PUBLIC

# Code Assessment

## of the Symbiotic-Aera Adapter Smart Contracts

September 6, 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 Symbiotic-Aera Adapter according to Scope to support you in forming an opinion on their security risks.

Swell implements Symbiotic-Aera Adapter, an adapter smart contract that allows the Aera treasury management protocol to deposit funds in the Symbiotic restaking protocol. Swell intends to use this within the scope of the swBTC project, to allocate *WBTC* to Symbiotic through a managed Aera vault.

The most critical subjects covered in our audit are frontrunning, effect of withdrawal delays, and jumps in value because of slashings. Security regarding all the aforementioned subjects is high.

The general subjects covered are accounting of assets, and correct integration. Security regarding all the aforementioned subjects is high.

In summary, we find that the codebase provides a high 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 | 0 |
| Low-Severity Findings | 1 |
| • Code Corrected | 1 |

# 2 Assessment Overview

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

## 2.1 Scope

The assessment was performed on the source code files inside the Symbiotic-Aera Adapter 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 | 26 August 2024 | 0f35ead494e5549a9a6290811484f1971b67ac2b | Initial Version |
| 2 | 6 September 2024 | f833241654d9f98d6d42c4b7c58f3134852473dc | Second Version |

For the solidity smart contracts, the compiler version `0.8.25` was chosen.

### 2.1.1 Included in scope

This report covers Swell Symbiotic-Aera Adapter, the following file is in scope:

- src/adapters/SymbioticAdapter.sol

The correctness of integrations with the Aera, Swell, and Symbiotic protocols has been evaluated according to their specification, which was inferred from documentation and code.

### 2.1.2 Excluded from scope

External systems and libraries that Symbiotic-Aera Adapter interacts with are assumed to behave correctly according to the specification.

## 2.2 System Overview

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

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

Swell offers SymbioticAdapter, a smart contract that allows the Aera vault where Swell funds are allocated to deposit and interact with the Symbiotic restaking protocol.

Swell implements a BTC liquid restaking token, that allows user to deposit WBTC (Wrapped Bitcoin), and receive swBTC tokens whose value accrues through the allocation of the underlying WBTC in restaking protocols for the purpose of securing third party networks. Depositors of WBTC into restaking protocols earn the network rewards, and are liable to slashing in case operators misbehave. Swell manages the allocation of user funds into several restaking vaults to maximize returns.

Users interact with the Swell protocol by depositing their funds (WBTC) into a tokenized vault, and they become owners of shares of the vault. They can redeem their shares to get the initial amount back plus a gain or loss, depending on the performance of the underlying restaking vaults. User funds are therefore pooled together, and managed by Swell. Swell deposits user funds in an Aera vault. Aera is a "wealth

management" protocol, which implements managed vault, where whitelisted operators have capabilities to manage the funds according to configurable strategies. One of these strategies consists in taking funds from the Aera vault, and depositing them in a vault of the Symbiotic restaking protocol.

## 2.2.1 Symbiotic Vaults

The purpose of the *SymbioticAdapter* is to deposit staking tokens from an Aera vault to a Symbiotic vault, and viceversa withdraw them. Symbiotic vault allows stakers to deposit tokens, which are used as economic guarantee to secure arbitrary distributed networks. The staked tokens can be slashed, in case the network operators misbehave. Slashing means that some of the balance of the Symbiotic vault is sent to a *Burner* contract to be destroyed. In this way, depositors of Symbiotic can suffer some losses on their position. The networks however also reward the token stakers. The rewards can be claimed in a separate smart contract, the Symbiotic *stakerRewards* contract, where the depositor of Symbiotic can claim to receive tokens to a specified address. In particular, rewards to not accrue automatically in the Symbiotic vault, meaning that the Symbiotic vault share value can only decrease, as consequence of slashings.

