Ribbon Finance Audit

SEPTEMBER 9, 2021 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

The Ribbon Finance team asked us to review and audit their Theta Vault and Delta Vault smart contracts. We looked at the code and now publish our results.

Scope

We audited commit `3fa3bec15ad1e2b18ad87f979b87a68368497f13` of the `ribbon-finance/ribbon-v2` repository. In scope were the following contracts:

- `GammaInterface.sol`
- `IERC20Detailed.sol`
- `IGnosisAuction.sol`
- `IRibbon.sol`
- `IRibbonThetaVault.sol`
- `IWETH.sol`
- `GnosisAuction.sol`
- `ShareMath.sol`
- `SupportsNonCompliantERC20.sol`
- `Vault.sol`
- `VaultLifecycle.sol`
- `OptionsVaultStorage.sol`

- `StrikeSelection.sol`
- `RibbonDeltaVault.sol`
- `RibbonThetaVault.sol`
- `RibbonVault.sol`

Update: *The Ribbon Finance team has partially fixed the issues identified in this report through pull requests in the audited Github repository.*

Overall Health

In general, we found the Ribbon team to be highly responsive, easy to work with, and open to feedback.

Most of the codebase was well documented, but there is still some room for improvement. We found the commit we audited to be fully functional, but harder than necessary to reason about at times. The codebase has numerous casts and calls to helper functions to assert that those casts are safe; they make the code harder to read and can be error prone. We noticed an inconsistent coding style, imprecise language and naming conventions, unnecessarily complex control flows, and occasional remnants of previous refactors.

The Ribbon team has been rapidly iterating on their contracts even while our audit was underway. We must note that safely porting over fixes from this audit to a codebase that has been substantially iterated on can be non-trivial. We recommend *all iterations* of the codebase undergo audits and that no code that has not been audited be relied upon to handle user funds.

System Overview

Ribbon Finance is a protocol for creating structured financial products on Ethereum. The system provides "vaults" that run automated strategies which allow users to earn yields on their deposits. Users pay vault fees for this service on a weekly basis, but only if the vault is profitable in that week.

Currently, Ribbon offers two types of vaults, Theta and Delta vaults.

Theta vaults run an automated options selling strategy, which consists of writing and auctioning *out of the money* options and collecting the premiums. In order to create and manage options, Theta vaults make use of Opyn's Gamma Protocol to mint `oTokens`, which represent the right to buy or sell a certain asset at a predefined price. Those `oTokens` are then auctioned via Gnosis Batch Auctions.

Delta vaults, on the other hand, provide a way to hedge long positions, implementing a strategy that consists of using a percentage of all the deposited funds to buy options from Theta vault auctions and claiming profits if those options expire *in the money*.

Privileged Roles

There are two main roles in the system, the `owner` and the upgradeability `admin`.

The `owner` of a vault is responsible for its correct operation and is required to periodically execute administrative functions throughout a vault's lifecycle. The `owner` can:

- Manage the ownership of the vault
- Change the deposit cap of a vault
- Change the performance and management fees charged to vault users
- Change the recipient for all of the collected vault fees

- Change the duration of Gnosis auctions
- Control the option premium that Delta vaults will pay
- Manually set a strike price that the Theta vaults will use
- Control the `step` and `delta` used in the automatic strike selection contract
- Close an existing position and create a new one
- Start a new round that makes use of the newly created option
- Burn `oTokens` that were not sold during an auction
- Set the percent of the Delta vault that will be used to purchase options

The `admin` has the power to completely overwrite the vault's implementation.

Initially, both the `owner` and `admin` accounts will be completely controlled by the Ribbon Finance team. Considering the amount of control that these roles have within the system, users must fully trust that the entities controlling those roles will always behave correctly and and in the best interest of the system and its users.

Findings

Here we present our findings.

Critical severity

[C01] Instant withdrawals can disable essential vault functionality

When users deposit some amount of asset into a vault, their newly deposited asset are considered *pending* until the beginning of the vault's next round. While pending, deposits are not being actively managed by the vault, and they are, appropriately, intentionally disregarded for many of the vaults internal accounting purposes. Pending deposits do not share in profits or losses of the vault while pending and they should not be included in the determination of the vault's price per share. Vaults keep a running count of the total pending asset in the `vaultState.totalPending` state variable so that they can actively disregard all of the pending asset where appropriate, and also so that they can mark all of the pending asset as active when the next round starts.

Delta vaults provide a `withdrawInstantly` function, which permits users to withdraw asset, including any pending asset, at any time. However, this function fails to update `vaultState.totalPending`, even after the user has withdrawn some amount of pending asset. As a consequence, for any sequence of deposits that are followed by instant withdrawals during the same round, `vaultState.totalPending` will only ever increase. This results in a mismatch between the number that is stored in `vaultState.totalPending` and the actual amount of pending asset the vault possesses.

This has far-reaching ramifications, because so many of the calculations in the vault are affected, including:

- The `balance` calculation within the `pricePerShare` function of `RibbonVault`
- The `pps` calculation in `accountVaultBalance` of `RibbonVault`
- The `roundStartBalance` calculation in the `rollover` function of `VaultLifecycle`

In many cases, OpenZeppelin's `SafeMath` is used to subtract the value of `vaultState.totalPending` from the *total* asset balance held by the vault, to exclude the former from being counted as active. This approach is reasonable and accurate, as long as `vaultState.totalPending` reflects the *actual* amount of pending asset in the vault. However, since `vaultState.totalPending` is never decremented when pending asset is withdrawn, it can become larger than the total balance. When this happens, many of the internal arithmetic operations of the vault overflow and then revert.

In this scenario, users can still withdraw funds, but the vault cannot carry out many other essential functions, including rolling to the next option. The only way to recover complete functionality would be to replace the logic of the Delta vault via an upgrade.

It is important to note that this critical loss of functionality can easily arise as a result of normal vault activity.

However, there is another scenario that could play out that stems from the failure to decrement `vaultState.totalPending` correctly. That scenario would probably be less likely to happen during the normal operation of the vault, but it can result in all user funds being locked in the vault. It would be trivial for a malicious party to put the vault in such a state.

