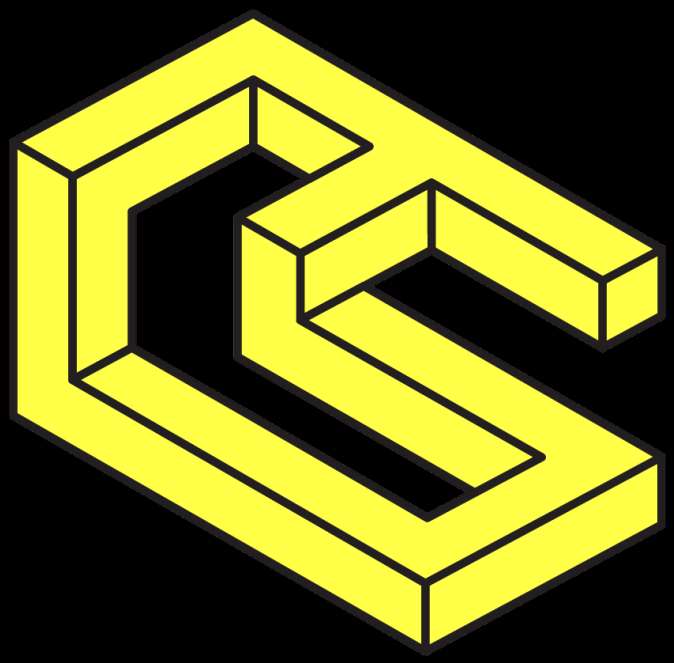

Ribbon Finance Smart Contracts Review

By: ChainSafe Systems

November 2021

Ribbon Finance Smart Contracts Review

Auditor: Oleksii Matiiasevych

WARRANTY

1. Introduction

Ribbon Finance requested ChainSafe Systems to perform a review of the Ribbon V2 smart contracts. The contracts can be identified by the following git commit hash:

`cf498d16fc4a51e38ada887467735a3272fc5aed`

There are 10 contracts/libraries in scope.

After the initial review, Ribbon Finance team applied a number of updates which can be identified by the following git commit hash:

`8328177fc188a344ea22591e872077d100a9e8a3`

Additional verification was performed after that.

2. Disclaimer

The review makes no statements or warranties about utility of the code, safety of the code, suitability of the business model, regulatory regime for the business model, or any other statements about fitness of the contracts for any specific purpose, or their bug free status. The review documentation below is for internal management discussion purposes only and should not be used or relied upon by external parties without the express written consent of ChainSafe Systems.

3. Executive Summary

All the initially identified issues were promptly fixed and are not present in the final version of the contract, except for one minor point which is planned to be addressed in the next version of the protocol.

There are **no** known compiler bugs for the specified compiler version (0.8.4), that might affect the contracts' logic.

There were 0 critical, 7 major, 5 minor, 23 informational/optimizational issues identified in the initial version of the contracts. An additional issue of excessive fee charge for new deposits was identified by the Ribbon Finance team, which is not included here. The issues found in Ribbon contracts were not present in the final version of the contracts, and no new issues were discovered. They are described below for historical purposes. The remaining issues were about insignificant optimizations (5.1, 5.3, 5.4) and a minor problem which could lead to a temporary inability to complete scheduled withdrawal until the end of the current round. See 5 for more details.

Ribbon V2 manages deposited funds to create options on the OPYN platform and then sell those options on GnosisAuction. There are 2 special vaults that use Yearn and STETH wrapped assets as collateral. This review is made on the **assumption** that OPYN, GnosisAuction, Yearn and STETH products are secure and safe to use. If that assumption is wrong, then all deposited funds could be lost.

Ribbon V2 vaults by itself are deployed using an `upgradable proxy` pattern that is supposed to be managed by Ribbon multisig wallet. This implies that the following review only applies to the exact version identified by the commits in section (1). As soon as Ribbon decides to utilize the upgrade function, this review becomes void, as the upgrade could change any piece of logic, ultimately `taking hold of any deposited funds.`

It is the users' responsibility to make sure that the Ribbon multisig is controlled by a sufficient number of trusted members.