### 2.2.1.1 Deposits and Withdrawals

Deposits to the Symbiotic vault are requested through the `deposit()` function. The amount of shares minted is returned as a value to the user. These shares are however non-transferable. Withdrawals from the Symbiotic vault are more complex, as they require the user to wait a time delay. The essential purpose of the time delay on withdrawal of restaking protocols is to ensure that slashings, which might be processed with a delay with respect to the offending action that triggered them, cannot be evaded by withdrawing before the slashing take effects. For this purpose Symbiotic uses the concept of *epochs*. Epochs are consecutive time windows of equal length, which start at the initialization time of the vault. During epoch `T`, a user wishing to deallocate their funds can request a withdrawal through function `withdraw()` of the Symbiotic *Vault*. The withdrawal is added to the set of withdrawals of epoch `T + 1`, and becomes claimable during epoch `T + 2`. If a user schedules multiple withdraws for the same epoch, the withdrawals accumulate together as a single one. While withdrawals are scheduled but not claimable, the amounts requested are still slashable. This means that when the user finally claims the withdraw after epoch `T + 1` with function `claim()`, the amount received could be less than the `withdraw()` amount because of slashings. The `claim()` function takes as argument the epoch at which the withdrawal was scheduled. The claiming user needs therefore to keep track of the epochs in which their withdrawals have been scheduled. The time required for a withdrawal to become claimable is between 1 epoch duration (if the withdrawal is requested at the end of an epoch), and 2 epoch durations (if the withdrawal is requested right at the start of an epoch).

### 2.2.1.2 User balances in Symbiotic

Symbiotic *Vaults* expose the functions `balanceOf()`, `activeBalanceOf()`, and `withdrawalsOf()` to query different concepts of *balances* of users. `activeBalanceOf()` returns the amount currently staked by a user, that is the amount that has been deposited by a user, which has not been yet requested for withdrawal, minus slashing penalties. `withdrawalsOf()` takes as arguments the user address and a given epoch index, and returns the amount requested for withdrawal by that user for that epoch (reminder that at epoch `T`, users can request withdrawals indexed with epoch `T + 1`, which become claimable at `T + 2`). `balanceOf()` returns the *slashable* balance of a Symbiotic staker, which means the current active balance (stake), plus the unclaimable withdrawals, that is withdrawals with indexes `T` and `T + 1`. Claimable withdrawals, which are not affected by new slashings, can be queried manually by knowing their epoch index, through function `withdrawalsOf()`. To calculate the full balance of a user within a Symbiotic vault, the active balance (`activeBalanceOf()`) must be added to all the unclaimed withdrawals (claimable and unclaimable). Symbiotic does not expose a method to perform this calculation. The main purpose of the Swell Symbiotic - Aera adapter is to allow the calculation of this value, for the purpose of correct accounting of the value under management in the Aera vault.

### 2.2.2 Swell shares pricing

For the purpose of minting shares to depositing users of Swell, the Swell vault (Yearn v3 fork) has to estimate the total value of the assets it controls. The Aera vault is queried by the Swell vault regarding the value it is currently managing. The Aera vault calculates the value as the sum over whitelisted *assets* of `asset.balanceOf(vault) * asset.price` (not verbatim). In the case of a Symbiotic vault, as we mentioned, `balanceOf()` returns only the *slashable* balance, while the claimable withdrawals are not accounted. The purpose of *SymbioticAdapter* is to wrap around the Symbiotic *Vault* so that claimable rewards are also accounted for.

### 2.2.3 SymbioticAdapter

Swell introduces *SymbioticAdapter* as an adapter through which an Aera vault can interact with a Symbiotic vault. An instance of *SymbioticAdapter* is controlled by a single Aera vault, which can call its methods `deposit()`, `withdraw()`, and `claim()`. The main purpose of the adapter is to implement the method `balanceOf()`, such that it accounts for the whole value that the Aera vault holds in the Symbiotic vault, and not only its active stake. For this purpose, *SymbioticAdapter* keeps track of all currently open withdrawal requests, in an array, and computes its own balance as the sum of the active balance it holds in the Symbiotic vault (`activeBalanceOf()`), and the sum of the amounts it holds in unclaimed withdrawal requests in the Symbiotic vaults (`withdrawalsOf()`).

The `deposit()` function of *SymbioticAdapter* simply forwards a deposit from Aera to Symbiotic. The function `withdraw()` does some extra work, which consists in recording the withdrawal epoch in the `withdrawalEpochs` list (if it is not already present). Up to `MAX_WITHDRAWAL` elements can be added to the `withdrawalEpochs` list, or the call to `withdraw()` will revert. In case the list is full, the Aera vault operator has to claim some withdrawals to free space in the list.