In fact, this state can easily be achieved by performing the following steps immediately after an `oToken` auction ends:

1. Deposit an amount of asset equal to the current balance held by the vault (using a flash loan if needed).
2. Perform an instant withdrawal of all the funds that were just deposited. Then, because of the failure to decrement, `vaultState.totalPending` would be equal to the current balance of the vault.
3. Call the `claimAuctionOtokens` function, which would force a reevaluation of the price per share because of its `updatePPS` modifier.

While updating the price per share, the `roundStartBalance` would incorrectly be set to `0` because of the subtraction of `pendingAmount` from the `currentBalance` of the vault. When the modifier calls the `getPPS` function in `VaultLifecycle`, the price per share is derived from a multiplication of the `roundStartBalance`, always returning `0` under these conditions.

Whenever `roundPricePerShare` is equal to `0`, withdrawals will fail due to `require` statements in the `ShareMath` library's `underlyingToShares` and `sharesToUnderlying` functions that are called as part of every withdrawal. All user funds will be locked in the vault.

Consider decrementing `vaultState.totalPending` correctly whenever any amount of pending asset is withdrawn from a vault. Also consider expanding the test suite to cover more complicated deposit and withdrawal scenarios.

Update: *Fixed in commit `f882323642c23f02965adfa378a6b062e0ca65ce` of PR#79.*

High severity

[H01] Corruptible storage upgradeability pattern

Both `RibbonThetaVault` and `RibbonDeltaVault` inherit from `RibbonVault` and either `OptionsThetaVaultStorage` or `OptionsDeltaVaultStorage`, respectively. `RibbonVault` itself inherits from `OptionsVaultStorage`, so that the latter is also inherited by both the `RibbonThetaVault` and `RibbonDeltaVault`.

All of these storage contracts reside in `OptionsVaultStorage` as "top-level" storage contracts. Each of these top-level contracts inherit from additional "storage contracts" that are suffixed with a version number, e.g.: `OptionsVaultStorage` inherits from `OptionsVaultStorageV1`. The intent is to allow for upgradeability of the storage layout of the top-level storage contracts by just having them inherit from additional storage contracts as necessary.

The inline comments explain the intention well:

> When we need to add new storage variables, we create a new version of OptionsVaultStorage e.g. OptionsVaultStorageV, so finally it would look like contract OptionsVaultStorage is OptionsVaultStorageV1, OptionsVaultStorageV2

The issue is that multiple top-level storage contracts are being inherited by the `RibbonThetaVault` and `RibbonDeltaVault` contracts. This storage upgradeability pattern is not composable in this manner without modifications. In both cases, upgrading `OptionsVaultStorage` to include additional storage slots would shift storage slots from any subsequently inherited top-level storage contracts, thus corrupting the storage layout.

Since the newly upgraded smart contracts would be reading from some storage slots that held data which would no longer correspond to the new storage layout, the system would break in an unpredictable manner that would be dependent on the number of storage slots added as part of the upgrade.

Consider reserving storage slots in `OptionsVaultStorage` so that any future upgrades to that contract will not shift the storage slots of other top-level storage contracts. A common pattern is to use a `__gap` array for this purpose and to decrement the number of slots used by the `__gap` as necessary during upgrades. Alternatively, consider making `OptionsVaultStorage` non-upgradeable and only allowing `OptionsThetaVaultStorage` and `OptionsDeltaVaultStorage` to be upgradeable. Also consider adding additional inline comments explaining the limitations of this upgradeability pattern in practice to avoid future iterations reintroducing corruptible upgradeable storage layouts.

Update: *Fixed in commit `b362625ae3c6917f41e801ddf7a5bd5d4d8fb295` of PR#85.*

Medium severity

[M01] Documentation issues

Although many functions in the codebase are well documented, there are numerous other functions that are lacking sufficient inline documentation. For example:

- Missing NatSpec for all functions of `VaultLifecycle`.
- Missing NatSpec for the return value of `accountVaultBalance` in `RibbonVault`.
- Missing NatSpec params for `initialize` in `RibbonThetaVault`.
- Missing NatSpec for return value of `burnOtokens` in `VaultLifecycle`.
- In `RibbonThetaVault`, block comments are used to group functions into categories. `withdrawInstantly` is incorrectly categorized as a `SETTER`, and `setStrikePrice` is incorrectly categorized as a `VAULT OPERATION`.

There are also instances of misleading documentation. For example:

- The comment `address to transfer to` should be `address to transfer from`.
- The comment `3 decimals` should be `2 decimals`.
- The comment `Burning all otokens that are left from the gnosis auction`, is misleading. In fact the *provided* `amount` of oTokens are burned.
- The comment that references `closeShort` is confusing. That is not the name of a function and it is unclear if this should be `_closeShort`, `settleShort`, or the name of the action being sent to the `gammaController`.
- The comment `we need this contract to receive so we can swap at the end` is misleading, as the value that this comment is referring to is not used by the `OpenVault` action.
- The comment `but gnosis will transfer all the otokens` should be `otokens will be transferred to gnosis`.

To further clarify intent and improve overall code readability, consider adding additional inline documentation, fixing documentation mistakes, and carefully verifying all documentation throughout the codebase.

Update: *Fixed in commit `8b7bcedd0050823842b2b65c5d459fc90542c82a` of PR#80. However, this PR also introduces a few unaudited functions such as `getVaultFees` in `VaultLifecycle`, `setOptionsPremiumPricer` in `RibbonThetaVault`, and others. Additionally, the `verifyOtoken` function was modified to accept additional arguments, but those arguments are not supplied where the function is used.*

[M02] Rounds with no locked asset can lead to fee miscalculations

In the `_closeShort` function of `RibbonThetaVault`, the state variable `vaultState.lastLockedAmount` is set to `vaultState.lockedAmount` unless the latter is zero, in which case `lastLockedAmount` is just copied back on to itself. Aside from using gas for an unnecessary `SSTORE`, this has the effect of essentially skipping over recording rounds with no `lockedAmount` of asset.