We believe that while the upgradability could be useful, and safe, in the early stages of the protocol, the increase of total value locked (TVL) in the contract might cause unnecessary psychological pressure on the **manager** (being a group might lower the risk, but not remove it), by creating an incentive to seize the power for themselves and take hold of all the funds.

This engagement with the Ribbon Finance team involved plenty of discussions on how to improve protocol and solve issues efficiently, which we enjoyed greatly.

4. Line-by-Line Review

4.1. GnosisAuction, line 65: Optimization, no need to poll balance of `otokens` again, reuse `oTokenSellAmount` variable instead.

4.2. VaultLifecycle, line 193: **Minor**, using `newPricePerShare` to calculate `queuedWithdrawAmount` is not exactly correct, as the shares were queued at a variety of prices.

4.3. VaultLifecycle, line 248: Note, the `mintAmount` value calculation could be simplified to: (depositAmount*10**Vault.OTOKEN_DECIMALS*10**Vault.OTOKEN_DECIMALS) / (oToken.strikePrice()*10**collateralDecimals)

4.4. VaultLifecycle, line 254: **Minor**, the `collateralDemicals.sub(8)` expression will revert on collateral with decimals < 8. The `scaleBy` calculation should be put inside of the if condition.

4.5. VaultLifecycle, line 257: Note, the `mintAmount` value for collaterals with < 8 decimals should probably be `scaledUp` instead of down.

4.6. VaultLifecycle, line 263: Note, the `safeApproveNonCompliant` call could be replaced with `CustomSafeER20.safeApprove.`

4.7. VaultLifecycle, line 294: Note, 'deposited asset' comment should state 'option address'.

4.8. VaultLifecycleSTETH, line 158: **Minor**, using `newPricePerShare` to calculate `queuedWithdrawAmount` is not exactly correct, as the shares were queued at a variety of prices.

4.9. VaultLifecycleSTETH, line 205: Note, the double approve comment is outdated. Helper is handling it all now.

4.10. VaultLifecycleSTETH, line 364: **Major**, specifying only the `minETHOut` is not enough, as the `amountToUnwrap` could be manipulated with frontrunning. Consider requiring `amountETHOut>=minETHOut` instead.

4.11. VaultLifecycleYearn, line 164: **Minor**, using `newPricePerShare` to calculate `queuedWithdrawAmount` is not exactly correct, as the shares were queued at a variety of prices.

4.12. VaultLifecycleYearn, line 266: **Major**, invalid `underlyingTokensToWithdraw` calculation, should use `ShareMath.sharesToAsset` instead.

4.13. BaseVaults\base\RibbonVault, line 263: Note, the `setManagmentFee` function should emit `ManagmentFeeSet` event.

4.14. BaseVaults\base\RibbonVault, line 288: Note, the `setCap` function should emit `CapSet` event.

4.15. BaseVaults\base\RibbonVault, line 524: Optimization, excessive assignment of `depositReceipts[msg.sender].amount` if condition is not met.

4.16. BaseVaults\base\RibbonVault, line 552: Optimization, the overflow check is not needed as the Solidity compiler 0.8+ performs it implicitly.

4.17. BaseVaults\base\RibbonVault, line 596: **Major**, adding the `queuedWithdrawAmount` to the locked balance for fee calculation is not always right. Consider a case last locked was 100 and queued 10, nothing changed in the current round, so current locked is 100 and queued is 10 again, the fee will be calculated from (100+10)-100 = 10. While there was no profit.

4.18. RibbonThetaVault, line 384: Note, the `unlockedAssedAmount` variable has a typo and should be named `unlockedAssetAmount`.

4.19. STETHVault\base\RibbonVault, line 157: Note, the 'LDO contract' comment should be replaced with 'wstETH contract'.

4.20. STETHVault\base\RibbonVault, line 284: Note, the `setManagmentFeeSet` function should emit `ManagmentFeeSet` event.