The withdrawals list `withdrawalEpochs` is used again in the `claim()` function, which accepts as parameter an epoch number. The `claim()` function forwards the claim to Symbiotic, and then removes the epoch number from the `withdrawalEpochs` list. By removing it from the list, its amount is no longer used when calculating `SymbioticAdapter.balanceOf()`. Methods `claimBatch()` and `claimAll()` allow to respectively claim withdrawals for multiple epochs in a single call, or claiming all the withdrawals for the *claimable* epochs recorded in `withdrawalEpochs`, by forwarding the calls to Symbiotic's `claimBatch()`. Like for `claim()`, the epochs which have been claimed are removed from the `withdrawalEpochs` list.

Rewards can be claimed by the Aera vault by calling the `claimRewards()` function of *SymbioticAdapter*, which forwards the request to the specified `stakerRewards` contract. The contract to which the request is forwarded has to be whitelisted through function `toggleStakerRewardWhitelist()`.

Some other administrative functions are present in the contract:

- `setAeraVault()` allows the Aera vault of the adapter to renounce its role and set a new vault Aera that will be capable of administrating the adapter.
- `setMaxWithdrawal()` allows the Aera vault to set the maximum size of the `withdrawalEpochs` array.
- `execute()` allows the adapter owner (distinct from the Aera vault) to perform arbitrary external calls.

### 2.2.4 Trust model and roles

Symbiotic-Aera Adapter uses OpenZeppelin's AccessControl module to implement roles, which grant access to privileged functions through the `onlyRole()` modifier. Two roles are defined in Symbiotic-Aera Adapter: `OWNER` and `VAULT_ROLE`.

The `collateral` token can be a token with fees or a token implementing callbacks to the sender or the receiver, rebasing tokens however are not supported.

`VAULT_ROLE` has access to the `deposit()`, `withdraw()`, and `claim*()` functions. `VAULT_ROLE` is fully trusted and expected to behave correctly. A single address with `VAULT_ROLE` is ever present at any time. This address is the same as the value of the `aeraVault` state variable.

`OWNER` has access to the `execute()` function. `OWNER` is fully trusted and expected to behave correctly. `OWNER` can specify another `OWNER` through the `grantRole()` function. There can be multiple `` `OWNER `` at a time.

The *SymbioticAdapter* contract is expected to be deployed as an upgradeable proxy. The owner of the proxy is fully trusted.

**Version 2** fixes issues and allows `claim()` to claim for an epoch that is not in the memorized `withdrawalEpochs` list, allowing for donations to the vault.

# 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

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 | 0 |
| Low-Severity Findings | 1 |

- Slashing Penalties Avoidance Code Corrected

| | |
|---|---|
| Informational Findings | 2 |

- Missing Events Code Corrected
- Zero Address Will Lead to a Self DoS Code Corrected