This is problematic, because prior locked amounts are used to assess performance fees and management fees in the `_collectVaultFees` function.

The performance fee is directly impacted by the difference between the amount of locked asset in the current round and the round immediately prior. Disregarding a round that legitimately had no `lockedAsset` (an empty round), distorts this calculation.

In some cases, the fee assessment will just be incorrectly skipped altogether.

The `_collectVaultFees` function relies on the condition that the current round's locked balance must be greater than the prior round's locked balance to assess any fees at all. Since the empty round was not recorded, in the case where the round prior to the empty round had more asset than the current round after profits, *no* fees would be assessed.

Consider keeping track of all actual amounts of locked asset to ensure that vault fees are always assessed correctly.

Update: *Fixed in commit `ce98d07920da508ee6ad4907d55bf4e901a0aa5b` of PR#81. The code no longer skips recording rounds with no `lockedAmount` of asset. However, code to initialize the value of `vaultState.lastLockedAmount` was also removed. The latter removed code was not directly related to to the issue outlined above. Not initializing the value of `vaultState.lastLockedAmount` to the vaults' balance of `asset` should not be problematic for the assessment of fees for the general case where the token balance at time of initialization is zero. Outside of that context, given the scope constraints of fix reviews, we have not considered the implications of that change.*

[M03] Errors and omissions in events

Throughout the codebase, events are used to signify when changes are made to the contracts. However, many events lack indexed parameters or are missing important parameters. Some sensitive actions are lacking events altogether.

Events lacking indexed parameters include:

- The `InitiateWithdraw`, `CapSet` and `Withdraw` events in `RibbonVault` should index their `address` arguments.
- Most of the events in `RibbonThetaVault` should index their `address` arguments.
- The events in `GnosisAuction` should index their `address` arguments.
- The events in `StrikeSelection` should index their `address` arguments.

Events missing important parameters include:

- The `Deposit` event in `RibbonVault` does not emit the address of the account performing the deposit.
- The `CollectVaultFees` event does not emit the `feeRecipient` nor the `managementFee`.

Event inconsistencies include:

- The `Deposit`, `InitiateWithdraw`, `Redeem` and `CollectVaultFees` events use inconsistent parameter types for rounds in `RibbonVault`.

Sensitive actions that are lacking events include, but are not limited to:

- The `setFeeRecipient` function in `RibbonVault`

- The `setStrikePrice` function in `RibbonThetaVault`
- The `baseInitialize` function in `RibbonVault` does not emit events for most of the state variables it sets.
- The `initialize` function in `RibbonThetaVault` does not emit events for most of the state variables is sets.
- The `initialize` function in `RibbonDeltaVault` does not emit events for most of the state variables is sets.

Consider more completely indexing existing events, adding new indexed parameters where they are lacking, and being consistent with event argument types to avoid hindering the task of off-chain services searching and filtering for events. Consider emitting all events in such a complete manner that they could be used to rebuild the state of the contract.

Update: *Partially fixed in commit `1d3e518eed09598c1f9a9105c94032a96a12de6e` of PR#82.*

[M04] Inconsistent usage of reentrancy guard

Throughout the codebase, most of the `external` and `public` functions that modify state use the `nonReentrant` modifier in order to explicitly prevent reentrancy.

A notable exception to this is the `startAuction` function in `RibbonThetaVault`. It performs a call to the library function `VaultLifecycle.startAuction`, which then makes a call to the library function `GnosisAuction.startAuction`, where an external call to `IGnosisAuction.initiateAuction` is made.

At the end of the originally called `startAuction` function, the `optionAuctionID` state variable is modified, violating the recommended `Check-Effects-Interaction` pattern.

Even if there are no immediate negative implications of allowing reentrancy in this case, this pattern makes it easier for future iterations of this codebase, or the codebase of the projects this project integrates with, to introduce vulnerabilities.

Consider always explicitly protecting functions from reentrancy when they make external calls to third-party code, as is the case with the `startAuction` function in `RibbonThetaVault`.

Update: *Fixed in commit `fc3ebff1b54340c0045989fc3e8e11f2fbd81cae` of PR#83. However, an unimplemented, unaudited modifier `onlyKeeper` was added to the `startAuction` function alongside the suggested `nonReentrant` modifier.*

[M05] Vault fees are miscalculated

In the `_collectVaultFees` function of the `RibbonVault`, if the vault was profitable during the preceding week, then vault fees are tabulated and assessed for that week.

If a user deposits funds mid-week, then those funds remain "pending" until the vault's next round begins. Since those funds are pending and are not actively participating in the vault, they should not share in any of the profits or losses that the vault incurs over that time period.

To enforce this, one component of the vault fees, the "performance fee", is charged on the amount of asset locked in the vault, less any pending amount (appropriately labeled `lockedBalanceSansPending`).

The second component of the vault fees, the "management fee", should be assessed against the same base of assets. However, the function fails to exclude the pending assets when determining the management fee – instead, it is incorrectly assessed against *all* of the locked assets, pending or not. This causes the assessed fees to be higher than intended.

To clarify and align the intent of the function with the implementation, and to avoid over-assessing vaults fees, consider excluding the amount of pending asset from the management fee calculation.

Update: *Fixed in commit `cb619dcc2407a0c568f25affbb33f5e617bcf152` of PR#73. **However**, that PR includes unaudited code from commits that we did not review. While we can confirm the changes in the referenced commit do address the issue in isolation, if* *they had been applied directly to the commit we audited, we cannot confirm that any of the prior unaudited changes the PR is based on have not introduced other issues.*

Low severity

[L01] Numerous `uint` types and resultant casts increase system complexity

Throughout the codebase, numerous unsigned integer (`uint`) values of various sizes are used. Those less than 256-bits (`uint256`) are used extensively. Non- `uint256` sizes are generally chosen to facilitate tight packing inside of `struct` s to save on storage costs. However, projects must carefully weigh the realized gas savings against the additional complexity such a design decision introduces.

