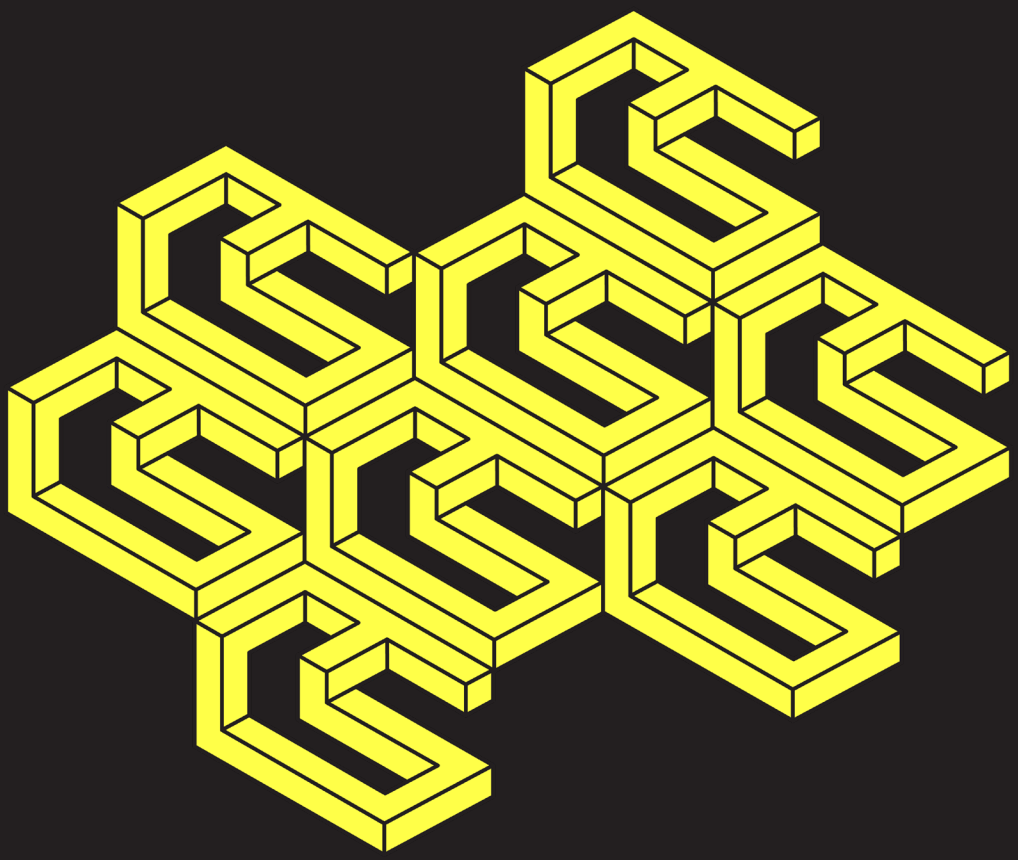

# Ribbon Finance Smart Contracts Review

By ChainSafe Systems

July 2021

ChainSafe

# Ribbon Finance Smart Contracts Review

Auditors: Oleksii Matiiasevych, Anderson Lee

## Warranty

This Code Review is provided on an “as is” basis, without warranty of any kind, express or implied. It is not intended to provide legal advice, and any information, assessments, summaries, or recommendations are provided only for convenience (each, and collectively a “recommendation”). Recommendations are not intended to be comprehensive or applicable in all situations. ChainSafe Systems does not guarantee that the Code Review will identify all instances of security vulnerabilities or other related issues.

## 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.

# 1. Introduction

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

`9972b464abe787e1bfafb309ebdeab1598e0f45d`

There are 7 contracts in scope.
After the initial review, Ribbon Finance team applied a number of updates which can be identified by the following git commit hash:

`4b9045d69a20a505b823f2cb8c32cdc3c6e3b79f`

Additional verification was performed after that.

# 2. 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 (0.7.3), that might affect the contracts' logic.

There were 1 critical, 3 major, 2 minor, 61 informational/optimizational issues identified in the initial version of the contracts. An additional major issue 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 (3.9, 3.18) and a minor problem which could lead to a temporary inability to complete scheduled withdrawal until the end of the current round. See 3.23 for more details.

RibbonThetaVault V2 manages deposited funds to create options on the OPYN platform and then sell those options on GnosisAuction. This review is made on the **assumption** that both OPYN and GnosisAuction platforms are secure and safe to use. If that assumption is wrong, then all deposited funds could be lost.

RibbonThetaVault V2 by itself is 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.