## 6.1 Slashing Penalties Avoidance

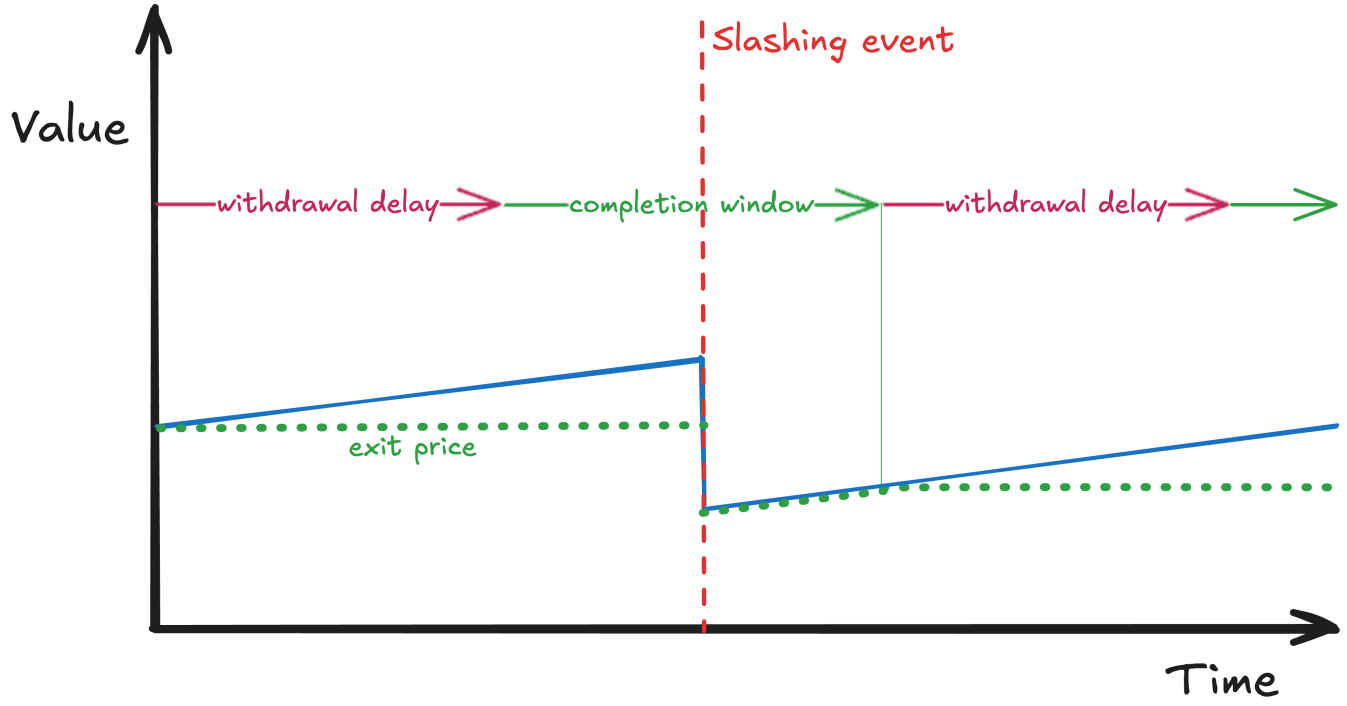
Design Low Version 1 Code Corrected

*CS-SWELL-SAA-001*

A user can try to avoid slashing penalties by withdrawing just before the penalty is applied and redepositing afterwards.

This is possible for a user if the withdrawal delay enforced on the Swell's vault is less than the time it takes for the slashing penalty to be applied. In this case a user just needs to start the withdrawal process from the Swell vault as soon as they anticipate a slashing, and then redeposit after the penalty is applied. The configuration of Swell's withdrawal delay needs therefore to be adapted to the duration of a Symbiotic epoch.

Another possibility for a malicious user that wishes to avoid a slashing penalty is to constantly keep an open withdrawal request. The current configuration of swBTC is 3 days for withdrawal delay and 3000 days for completion delay. A user can choose to start a withdrawal, and then wait before claiming the withdrawal. If a slashing happens during the 3000 days of the completion delay, the user can just withdraw before the slashing event is applied (frontrun it) and redeposit afterwards. This gives probabilistically a high ($\frac{3000}{3+3000}$) chance of avoiding serious slashings. This is described in the schema below. Since a user is always getting the worst price between the price at the time the withdrawal is started and the time of completion when withdrawing, he will withdraw at the price when the withdrawals request was made, missing the reward for time between the withdrawal is initiated and claimed. The claiming price is the green-dotted line on the schema. During that time the share price (in blue) will slowly increase thanks to rewards. Now if a slashing event happens during the completion window, the user will be able to withdraw at the price of the green dot line and not suffer from the difference between the green dot line and price after the slashing event. The user can then redeposit at a lower share price after the slashing event, get more shares than before, and restart the process. If no substantial slashing occurs, the user can simply cancel the withdrawal reward, and initiate a new one with a share price that includes the rewards.




---

**Code corrected:**

The issue was fixed by changing the mechanism of `DelayedWithdraw`. In the new version, a user cannot modify or cancel a withdrawal request and the time to complete this withdrawal is unlimited. Thanks to this change, a user cannot use the previously described strategy to avoid slashing penalties. This file is out of scope for this report, please find the corresponding report here.