Since the Ethereum Virtual Machine operates on 256-bit integers, additional operations must be performed to process non- `uint256` values. These additional operations increase the bytecode size of contracts and consume additional gas during execution. This non-trivial increase in complexity at the bytecode level is not the only place complexity is introduced, however.

Because this project relies extensively on math libraries such as OpenZeppelin's `SafeMath`, which expect `uint256` values, the codebase is filled with explicit casts transforming non- `uint256` values back in to `uint256` values before they can be used in the business logic of the project. Often times, the values are cast back to their original types after the business logic is complete. These numerous casts make the codebase harder to read and understand.

In response to the resultant increased propensity for inadvertent truncations, the codebase often uses helper functions to assert `uint` sizes before explicit downcasts. This makes explicit casting less error prone, but, again, increases the complexity and decreases the readability of the codebase.

To do less explicit casting in the business logic, functions routinely receive parameters as `uint256`, even when the associated underlying value is of a smaller type. For example, `redeem` and `_redeem` receive `uint256 shares` parameters but `unredeemedShares` is a `uint128`. In such cases, the helper functions are used to manually assert that `shares` fits in a `uint128`. This is much more error prone than letting the compiler enforce the proper types.

In other cases, there are no explicit checks that these casts will not overflow. For example, in `RibbonVault` when the `_depositFor` function accounts for new deposits, the `uint256 amount` variable, which represents the token amount being deposited, is cast to a `uint104`. At a glance this looks like an unsafe cast, but, in fact, a `require` statement earlier in the function implicitly ensures the cast is safe. This makes spotting *unsafe* casts, as reported in issue L14, harder to notice.

In addition to the general increase in complexity of the codebase, we also noted a few specific cases where these non- `uint256` values and associated casts led to additional confusion, unnecessary operations, or were particularly illustrative of how they lend themselves to the introduction of errors:

1. There is an assertion that the `shares` parameter in the `_redeem` function of `RibbonVault` is a `uint104` when it should be a `uint128`. This happens in the `ShareMath` contract's `getSharesFromReceipt` function as well. In fact, the appropriate type for `shares` values is often unclear, with the structs using `uint128` values and functions that deal with shares sometimes specifying `uint104` and other times specifying `uint128`.

2. In several cases, the `decimals` value is used for exponentiation of the literal base `10`. The practice of casting `uint` values used in business logic of the code to `uint256` is applied in these cases, even though it is unnecessary. Since Solidity 0.7.0, "Exponentiation and shifts of literals by non-literals will always use uint256 or int256 as a type."

3. In the `ShareMath` library, it is unnecessarily asserted that `shares` is a `uint104` *after* all calculations involving `shares` have

already been completed and despite the fact that `shares` is not being returned.

4. In `RibbonDeltaVault`, `receiptShares` is unnecessarily cast to `uint256`.

5. `vaultState.round` is unnecessarily cast to a `uint256`, only for it to be cast back to a `uint16` without modifying it. This happens again in the `initiateWithdraw` function and the `completeWithdraw` function.

To reduce the overall complexity of the code, consider using non-`uint256` values only when necessary. To reduce potential confusion, when smaller `uint` types are used, consider using the smaller types consistently to allow the compiler to help type-check values. To make the code easier to read, consider using the OpenZeppelin SafeCast library where possible to minimize the need for separate casting and type assertion operations. Alternatively, consider updating the codebase to Solidity 0.8 (as recommended in issue L12), which has checked math by default and could eliminate the need for explicit casting solely to perform `SafeMath` operations.

Update: *Fixed in commit `79ce9cbfb2d7102859ca096997bcd7c5852b5116` in PR#88. List item 5 from above was not modified, because the current behavior is correct. It is more gas efficient than alternatives. Many of the unnecessarily small `uint` function arguments were made to be `uint256`.*

[L02] Instances of unnecessarily convoluted control flow

Unnecessarily complicated code increases the potential for the introduction of bugs, decreases the readability of the codebase, and makes the project harder to reason about and maintain. As a result, even just occasional sections of unnecessarily complicated code can have a negative impact on the overall security of the project.

There are parts of the codebase that could benefit from being rewritten to reduce their complexity. For example the `_getBestStrike` function is currently written as follows:

```
function _getBestStrike(
uint256 finalDelta,
uint256 prevDelta,
uint256 strike,
bool isPut
) private view returns (uint256 finalStrike) {
if (isPut) {
if (finalDelta == prevDelta) {
finalStrike = strike.add(step);
} else {
finalStrike = strike;
}
} else {
if (finalDelta == prevDelta) {
finalStrike = strike.sub(step);
} else {
finalStrike = strike;
}
}
}
```

However, it could be simplified to something significantly shorter and much easier to read, such as the following pseudocode:

```
func _getBestStrike(
finalDelta,
prevDelta,
strike,
isPut
) {
if (finalDelta != prevDelta) {
return strike
}

return isPut ? strike.add(step) : strike.sub(step)
}
```

There are opportunities for similar simplifications throughout the codebase. For example:

- The lines 329 to 340 of `RibbonVault`
- The `_getBestDelta` function of `StrikeSelection`

To increase the readability, maintainability, and overall clarity of the codebase, consider trying to simplify control flow logic wherever possible, but particularly in the instances identified above.

Update: *Fixed in commit* `e61dfef6019e03f52f8c295bc8612b2afc3e5620` *in PR#89*.

[L03] Duplicated code

There are instances of duplicated code within the codebase. Duplicating code can lead to issues later in the development lifecycle and leaves the project more prone to the introduction of errors. Errors can inadvertently be introduced when functionality changes are not replicated across all instances of code that should be identical. Examples of duplicated code include:

- The whole `Vault` library is duplicated inside IRibbonThetaVault's file.
- Some of the functions of the vendored `DSMath` library are duplicated inside GnosisAuction and VaultLifecycle, albeit with different names.

Instead of duplicating code, consider having just one contract or library containing the duplicated code and using it whenever the duplicated functionality is required.

Update: *Fixed in commit* `e1ec5868835ee2ed718668695157bea11f209553` *of PR#90*.