We are happy to continue working with the Ribbon Finance team who never stops to create and improve.

# 3. Line-by-line review

3.1. GnosisAuction, line 29: Note, `manager` not used in auction creation.

3.2. GnosisAuction, line 46: Optimization, in most cases double approval is not needed. Consider using a helper.

3.3. GnosisAuction, line 70: Note, `GnosisEasyAuction` is out of scope.

3.4. ShareMath, line 21: Note, check `pps > 1` should be `pps > PLACEHOLDER_UINT`.

3.5. ShareMath, line 25: Note, check `shares < type(uint104).max` should use `assertUint104()` or <=.

3.6. ShareMath, line 38: Note, check `pps > 1` should be `pps > PLACEHOLDER_UINT`.

3.7. ShareMath, line 42: Note, `shares < type(uint104).max` should use `assertUint104()` or <=.

3.8. ShareMath, line 49: Note, function `getSharesFromReceipt` is missing comments about 3 params.

3.9. ShareMath, line 60: Optimization, instead of checking if the round is already passed it is enough to check `pps > PLACEHOLDER_UINT`.

3.10. ShareMath, line 61: Optimization, check `!depositReceipt.processed` is not needed, `pps > PLACEHOLDER_UINT` is enough.

3.11. ShareMath, line 66: Note, check `sharesFromRound < type(uint104).max` should use `assertUint104()` or <=.

3.12. ShareMath, line 70: Note, check `unredeemedShares256 < type(uint128).max` should use `assertUint128()` or <=.

3.13. ShareMath, line 83: **Minor**, check `num < type(uint104).max` should use <=.

3.14. ShareMath, line 86: Note, function `assertUint128()` is not used.

3.15. ShareMath, line 87: Note, check `num < type(uint104).max` is wrong should be `num <= type(uint128).max`.

3.16. Vault, line 12: Note, leftover comment.

3.17. Vault, line 14: Note, `underlying` is not validated in `verifyConstructorParams()`.

3.18. Vault, line 22: Optimization: if `cap` is made `uint96`, would fit with `strikeSelection`.

3.19. Vault, line 41: Optimization, `premiumDiscount` would fit in `uint16`.

3.20. VaultLifecycle, line 37: Optimization, instead of passing the whole `vaultParams` consider using a storage pointer.

3.21. VaultLifecycle, line 38: Optimization, instead of passing the whole `vaultState` consider using a storage pointer.

3.22. VaultLifecycle, line 79: Optimization, `verifyOtoken()` should never fail and could probably be removed.

3.23. VaultLifecycle, line 155: **Minor**, `queuedWithdrawAmount` could be less than the sum of all withdrawals ready to be completed.

3.24. VaultLifecycle, line 207: Note, double division leads to loss of precision. Consider doing `mul(10**10).mul(10**collateralDecimals)` instead of second `div`.

3.25. VaultLifecycle, line 216: Optimization, `mintAmount > 0` is always `true` and should be removed.

3.26. VaultLifecycle, line 222: Optimization, double approval is not needed in most cases. Helper function could be useful here.

3.27. VaultLifecycle, line 421: Note, `performanceFee` is not validated to be below 100%.

3.28. VaultLifecycle, line 427: Note, `_vaultParams.decimals > 0` might not be always valid. There are some tokens with 0 decimals.

3.29. VaultLifecycle, line 445: Note, `86400 * 7` could be replaced by `'7 days'` for improved readability.

3.30. VaultLifecycle, line 446: Note, `dayOfWeek` starts with 0 which is Sunday. One of the ways to fix the following major issue is by doing `+1` here.

3.31. VaultLifecycle, line 449: **Major**, if `currentExpiry` is Sunday, then the `friday8am` variable will be in 12 days.

3.32. VaultLifecycle, line 456: Note, consider replacing `(60 * 60 * 24)` with `'24 hours'` and `(8 * 60 * 60)` with `'8 hours'`.

3.33. OptionsVaultStorage, line 14: Note, `StrikeOverride` is not used here.

3.34. OptionsVaultStorage, line 22: Note, `scheduledWithdrawals` is not used.

3.35. StrikeSelection, line 26: Optimization, `assetOracleMultiplier` should be made `immutable`.

3.36. StrikeSelection, line 80: Note, `strike` should start with increased price in case of a call.

3.37. StrikeSelection, line 117: Note, unreachable code.

3.38. StrikeSelection, line 140: Optimization, `min(lowerBoundDiff, upperBoundDiff) == lowerBoundDiff` condition should be replaced with `lowerBoundDiff <= upperBoundDiff` to improve readability.

