user whose request has been requested but not yet completed or canceled. This means that in the case a user transfers shares of the allocator vault to the `DelayedWithdraw` contract by mistake, the shares would be locked in the contract indefinitely.

Code corrected

A function `safeLockedShares()` was added to the `DelayedWithdraw` contract, it allows a privileged party to transfer the locked shares out of the contract. The function should be called with care as no protection is in place to prevent the shares of users in the process of withdrawing from being transferred out.

6.6 Setting a Wrong Address as the `vaultAera` Can DOS

Correctness | Low | Version 1 | Code Corrected

CS-SWELL-SWBTC-005

In the `AeraStrategy`, if `vaultAera` is set to an incorrect address by `setVaultAera()`, it can lead to DOS. This cannot be recovered by this same setter given the check that `IAeraVaultV2(vaultAera).value() == 0`, which will always revert if the address does not point to a contract or if the pointed contract does not return at least 32 null bytes when called with the function selector `value()`.

Code corrected

A function `setForceVaultAera()` was added to the `AeraStrategy`, it allows to set the `vaultAera` without the check for the value and is only callable by the management.

6.7 Vyper Version

Design | Low | Version 1 | Code Corrected

CS-SWELL-SWBTC-006

For the Vyper contracts, version 0.3.8 has been chosen, however, the version introduces a regression leading the initcode size to be much larger than it should be as dead code is not correctly pruned. This regression was fixed shortly after in version 0.3.9 with PR-3450. Although this issue does not affect the behavior of the contract, the fix introduces some significant gas savings.

Code corrected

The Vyper contracts have been updated to version 0.3.10.

6.8 Duplicated Code

Informational | Version 1 | Code Corrected

CS-SWELL-SWBTC-007

`WithdrawLimitModule._assessShareOfUnrealisedLosses()` and `VaultV3.assess_share_of_unrealised_losses()` are redundant since they are performing the same computation, code duplication is error-prone.

Acknowledged

`WithdrawLimitModule._assessShareOfUnrealisedLosses()` has been removed from the code base.

6.9 Non-indexed Events

Informational | Version 1 | Code Corrected

CS-SWELL-SWBTC-014

No parameters are indexed in the events defined in `WithdrawLimitModule`. It is recommended to index the relevant event parameters to allow integrators and dApps to quickly search for these and simplify UIs.

Code corrected

The relevant parameters in the events have been indexed.

6.10 Typo in Function Name

Informational | Version 1 | Code Corrected

CS-SWELL-SWBTC-016

The function `WithdrawLimitModule.acceptOwnershipt` suffers from a typo in the function name.

Code corrected

The function name has been corrected to `acceptOwnership`.

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 Events Can Be Emitted With No Change

Informational **Version 1** **Acknowledged**

CS-SWELL-SWBTC-008

Several events are emitted even if nothing has changed, here are some examples:

1. `DelayedWithdraw.pause` emits `Paused` even if the contract is already paused.
2. `DelayedWithdraw.unpause` emits `Unpaused` even if the contract is already unpaused.
3. `DelayedWithdraw.changeWithdrawDelay` emits `WithdrawDelayUpdated` even if the delay is not updated.
4. `DelayedWithdraw.changeCompletionWindow` emits `CompletionWindowUpdated` even if the window is not updated.
5. `DelayedWithdraw.changeWithdrawFee` emits `WithdrawFeeUpdated` even if the fee is not updated.
6. `DelayedWithdraw.changeMaxLoss` emits `MaxLossUpdated` even if the loss is not updated.
7. `DelayedWithdraw.setFeeAddress` emits `FeeAddressSet` even if the address is already set to the new address.
8. `DelayedWithdraw.setAllowThirdPartyToComplete` emits `ThirdPartyCompletionChanged` even if the value is not changed.
9. `AeraStrategy.setPendingQuickManagement` emits `UpdatePendingQuickManagement` even if the address is already set to the new address.
10. `AeraStrategy.setVaultAera` emits `UpdateVaultAera` even if the address is already set to the new address.
11. `AeraStrategy.setPendingQuickManagement` emits `UpdatePendingQuickManagement` even if the address is already set to the new address.
12. `WithdrawLimitModule.setWithdrawManager` emits `WithdrawManagerSet` even if the address is already set to the new address.
13. `WithdrawLimitModule.transferOwnership` emits `PendingOwnerSet` even if the address is already set to the new address.

Acknowledged

Swell acknowledged the informational issue.

7.2 Gas Savings

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

CS-SWELL-SWBTC-009

The following gas savings were found:

1. `AeraStrategy.vaultYearn` could be immutable.
2. `WithdrawLimitModule.acceptOwnershipt()` is loading `pendingOwner` from storage twice.
3. `DelayedWithdraw` handles all the logic to support multiple assets whereas only one asset is needed.

In Version 2, the following additional gas saving was found:

4. `DelayedWithdraw.requestWithdraw()` loads twice `req.shares` from storage. The second load is unnecessary as the value could be computed from the first load.

In Version 3, the following additional gas saving was found:

5. `DelayedWithdraw.safeLockedShares()` takes as argument `asset`, but it can only be `lrtVault`, otherwise the function will revert. The argument could be removed.

Code partially corrected

Only items 1 and 2 have been addressed.