[L04] Duplicated price per share calculations

A vault is simultaneously an ERC20 token, and balances of accounts correspond to the number of shares users hold. These shares represent how much of the collateral held in a vault can be withdrawn by each user when the current round ends. Throughout the lifecycle of a vault, the amount of collateral that backs each share changes, and the vault calculates this rate by dividing the total amount of collateral by the total share supply.

Multiple slightly different implementations of this calculation can be found throughout the codebase:

- In `updatePPS` of `RibbonDeltaVault`
- In `rollover` of `VaultLifecycle`

- In pricePerShare and accountVaultBalance of RibbonVault

Even though these are essentially equivalent to each other, having multiple different implementations for the same calculation leaves the project more prone to the introduction of errors. Refactors and changes made to some of these implementations might not get replicated over to the others.

Instead of relying on multiple equivalent but separate ways of calculating the price per share, consider consolidating this calculation into only one function that can be reused throughout the codebase when needed.

Update: *Fixed in commit 4e7890a4a110e6b7aacac875e486106edd908c10 in PR#91. Unrelated to the issue, the PR also removes the concept of an initialSharePrice from the system, a change which is outside the scope of this fix review.*

[L05] Inconsistent upper bounds on optionAllocationPct

The optionAllocationPct state variable in RibbonDeltaVault is required to be less than 100% upon initialization. However, the setOptionAllocation function requires the optionAllocationPct variable to be less than 10%.

Misleading comments about the number of decimals this variable should use makes the inconsistent require statements even more confusing.

Consider clarifying the documentation around this value and either using a consistent range for associated require statements or adding additional inline comments explaining the intentionality of the different upper bounds.

Update: *Fixed in commit 1bcdedeb197d0c270e1a7888ef4ad7729468fe2d of PR#94.*

[L06] Lack of input validation

Although most of the functions throughout the codebase properly validate function inputs, there are some instances of functions that do not. For example:

- verifyConstructorParams is called from baseInitialize of RibbonVault. However, neither function validates the input for the _managementFee argument of baseInitialize.
- verifyConstructorParams does not check that _vaultParams.minimumSupply < _vaultParams.cap. If this inequality does not hold, there could be states where it would be impossible to deposit asset into the vault.
- setFeeRecipient does not check that newFeeRecipient is not the same as the existing feeRecipient.
- setDelta and setStep do not perform any sort of input validation.
- StrikeSelection's constructor does not check that _delta < 10000, with the value 10000 representing an option delta of 1. The maximum value for an option's delta is 1, so this inequality must always hold.

To avoid the potential for erroneous values to result in unexpected behaviors or wasted gas, consider adding input validation for all user-controlled input, including owner-only functions.

Update: *Fixed in commit 83584f43ac494092cc744ae6c9b21eb5b23b7453 in PR#92.*

[L07] Magic numbers are used

Throughout the codebase, there are several occurrences of literal values with unexplained meaning. For example:

- In RibbonVault, both management and performance fees are percentages with 6 decimals, so 100 * 10**6 is inlined when performing checks.
- In RibbonVault's initRounds function, numRounds is required to be lower than 52.
- In RibbonThetaVault, premiumDiscount is limited to be in the [1, 999] range. The reason for this range is not documented,

and the parameters are hardcoded.

- In `RibbonThetaVault`, `auctionDuration` has to be bigger than `1 hour`, this value is inlined.
- In `GnosisAuction`, there is a division by `1000`.
- In `GnosisAuction`, there is a division by `10000`.
- In `StrikeSelection`, several inlined hardcoded numbers can be found, such as the initial value of prevDelta and the value by which the annualized volatility is multiplied.

To improve the code's readability and facilitate refactoring, consider defining a constant for every magic number, giving it a clear and self-explanatory name. For complex values, consider adding an inline comment explaining how they were calculated or why they were chosen.

Update: *Partially fixed in commit `c9ddbbd86d4434f1e7fdc0f760dff6f9d87e6322` in PR#93. The literal "value by which the annualized volatility is multiplied", as discussed in the last bullet, was not addressed.*

[L08] Mismatched lower bounds for vault fees

When a round is profitable, vaults assess two fees – a performance fee and a management fee.

The `setPerformanceFee` function allows `performanceFee` to be set to zero. However, the `verifyConstructorParams` function, which is called in `baseInitialize` for the vault, requires a non-zero value for `performanceFee`. This discrepancy, coupled with a lack of inline documentation on the matter, can lead to confusion.

Consider having a consistent permissible range for `performanceFee`. Further, to clarify intent and improve the overall readability of the codebase, consider documenting the reasoning behind the upper and lower bounds for all configurable values.

Update: *Fixed in commit `103568180c726dcc3dc5f3bf0f683c871d74132b` of PR#95.*

[L09] Inconsistently formatted, unhelpful, or missing revert messages

Many of the error messages in `require` statements throughout the codebase were found to be too generic, not accurately notifying users of the actual failing condition causing the transaction to revert. Additionally, error messages throughout the codebase were found to be inconsistently formatted.

For instance, the `require` statements throughout `GnosisAuction` contain entire equations along with their error messages, but elsewhere the `require` error messages, like these in `ShareMath` and this in `RibbonVault`, are very short and non-descriptive.

The `require` statements on line 129 and line 149 of `RibbonThetaVault` both check that a premium discount is within the correct range, but provide different error messages.

Finally, the `require` statement on line 123 of `StrikeSelection` does not provide any error message at all.

Error messages are intended to notify users about failing conditions, so they should provide enough information so that appropriate corrections can be made to interact with the system. Uninformative error messages greatly damage the overall user experience, thus lowering the system's quality. Therefore, consider reviewing the entire codebase to make sure every `require` has an error message that is consistently formatted, informative, and user-friendly.