3.39. RibbonThetaVault, line 89: Note, `WithdrawalFeeSet` event is not used.

3.40. RibbonThetaVault, line 105: Note, `InitiateGnosisAuction` event seems to only be needed to have event definition in the ABI.

3.41. RibbonThetaVault, line 178: Note, `managementFee` could be set to 0 here, but not later. Not validated to be below 100%. To not lose precision do: `_managementFee.mul(7).div(uint256(365))`.

3.42. RibbonThetaVault, line 201: Note, `uint16 newPremiumDiscount` is mismatched with state variable type `uint32 vaultState.premiumDiscount`.

3.43. RibbonThetaVault, line 225: Optimization, no need to cast to `uint16`, as the `managementFee` is `uint256`.

3.44. RibbonThetaVault, line 226: Note, `newManagementFee` is excessively cast to `uint256`. To not lose precision do: `_managementFee.mul(7).div(uint256(365))`.

3.45. RibbonThetaVault, line 251: Note, `newCap` is not validated, could be set to 0.

3.46. RibbonThetaVault, line 260: Note, ‘if the underlying is not WETH’ should be replaced with ‘if the asset is not WETH’.

3.47. RibbonThetaVault, line 292: Optimization, variable `currentRound` has a `uint16` type, while it is cheaper to use `uint256` local variables.

3.48. RibbonThetaVault, line 295: Note, check `totalWithDepositedAmount < vaultParams.cap` should use <=.

3.49. RibbonThetaVault, line 307: Optimization, variable `unredeemedShares` has a `uint128` type, while it is cheaper to use `uint256` local variables.

3.50. RibbonThetaVault, line 314: Optimization, variable `depositAmount` has a `uint104` type, while it is cheaper to use `uint256` local variables.

3.51. RibbonThetaVault, line 318: Optimization, check `!depositReceipt.processed` always passes and should be removed.

3.52. RibbonThetaVault, line 369: Note, checking `depositReceipt.round < currentRound` is inconvenient. If a deposit is made in the current round, then just don't process it. Redeem could still work. Another way would be to process deposits as part of `getSharesFromReceipt` and never here.

3.53. RibbonThetaVault, line 384: Optimization, `depositReceipts[].processed` is not needed at all. `amount == 0` is the same as `processed == true`.

3.54. RibbonThetaVault, line 404: Optimization, check `!depositReceipt.processed` is not needed. It will only fail if `receiptAmount == 0` which is checked balow.

3.55. RibbonThetaVault, line 410: **Major**, `vaultState.totalPending` must be decreased here as well.

3.56. RibbonThetaVault, line 428: Optimization, `topup` could be concluded from just `withdrawal.round == currentRound`.

3.57. RibbonThetaVault, line 430: **Critical**, `heldByVault` is not decreased after withdrawal allowing users to withdraw the same `unredeemedShares` multiple times.

3.58. RibbonThetaVault, line 442: Note, check `increasedShares < type(uint128).max` should use `assertUint128()` or <=.

3.59. RibbonThetaVault, line 444: Optimization, instead of `!withdrawal.initiated` could use `withdrawal.shares == 0`.

3.60. RibbonThetaVault, line 445: Optimization, `withdrawals[].initiated` is not needed. Could use `withdrawal.shares > 0` instead.

3.61. RibbonThetaVault, line 455: **Major**, `vaultState.queuedWithdrawShares` always increases.

3.62. RibbonThetaVault, line 472: Optimization, check `withdrawal.initiated` is not needed. If not initiated then `shares == 0` then `withdrawAmount == 0`.

3.63. RibbonThetaVault, line 473: Optimization, instead of checking that the withdrawal round has passed, fetch the `pricePerShare` which will be used/checked in `ShareMath.sharesToUnderlying`.

3.64. RibbonThetaVault, line 500: Note, if `commitAndClose()` is called 2 times, `lastLockedAmount` will become 0, and `performance fees` will be collected even after loss.

3.65. RibbonThetaVault, line 522: Optimization, `assertUint104(premium)` cannot fail because `GnosisAuction` asserts `uint96`.

3.66. RibbonThetaVault, line 620: Optimization, `VaultLifecycle.burnOtokens` returns value equal to `assetBalanceAfterBurn.sub(assetBalanceBeforeBurn)`.

3.67. RibbonThetaVault, line 685: Note, condition `managementFee > 0` could be removed as 0 `managementFee` will result in 0 `feeInAsset` anyway.

Anderson Lee

Oleksii Matiiasevych