7.3 Hardhat Config Specifies `london` Hardfork

Informational | Version 1 | Acknowledged

CS-SWELL-SWBTC-010

The `foundry.toml` sets the EVM version to `shanghai`, however, `hardhat.config.js` specifies the `london` hardfork. This may lead to confusion and should be aligned.

Acknowledged

Swell acknowledged the informational issue

7.4 Misleading Comments and NatSpecs

Informational | Version 1 | Code Partially Corrected

CS-SWELL-SWBTC-011

The following comments and NatSpecs are incorrect or do not describe correctly what the code is doing:

1. In `RoleManager._newVault()` the `_symbol` is described to be `swell{SYMBOL}-LRT-{CATEGORY}`, which is not what the code is doing.
2. In `DelayedWithdraw`, the functions `pause` and `unpause` have outdated comments from forked contract.
3. In `DelayedWithdraw._completeWithdraw()`, the following comment is misleading:

```
// Make sure minRate * maxLoss is greater than or equal to maxRate.
```

Code partially corrected

Only items 1 and 2 have been addressed.

7.5 Missing Constant

Informational | Version 1 | Acknowledged

CS-SWELL-SWBTC-012

`1e4` is used in multiple places in the codebase, it should be defined as a constant. This will also improve readability and maintainability.

Acknowledged

Swell acknowledged the informational issue

7.6 Missing or Incomplete Events

Informational | Version 1 | Code Partially Corrected

CS-SWELL-SWBTC-013

Several events are missing or incomplete:

1. `DelayedWithdraw.constructor()` does not emit an event for the `feeAddress` being set.
2. `DelayedWithdraw.setupWithdrawAsset()` does not emit any information about the `completionWindow`.
3. `DelayedWithdraw.requestWithdraw()` does not emit any information about `allowThirdPartyToComplete`
4. `DelayedWithdraw._completeWithdraw()` could emit who completed the withdrawal.
5. `AeraStrategy.constructor()` does not emit an event for the `vaultAera` being set neither for the `quickManagement`.
6. `WithdrawLimitModule.constructor()` does not emit an event after setting the `owner` and `withdrawManager`.

Code partially corrected

Items 1, 2, 3, 5, and 6 have been corrected.

Item 4 is not corrected as:

- `completeUserWithdraw()` still emits the same `WithdrawCompleted` event as before, lacking information about who completed the withdrawal (a privileged party in this case).
- `completeWithdraw()` emits one `WithdrawCompleted` and one `WithdrawalCompleted` event, however, the latter logs the account from which the withdrawal was completed, and not the party that completed the withdrawal:
    - `WithdrawCompleted(address indexed account, ERC20 indexed asset, uint256 shares, uint256 assets)`
    - `WithrawalCompleted(address indexed account)`

7.7 Openzeppelin Library Does Not Use the Latest Version

Informational | Version 1 | Acknowledged

CS-SWELL-SWBTC-015

The Openzeppelin library used in the codebase is not the latest version. It is important to note that even if no major issue was discovered in the version used, it is always recommended to use the latest version of the library to benefit from the latest features and security updates as emphasized in the Openzeppelin documentation.

Acknowledged

Swell acknowledged the informational issue.

7.8 `solmate` Is an Experimental Library

Informational | Version 1 | Code Partially Corrected

CS-SWELL-SWBTC-017

The codebase uses the `solmate` contracts and library. It is important to note that the code is still experimental, as the README file in the project highlights:

```
This is **experimental software** and is provided on an "as is" and "as available" basis.

We **do not give any warranties** and **will not be liable for any loss** incurred through any use of this codebase.
```

Code partially corrected

In `DelayedWithdraw`, the `ReentrancyGuard` is now used from the OpenZeppelin library. However, the `solmate` library is still used in the codebase.

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 Completing a Withdrawal Can Fail Even Due to an Increase in the Price of the Shares

Note Version 1

In `DelayedWithdraw`, `completeWithdraw()` reverts in case the amount of assets to be obtained by the users is smaller than the originally quoted amount of assets at the time of the request. This threshold is set by `maxLoss`.

However, this check can also prevent a user from completing their withdrawal if the price of the shares has increased since the time of the request. In such a case, although the user would get the same amount of assets they were initially quoted (minus the fees), the check would prevent them from completing the withdrawal.

8.2 Privileged Roles Capabilities

Note Version 1

Privileged roles of the system have powerful capabilities, it should be noted that they can perform actions that could be harmful to the system, such as withdrawing all funds from the system or locking the system in many ways. It is assumed that all precautions are taken to ensure that these roles are not compromised and act honestly.

8.3 User Always Get the Worst Price Between Request and Completion Prices

Note Version 1

In `DelayedWithdraw`, the amount of asset to be sent to the user is the lowest price between the amount at the time of the request and the one at the time of the completion, which means the user price is always the lowest possible price.

8.4 Withdrawals Can Be Impossible if There Are Not Enough Idle Assets

Note Version 1

In the `AeraStrategy`, `availableWithdrawLimit()` returns `asset.balanceOf(address(this))`, the funds idle in the strategy. This means that `TokenizedStrategy.redeem()` and `TokenizedStrategy.withdraw()` will never withdraw from the `AeraVault`. Hence, without action from the quick management or management, it might be that both the allocator vaults idle assets and the strategy idle assets are not enough to cover a withdrawal.