Update: *Partially fixed in commit `b52498add9d7f4c467ec5ca9d4288568010e26b2` of PR#96. Although the specific instances we flagged above were addressed, the codebase is still inconsistent. Some error message are helpful and others are single word error messages prefixed with an exclamation point.*

[L10] Naming issues hinder code understanding and readability

To favor explicitness and readability, several parts of the contracts may benefit from better naming. We noted the following general issues:

- The terminology around the `asset` in the vault is not as clear as it should be. It is sometimes referred to as `collateral` and other times implicitly referred to as `underlying`. Since vaults have an `underlying` parameter – this imprecise naming can lead to unnecessary confusion. Consider being precise and consistent when referencing tokens in the system either explicitly or implicitly.

- The library name `SupportsNonCompliantERC20` is misleading. In fact, it only supports a single non-compliant ERC20, which is `USDT`. There are other tokens that have the same "non-compliant" approval behavior and this library would not support any of them. (CRV for instance.) If the contract is only meant to handle USDT, consider renaming it. Alternatively, if it may handle additional non-compliant tokens in the future, consider adding inline comments signaling this intention.

Additionally, we recommend the following, more specific, suggestions related to naming:

- In `IOptionsPremiumPricer`, the arguments to the `getOptionDelta` event, `sp`, `st`, `v`, and `t` should be renamed to `underlyingPrice`, `strikePrice`, `volatility`, and `time` respectively.
- In the `SupportsNonCompliantERC20` library, the `safeApprove` method should be renamed to `safeApproveNonCompliant` or something similar, so that it does not share the same name as the `safeApprove` method in the `SafeERC20` library used throughout the codebase.
- In the `RibbonVault` contract, the `topup` variable should be renamed to `doCombineSameRound` or similar.
- In the `ShareMath` library:
- `pps` should be `pricePerShare` or `assetPerShare`.
- `sharesToUnderlying` should be `sharesToAsset`.
- `underlyingToShares` should be `assetToShares`.
- In the `VaultLifecycle` library:
- `currentSupply` should be renamed to `currentShareSupply`.
- `verifyConstructorParams` should be renamed to `verifyInitializerParams`.
- `OTOKEN_DECIMALS` should be `OTOKEN_MULTIPLER`. Alternatively, if it is not renamed, then its value should be changed to `8` instead of `10**8`, and any logic that depends on it should be updated appropriately.
- In the `OptionsVaultStorage` contract, `lastStrikeOverride` should be `lastStrikeOverrideRound`.
- In the `Vault` contract, `minimumSupply` should be `minimumInitialSupply`.
- In the `RibbonThetaVault` contract, `unlockedAssedAmount` should be `unlockedAssetAmount`.
- In the `RibbonDeltaVault` contract, `options` here and here should be `option`.
- In the `GnosisAuction` contract, `auctionCounter` should be `auctionID`.

Consider renaming any potentially confusing or misleading parts of the codebase to increase overall code clarity.

Update: *Partially fixed in commit `29d4de9b8bd3cbb7a8f34650daf20fb01be06e43` of PR#97.*

[L11] Identical constants defined independently

The constant `PLACEHOLDER_UINT` is defined in `ShareMath` and again in `RibbonVault`. In both cases the constant is set to the same value, `1`. However, having no single source of truth for identical constants is error prone, especially as the project is iterated on and refactored.

Since every contract that uses `PLACEHOLDER_UINT` also imports `ShareMath`, consider using `ShareMath.PLACEHOLDER_UINT` instead of redefining `PLACEHOLDER_UINT` in `RibbonVault`. Alternatively, consider setting the locally defined `PLACEHOLDER_UINT` equal to `ShareMath.PLACEHOLDER_UINT` directly, so that if the latter is updated the former will be as well.

Update: *Fixed in commit `a869738765a4c40b6f8bf5b03703bb255ba4da7f` of PR#98.*

[L12] Solidity version varies, is not the most recent

The version of Solidity used throughout the codebase varies, but it is not pinned, nor is it the latest stable version.

The choice of Solidity version should always be informed by the features each version introduces that the codebase could benefit from, as well as the list of known bugs associated with each version.

As we mentioned in issue L01, a benefit of using the latest version of Solidity (0.8.7 at time of writing) is that it provides checked arithmetic operations by default, making the use of the `SafeMath` and `DSMath` libraries, and all of the attendant casting, unnecessary in many cases.

Consider taking advantage of the latest Solidity version to improve the overall readability of the codebase and to help reduce the complexity discussed in issue L01. Regardless of which version of Solidity is used, consider pinning the version consistently throughout the codebase to prevent the introduction of bugs due to incompatible future releases.

Update: *Fixed in commit `343e681d8bbf89981062cb5a0cbe7239b655f2d2` of PR#101.*

[L13] Unnecessary max redemption of shares in some cases

The `withdrawInstantly` function in the `RibbonDeltaVault` contract allows a user to instantly withdraw `asset` from the vault. This is accomplished via the withdrawal of newly deposited asset first, and then the redemption of shares that are either held by the user or held by the vault, in that order.

However, as currently implemented, in any case where even a single share must be withdrawn from the vault, the `_redeem` function is called with the `isMax` boolean argument set to true. This behavior could lead to confusion and the unnecessary withdrawal of collateral from the vault.

To better clarify intent and to align the code with expectations, consider only withdrawing the minimum required number of shares to satisfy the instant withdrawal request.

Update: *Not fixed.*

[L14] Casting between types without overflow checks

Throughout the codebase, there are instances of larger `uint` types being cast to smaller `uint` types without overflow checks. For example:

- In both the `RibbonThetaVault` and the `RibbonDeltaVault`, the sum of `delay` and `block.timestamp` is cast to a `uint32`. Even with a small `delay`, in approximately 80 years this cast will overflow.
- In `RibbonThetaVault` the `uint256` `lockedBalance` is cast to a `uint104` without checking if it can fit.
- In `RibbonVault`, the `unit256` `balanceOf` an IERC20 `asset` is cast it to a `uint104` without checking that it can fit.
- In `RibbonVault`, the `uint128` `vaultState.totalPending` is cast to a `uint256`. Then the `uint256` `amount`, is added to it. The sum is cast to a `uint128` without checking that it can fit.
- In `RibbonVault`, the `uint128` `vaultState.queuedWithdrawShares` is cast to a `uint256`. Then the `uint128` `shares`, is added to it. The sum is cast to a `uint128` without checking that it can fit.

