PUBLIC

Code Assessment

of the Collateral Token Smart Contracts

May 14, 2024

Produced for

SYMBIOTIC

by

CHAINSECURITY

Contents

1 Executive Summary

Dear SymbioticFi Team,

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

Symbiotic Protocol offers a basic implementation of a Collateral token. A Collateral token is an ERC20 token that wraps an underlying asset to be staked within the Symbiotic Protocol.

The most critical subjects covered in our audit are the safety of the funds, the internal accounting, and the correct use of the Permit2 contract. Security regarding all the aforementioned subjects is high.

The general subjects covered are access control, gas efficiency, specification and documentation, and testing. 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
• Specification Changed	1

2 Assessment Overview

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

2.1 Scope

The assessment was performed on the source code files inside the Collateral Token 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	13 May 2024	64a98f83bfc303d9bcc909c169c2db6cbf959c02	Initial Version
2	14 May 2024	e1892732cb6f5c57dfd40a7b3713642f92a993bb	Fixes

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

The following contracts in the `src/contracts/` directory are in scope:

- `Factory.sol`
- `defaultCollateral/DefaultCollateral.sol`
- `defaultCollateral/DefaultCollateralFactory.sol`
- `libraries/Permit2Lib.sol` (diff only)

2.1.1 Excluded from scope

The `Permit2Lib.sol` was only reviewed as a diff of Uniswap's Implementation. Moreover, all the libraries used are considered to work as intended. There's no control over which underlying assets are going to be used. This means, the users should make sure they trust the tokens they're going to be using with the system. The system makes use of Uniswap's Permit2 contract deployed at address 0x000000000022D473030F116dDEE9F6B43aC78BA3. The contract is assumed to function correctly.

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.

Symbiotic Protocol proposes to tackle the problem of having to explicitly lock assets in Proof-of-Stake systems, which intrinsically require the possibility of immediate slashing of the stake, in case an operator misbehaves. As a remedy to this inconvenience, Collateral Token defines the concept of a generic `Collateral`, which represents an asset without the need to actually hold it or lock it: the way the asset is backed remains unspecified, and is left to the concrete implementations. The asset is redeemable through the `Collateral.issueDebt()` function: having such a stake slashed in a POS system, then, just means that the redeemable amount decreases.

Collateral Token further offers a first example implementation, called `DefaultCollateral`, which represents an ERC20 asset, and effectively falls back to the simple case of locking up a stake to receive an equivalent amount of collateral.

In what follows, we detail the functionality of the main smart contracts in the system.

2.2.1 DefaultCollateral

This is a basic implementation of the `Collateral` specification, which has an ERC20 token as its underlying asset. It lets anyone deposit an amount of the asset (up to a global limit, set by a trusted entity), in exchange for an equal amount of the `DefaultCollateral` ERC20 token; a version also exists that integrates with `Permit2` in order to pull assets with a signature. This token can later be burned, again in exchange for an equal amount of the backing asset. It is deployed behind a non-upgradeable proxy. It exposes the following state-changing functions:

- `deposit()`: pulls the specified amount of asset from the user, minting an equal amount of `DefaultCollateral` token for the specified recipient. The total supply is capped by a `limit`, that is set at initialization time, and can be increased by a trusted `limitIncreaser` address. This function is overloaded: one with the same name exists that pulls assets through `Permit2` with a user-provided signature.
- `withdraw()`: burns the specified amount of `DefaultCollateral` token, and sends an equal amount of backing asset to the specified receiver.
- `issueDebt()`: acts similarly to `withdraw()` (burn tokens + send back asset), but additionally updates the debt-tracking variables. The debt is considered as instantly repaid, therefore the public variables tracking the *outstanding* debt are left unaltered (and thus will always equal 0), whereas the *growing accumulators* tracking the *repaid* debt are increased by the appropriate amount.
- `increaseLimit()`: increases the limit on the total supply by the specified positive delta. Only callable by the `limitIncreaser`.
- `setLimitIncreaser()`: hands over the `limitIncreaser` privileges to a new address. Only callable by the current `limitIncreaser`.

2.2.2 DefaultCollateralFactory

A contract that lets anyone deploy a `DefaultCollateral` instance linked to the specified ERC20 underlying token by calling `create()`. There can be multiple collateral tokens deployed for the same underlying. Note that the tokens cannot be rebasing.

2.3 Roles and Trust Model

The system defines the following roles:

- `limitIncreaser`: the `DefaultCollateral` recognises a `limitIncreaser` address as a configuration role.
- End users: they can deploy a new collateral token, and they can deposit and withdraw assets from the system.

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:

- 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	0
Low-Severity Findings	1

- Specification Mismatch Specification Changed

Informational Findings	1

- Gas Optimizations Code Corrected

6.1 Specification Mismatch

Correctness Low Version 1 Specification Changed

CS-SYFC-001

The file `specs/Collateral.md` specifies that

```
The only way to obtain `Collateral.asset()` is through the issueDebt method.
```

However, the function `DefaultCollateral.withdraw()` also allows to obtain the underlying asset.

Specification changed:

The specification has been extended. It's currently stated:

1. It must be possible to obtain *Collateral.asset()* through the *issueDebt()* method. However, there could be other ways to do it.

6.2 Gas Optimizations

Informational Version 1 Code Corrected

CS-SYFC-002

The function `DefaultCollateral.deposit()` could detect the failure condition on the deposit limit with an early check, before minting the collateral token. This would save gas for reverting calls.

Code corrected:

The check was moved earlier.