## 6.2 Missing Events

Informational | Version 1 | Code Corrected

*CS-SWELL-SAA-003*

`SymbioticAdapter.initialize` is not emitting an `AeraVaultSet` event. The initial `aeraVault` value therefore cannot be deduced by events only. Note that this event could be improved by also including the old area vault.

The same remark applies also to `MaxWithdrawalUpdated`.

---

**Code corrected:**

The two events have been added to `SymbioticAdapter.initialize`.

## 6.3 Zero Address Will Lead to a Self DoS

Informational | Version 1 | Code Corrected

*CS-SWELL-SAA-007*

`SymbioticAdapter.initialize` does not sanitize the `_aeraVault` address, which can be set to zero. This will lead to a self DoS, as the contract has no way to set the address to a valid value without upgrading the contract. Without a valid address the purpose of the contract is not fulfilled.

**Code corrected:**

A check has been added to ensure that the address is not zero.

# 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 Natspec Issues

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

*CS-SWELL-SAA-004*

Some issues that have been identified in the natspec specification of `SymbioticAdapter`:

`name` and `symbol` are under the `Mutable state variables` natspec category, but they are not mutable.

`MAX_WITHDRAWAL` is under `Constants` natspec category but has a setter. The naming style in all caps is also unsuited for a mutable variable. Also note that the variable name is misleading as it doesn't represent the maximal withdrawal amount but the number of ongoing withdrawals (plural).

The natspec of `getWithdrawalEpochs` suffers from a typo.

---

**Code partially corrected:**

- `name` and `symbol` are now under the correct natspec category.
- `MAX_WITHDRAWAL` has also been moved but still has the same naming issue.
- The typo in `getWithdrawalEpochs` has been fixed.

## 7.2 Optimizations

**Informational** **Version 1**

*CS-SWELL-SAA-005*

Some possible gas and fee optimizations have been identified in the code. The following list provides an overview of the optimizations that could be implemented:

1. `SymbioticAdapter._findEpochIndex` could be optimized by removing the `curEpoch` variable.
2. `SymbioticAdapter.deposit` is not fee efficient to use if the token has a transfer fee because two transfers are performed, each one deducting the fee. In general, it also cost more gas than a direct transfer.
3. In `SymbioticAdapter`, the role `VAULT_ROLE` can be removed and the `onlyRole(VAULT_ROLE)` can be simplified, since only address with `VAULT_ROLE` is guaranteed to be value of the `aeraVault` variable.
4. `SymbioticAdapter.claim` always need to iterate over the array. This can be avoided by letting the user select an index instead of an epoch.

## 7.3 The Minimal Value of `MAX_WITHDRAWAL` Is Too Restrictive

**Informational** **Version 1**

*CS-SWELL-SAA-006*

The minimal value of `MAX_WITHDRAWAL` is enforced to be 1, but this can be too restrictive for an effective use of the adapter.

Indeed, if `MAX_WITHDRAWAL` is set to 1, it will only allow to vault to have one withdrawal at a time forbidding the vault to request withdrawals at each epoch, because once every two epochs, the `withdrawalEpochs` list will be full, but the withdrawal will not yet be claimable. `MAX_WITHDRAWL == 1` only allows one claim every two epochs at most.

# 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 Privileged Roles Capabilities

Note Version 1

Privileged roles of the system, namely the `OWNER` and the `VAULT_ROLE` roles can perform actions that can harm the system.

More specifically, the `OWNER` role has access to the `execute()` function of Symbiotic-Aera Adapter, which allows them to perform any action on the Symbiotic vault. In particular, beside malicious actions, they should be aware that interacting directly with the Symbiotic vault `withdraw()` and `claim()` can invalidate the correctness of the `withdrawalEpochs` list, and the adapter will report an incorrect `balanceOf()` as consequence.

## 8.2 Unfair Rewards

Note Version 1

Users and administrators of the system should be aware that the distribution of the rewards might be unfair. In fact, a depositor that enters the Swell vault just before Symbiotic rewards are claimed, will receive a part of the rewards of users that have been providing liquidity for the entire rewarded period. The unfairness is reduced when the rewards are claimed frequently and consecutive claims have similar value.