To ensure that type casts cannot corrupt values and lead to undesirable system behavior, consider using the OpenZeppelin `SafeCast` library for casting operations where possible.

Update: *Fixed in commit `ca15ff53c62d9f86de417e1060ba778939fe6892` of PR#103. Although the Ribbon Finance team did not use the recommended OpenZeppelin library, they did add overflow checks for the instances identified above.*

[L15] Unused code

Throughout the codebase, there are cases of unused code. For example:

- None of the inlined `DSMath` functions in VaultLifecycle are used.
- The `InitiateGnosisAuction` event is not used in `RibbonThetaVault`.
- The `PlaceAuctionBid` event is not used in `RibbonDeltaVault`.

To improve the readability of the codebase and limit the potential attack surface, consider always removing unnecessary lines of code.

Update: *Fixed in commit `e1ec5868835ee2ed718668695157bea11f209553` of PR#90. The Ribbon team explained that, as the `InitiateGnosisAuction` and `PlaceAuctionBid` events are emitted from libraries, they would not be part of the vaults' ABIs, so they included these events in the vault contracts in order to facilitate their detection and decoding.*

[L16] Error-prone variable shadowing

There are instances in the codebase where local variables are declared with the same name as existing declarations. For instance:

- The `shares` function is shadowed by the `shares` argument in the `initiateWithdraw`, `redeem`, and `_redeem` functions.
- The `decimals` function is shadowed by the `decimals` variable in the `accountVaultBalance` function.

Variable shadowing can make the codebase harder to read and variables harder to reason about. It can also prevent the compiler from raising warnings when local variables are refactored incompletely, leading to unexpected errors that can be difficult to debug.

To increase the overall readability and maintainability of the codebase, consider eliminating instances of variable shadowing by renaming the local variables where possible.

Update: *Fixed in commit `df70e607c4f53afe03ac97d070d5b32bfb768fbb` of PR#105.*

Notes & Additional Information

[N01] Broken links in inline documentation

Throughout the codebase, various links present in the inline documentation are broken. For example:

- These links in `RibbonVault`
- This link in `VaultLifecycle`

To make the codebase easier to understand and maintain, consider reviewing the links in the codebase and fixing any which are broken.

Update: *Fixed in commit `6edda917d1e019ba86a4fd5854ba96abaa1aab42` of PR#99.*

[N02] Implicit access control has drawbacks

Most administrative functions for the vaults use the `onlyOwner` modifier so that only the `owner` address can execute them. However, the `rollToNextOption` function of `RibbonThetaVault` does not use this modifier, but instead implicitly relies on the nested call to `startAuction` for access control. While this effectively prevents accounts other than the `owner` from calling `rollToNextOption`, it does so only after burning non-negligible amounts of gas and is less readable than an explicit access control declaration.

In order to avoid unnecessary gas consumption in some cases and to increase the legibility of the codebase, consider having all functions explicitly define their access control mechanisms.

Update: *Acknowledged. The Ribbon Finance team has said that this is addressed with their new* `keeper` *role and* `onlyKeeper` *modifiers, though those additions and modifications have not been audited by OpenZeppelin at this time.*

[N03] Incomplete `enum` definition

The `ActionType enum` is missing `Liquidate`, which is present in Gamma Protocol's Actions library.

To facilitate future iterations and minimize potential confusion, consider using the complete `enum` definition from Opyn's Gamma Protocol.

Update: *Fixed in commit* `9f1d067fa726adf8fe4c4bc61589429acd7794d0` *of PR#100*.

[N04] Inconsistent use of safe math libraries

The codebase uses two separate math libraries, namely, `DSMath` and `SafeMath`, to perform common math operations in a safe manner. The former library, despite being partially duplicated throughout the codebase, is used relatively sparsely.

While not inherently problematic to use two libraries that offer essentially the same functionality, it can lead to unnecessary confusion. For instance, in the `StrikeSelection` contract, `DSMath` is inherited from, but only its `sub` method is used. In the same contract, the `SafeMath` library is imported and used for `uint256` values. Then *its* `sub` method is used for half the `sub` calls in the contract. Without some associated documentation, such an implementation risks introducing unnecessary confusion and increasing the size of the codebase with no clear benefit.

In another case, `RibbonDeltaVault`](https://github.com/ribbon-finance/ribbon-v2/blob/3fa3bec15ad1e2b18ad87f979b87a68368497f13/contracts/vaults/base/RibbonVault.sol) inherits from `DSMath`, but only the `min` method is used. Since OpenZeppelin contracts are being imported as a dependency already, the OpenZeppelin `Math` library could be used to provide this `min` functionality.

To simplify the codebase, reduce bytecode size, and increase readability, consider using a single vendor for safe math functionalities and inheriting desired functionality from a single library where possible.

Update: *Fixed in commit* `810de79b6ea2f81e5c0d146d219fa51284e49c2a` *of PR#102*.

[N05] Inconsistent coding style

An inconsistent coding style can be found throughout the codebase. Some examples include:

- Constants not using UPPER_CASE format
- Inconsistent prefixes used for parameter naming within functions
- Inconsistent use of the `Pct` suffix to denote that a variable represents a percentage. This is inconsistent with other variables that represent percentages but do not use said suffix.
- Inconsistent grouping of immutable variables. Some contracts group all immutable variables together while others do

not.

The project includes commands to run a linter, but when running them they report several linting issues.

To improve the overall consistency and readability of the codebase, consider adhering to a more consistent coding style by following a specific style guide and by fixing issues reported by the linter.

Update: *Partially fixed in commit `b869e49b2daf2c860db10efb5fb33967ac42bf89` of PR#104.*

[N06] Inconsistent use of named return variables

There is an inconsistent use of named return variables across the codebase.

For instance, in the `VaultLifecycle` library alone, some functions return named variables, some return explicit values, and others declare a named return variable but override it with an explicit return statement.

Similar inconsistencies can be found throughout the contracts, interfaces, and libraries that comprise the project.