4.21. STETHVault\base\RibbonVault, line 307: Note, the `setCap` function should emit `CapSet` event.

4.22. STETHVault\base\RibbonVault, line 548: Optimization, excessive assignment of `depositReceipts[msg.sender].amount` if condition is not met.

4.23. STETHVault\base\RibbonVault, line 576: Optimization, the overflow check is not needed as the Solidity compiler 0.8+ performs it implicitly.

4.24. STETHVault\base\RibbonVault, line 622: **Major**, adding the `queuedWithdrawAmount` to the locked balance for fee calculation is not always right. Consider a case last locked was 100 and queued 10, nothing changed in the current round, so current locked is 100 and queued is 10 again, the fee will be calculated from (100+10)-100 = 10. While there was no profit.

4.25. RibbonThetaSTETHVault, line 402: Note, the `unlockedAssedAmount` variable has a typo and should be named `unlockedAssetAmount`.

4.26. YearnVaults\base\RibbonVault, line 281: Note, the `setManagmentFee` function should emit `ManagmentFeeSet` event.

4.27. YearnVaults\base\RibbonVault, line 304: Note, the `setCap` function should emit `Capset` event.

4.28. YearnVaults\base\RibbonVault, line 559: Optimization, excessive assignment of `depositReceipts[msg.sender].amount` if condition is not met.

4.29. YearnVaults\base\RibbonVault, line 588: Optimization, the overflow check is not needed as the Solidity compiler 0.8+ performs it implicitly.

4.30. YearnVaults\base\RibbonVault, line 628: **Major**, adding the `queuedWithdrawAmount` to the locked balance for fee calculation is not always right. Consider a case last locked was 100 and queued 10, nothing changed in the current round, so current locked is 100 and queued is 10 again, the fee will be calculated from (100+10)-100 = 10. While there was no profit.

4.31. YearnVaults\base\RibbonVault, line 686: **Major**, the `upgradeYearnVault()` function will not work correctly as long as `asset.balance(address(this))>0.`

4.32. YearnVaults\base\RibbonVault, line 688: Note, instead of `unwrapYieldToken` call, should use simple `collateral.withdraw().`

4.33. YearnVaults\base\RibbonVault, line 689: **Major**, asset amount is expected instead of collateral amount.

4.34. RibbonThetaYearnVault, line 289: **Minor**, the `commitAndClose` function could be executed twice resetting the `lastLockedAmount` to 0. This will result in fees taken from the whole amount.

4.35. RibbonThetaYearnVault, line 423: Note, the `unlockedAssedAmount` variable has a typo and should be named `unlockedAssetAmount`.

5. Line-by-Line Verification. Remaining and Acknowleged Issues

5.1. GnosisAuction, line 65: Optimization, no need to poll balance of `otokens` again, reuse `oTokenSellAmount` variable instead.

5.2. VaultLifecycle, line 257: **Minor**, using `newPricePerShare` to calculate `queuedWithdrawAmount` is not exactly correct, as the shares were queued at a variety of prices.

5.3. VaultLifecycle, line 315: Note, the `mintAmount` value calculation could be simplified to: (depositAmount*10**Vault.OTOKEN_DECIMALS*10**Vault.OTOKEN_DECIMALS) / (oToken.strikePrice()*10**collateralDecimals)

5.4. VaultLifecycle, line 332: Note, the `safeApproveNonCompliant` call could be replaced with `CustomSafeERC20.safeApprove`

5.5. VaultLifecycleSTETH, line 158: **Minor**, using `newPricePerShare` to calculate `queuedWithdrawAmount` is not exactly correct, as the shares were queued at a variety of prices.

5.6. VaultLifecycleYearn, line 164: **Minor**, using `newPricePerShare` to calculate `queuedWithdrawAmount` is not exactly correct, as the shares were queued at a variety of prices.

Oleksii Matiiasevych