Consider adopting a consistent approach to return values by removing all named return variables, explicitly declaring them as local variables, and adding the necessary return statements where appropriate. This would improve both the explicitness and readability of the code, and it may also help reduce regressions during future code refactors.

Update: *Partially fixed in commit `7eefb3e45d2c54411e00614f6230d38ff0f99c13` of PR#107. The new style, though consistently adopted, removes named returns from functions with a single return value, but uses both named returns and explicit returns with functions that return more than a single value. This can be an error prone style. The compiler will not flag functions that do not explicitly return values when named returns are used.*

[N07] Lack of explicit visibility in constants

Throughout the codebase there are constants that are implicitly using the default visibility. For example:

- The `USDT` constant in `SupportsNonCompliantERC20`
- The `DSWAD` constant in `GnosisAuction`
- The `DSWAD` constant in `VaultLifecycle`

To clarify intent and favor readability, consider explicitly declaring the visibility of all constants and state variables.

Update: *Fixed in commit `36b9f33db90a3eb357eace7c4fd69cad6c2980b6` of PR#108.*

[N08] Unnecessary complexity for `managementFee` calculation

When setting the `public` state variable `managementFee` in `RibbonVault`, both during initialization and in the `setManagementFee` function, the input parameter corresponds to the *yearly* management fee whereas the state variable stores the *weekly* management fee.

In both cases, the input parameter must be immediately divided by the `WEEKS_PER_YEAR` constant. Because the number of weeks per year differs between leap years and non-leap years, this division can introduce some imprecision. Additionally, having the relevant setter and getter functions dealing in different terms can be confusing.

The similar variable, `performanceFee`, is handled consistently in weekly terms. This makes the inconsistent handling of `managementFee` even more confusing and potentially error prone.

In order to make the effective `managementFee` more precise, make its handling consistent with `performanceFee`, and reduce the complexity of the code, consider passing the weekly management fee to the contract rather than the yearly management fee.

Update: *Not fixed. The Ribbon Finance team state, "we prefer leaving as is".*

[N09] Loss of accuracy as a result of divisions

Due to the fact that division truncates in the EVM, division should be done last unless overflows make such an order infeasible. While divisions are generally performed last throughout the codebase, on lines 128-132 of `GnosisAuction`, there are a series of `SafeMath` operations performed to calculate a `buyAmount` where the division happens in the middle of the calculation, before a multiplication. This can lead to the amplification of the truncation and further loss of precision.

Consider changing the order of operations so that the multiplications are done before divisions where possible.

Update: *Fixed in commit `9d036480c0850648b681847f421361dc6481f904` of PR#109.*

[N10] Not inheriting from available interfaces

Some of the provided interfaces are not inherited from by the contracts they are meant to describe. This can lead to issues if an interface or corresponding contract is modified in a way that would make them incompatible. Some examples of this lack of inheritance are:

- `StrikeSelection` does not inherit from `IStrikeSelection`.
- `RibbonThetaVault` does not inherit from `IRibbonThetaVault`.

To clarify intent, increase the readability the codebase, and allow the compiler to perform more robust error-checking, consider updating the contracts' inheritance declarations to explicitly inherit from their corresponding interfaces.

Update: *Not fixed. The Ribbon Finance team state, "we prefer leaving as is".*

[N11] Redundant `require` statements

The `ShareMath` library includes two helper functions, `assertUint104` and `assertUint128`, to check that a provided `uint` can fit inside `104` and `128` bits, respectively. These are just wrappers around `require` statements, and they are used extensively throughout the codebase.

In the `getSharesFromReceipt` function, the `sharesFromRound` value is set as the return value of `underlyingToShares`. However, `underlyingToShares` asserts that its calculated return value can fit inside a `uint104` before returning that value as a `uint104`. This makes the assertion that `sharesFromRound` is a `uint104` redundant.

Consider removing redundant `require` statements and assertions to reduce the overall complexity and gas consumption of the codebase.

Update: *Fixed in commit `78448cb3ab4626cc68f753141a5354c5579a9361` of PR#110.*

[N12] Gas-saving conditional too strict

In the `shareBalances` function of `RibbonVault`, there is a conditional statement to short-circuit the function and return `0` for `unredeemedShares` in cases where there can be no `unredeemedShares` to count. However, the conditional that is used is too narrow to capture all applicable conditions. Because a vault in its first round cannot have any `unredeemedShares`, the conditional should also short-circuit the function in the case of equality.

To further clarify the intent of the function and to save additional gas, consider modifying the conditional to be `<=` rather than strictly `<`.

Update: *Not an issue. It is possible for a vault to have unredeemed shares at round `PLACEHOLDER_UINT` so the equality from the audited codebase should not be modified.*

[N13] Undocumented implicit approval requirements

The `RibbonVault` contract's `deposit` and `depositFor` functions implicitly assume that they have been granted an appropriate allowance before calling `safeTransferFrom`.

In favor of explicitness and to improve the overall clarity of the codebase, consider documenting all approval requirements in the relevant functions' inline documentation.

Update: *Fixed in commit `f9fe0c7deee8bd4c05645ec5f7e8b5662d4730a3` of PR#112*.

[N14] Unused import statements

Within the codebase there are instances of files being imported unnecessarily. For example:

- `SafeERC20` and `IOptionsPremiumPricer` in `VaultLifecycle`
- `GnosisAuction`, `IOtoken`, `IGnosisAuction`, `IStrikeSelection`, and `IOptionsPremiumPricer` in `RibbonVault`
- `SafeERC20` and `IGnosisAuction` in `RibbonDeltaVault`
- `IOtoken` in `StrikeSelection`

To improve the overall legibility and maintainability of the codebase, consider removing any unused import statements.

Update: *Partially fixed in commit `35fd1001b4c0b4b76b47e1a88dac8f6c2606b6f1` in PR#113. `SafeERC20` is still imported in `RibbonDeltaVault` but not used.*

Conclusions

1 critical and 1 high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface.

RELATED POSTS

©2021. All rights reserved | Privacy | Terms of Service