Notional Audit

DECEMBER 22, 2020 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

Introduction

The Notional team asked us to review and audit their smart contracts after they had iterated their system design taking into consideration our first audit. We looked at the code and now publish our results.

We audited commit `b6fc6be4622422d0e34c90e77f2ec9da18596b8c` of the notional-finance/contracts repository. In scope are all of the contracts inside the `/contracts` directory, aside from `MockLiquidation.sol`.

About Notional

As specified in the first audit, Notional is a protocol enabling fixed-term, fixed-rate lending and borrowing on the Ethereum blockchain through a novel financial primitive named `fCash`, which provides a mechanism for users to commit to transfers of value at a specific point in the future. `fCash` tokens can represent either a claim on a positive or negative cash flow, and are always generated in pairs of negative and positive tokens which always net to zero across the protocol. Positive `fCash` balances represent an amount of a specific

currency type at a certain maturity, while negative `fCash` balances represent an obligation for the holder to provide an amount of a specific currency type at a certain maturity.

Actors

There are three main actors that interact with the system:

- **Lenders**: Users who lend a listed currency at a fixed interest rate, in exchange of `fCash`. After the maturity is reached, the lender is able to redeem the `fCash` in exchange of a greater fixed amount of the currency they initially deposited.
- **Borrowers**: Users who want to borrow cash from the system at a fixed interest rate. Borrowers deposit collateral into the Notional system in order to be able to mint a pair of negative and positive `fCash` of the specific currency they want to borrow at a specific maturity. When maturity is reached, the borrower can repay the amount of currency they owe, or let the system use the collateral to cover the debt.
- **Liquidity Providers**: Users in charge of providing liquidity to the system. They deposit an amount of a specific currency and its corresponding `fCash` into liquidity pools in exchange of Liquidity Tokens. Liquidity Providers can redeem their deposit at any time, and earn fees on each trade performed by lenders and borrowers in the liquidity pool in which they are participating.

Privileged Roles

Since the first audit, no changes were performed regarding governance management. The Notional team will initially administer (and then eventually transition to a decentralized community-led administration of) many aspects of the protocol, such as:

- Decide which currencies will be listed in the protocol
- Create and remove `fCash` markets
- Change governance parameters throughout the system, including liquidity and transaction fees
- Introduce functionality changes by performing upgrades to the core contracts of the system

Overall Health

We found the code to be in a much better state compared to the first audit we performed. We valued that the Notional docs had a general description of the

protocol's intended functionality, which made the process of understanding and auditing the code easier. While auditing the project, we identified some issues that stemmed from the architecture of the project, issues around the usage of the unstructured storage upgradeability pattern, and issues pertaining to the overall consistency of the codebase.

Ecosystem Dependencies

While the dependency on, and the role of, both Wrapped Ether (WETH) and Chainlink oracles has not changed from the first audit, Notional has modified their system design so that it no longer depends on UniswapV2. The liquidation dynamics have now been largely internalized. As in the first audit, in this round we assume that the WETH and Chainlink oracle protocols work as intended.

The removal of a secondary price source via UniswapV2 has made the system more reliant on Chainlink oracles. Price manipulation is still technically possible and could allow a manipulator to steal money from the protocol. This is a commonly accepted risk with DeFi protocols.

Additional Information

Throughout the course of the audit, Notional informed us that they had found a couple of issues in the codebase, specifically:

- The liquidation process forced liquidators to buy the full amount of collateral currency in order to return a severely under-collateralized account to a healthy collateralization level. This obligated liquidators to have enough capital to liquidate the account in one transaction. This issue was fixed by adding a `maxToLiquidate` parameter to allow liquidators to partially capitalize the liquidation. This was fixed in commit `d0035b4b6b96703b6a535af5a76f0c4df3f84e32`.
- In those cases where there were matured cash payer assets in the portfolio, matured assets were not being settled before attempting to settle the reserve. This issue was fixed by calling the `settleMaturedAssets` function before settling the reserve, in commit `8699c9111892b561f28457fce4141e7d81646ddc`.
- For users with a negative cash balance, it was not possible for them to perform a borrow through a batch operation even if they were collateralized. This was fixed in commits `b2d49de58d3f75da7f84949dd91b4c02315304d1` and `23ccaba53af784d4925eb832cc868e3f02e7b6b9`.

Update: *All of the following issues have been addressed or acknowledged by the Notional Team. We are in the process of reviewing the fixes and will update this report when that is completed.*

Critical

None.

High

None.

Medium

[M01] `contracts` addresses in `Governed` and `Directory` can get out of sync

The `Directory` contract facilitates storing the addresses and the dependency map of the core contracts of the system in the `contracts` data structure, which includes the `Escrow`, `Portfolios` `ERC1155Token`, and `ERC1155Trade` contracts. It also provides functionality for the owner to set these contract addresses in the forementioned core contracts through the `_setDependencies` function defined in the `Governed` contract, which accesses the `Directory` contract to get relevant addresses as needed.

There are scenarios where the distinct `contracts` data structures in the `Governed` and `Directory` contracts can get out of sync, which can cause several inconsistencies in the behavior of the system. This can happen in the following scenarios:

- When calling the `setContract` function of the `Governed` contract, which is called by the `setDependencies` function of the `Directory` contract, it will update the `contracts` variable in the `Governed` contract but not the `contracts` variable in the `Directory` contract. Note that this is inconsistent with, for instance, the behavior of the `_setDependencies` function called by the `CashMarket` `initializeDependencies` function.
- When calling the `setContract` function of the `Directory` contract, which does not update the `Governed` contract `contracts` variable, but instead relies on calling the `setDependencies` function of that same contract afterwards.

Consider modifying the system so that there is only a single way for the `contracts` data structure of the `Directory` contract to be set. Additionally, consider enforcing that the `contracts` data structure in the `Governed` contract is appropriately updated when this happens, to avoid desynchronization between the different `contracts` data structures.

[M02] Adding new variables to multi-level inherited upgradeable contracts may break storage layout

The Notional protocol uses the OpenZeppelin/SDK contracts to manage upgradeability in the system, which follows the unstructured storage pattern. When using this upgradeability approach, and when working with multi-level inheritance, if a new variable is introduced in a parent contract, that addition can potentially overwrite the beginning of the storage layout of the child contract, causing critical misbehaviors in the system.

It has to be noted that this same issue can arise from adding new variables to any other external contract used in the inheritance chain, such as the `Ownable` contract in the `upgradeable` folder.

For custom contracts, consider preventing these scenarios by defining a storage gap in each upgradeable parent contract at the end of all the storage variable definitions as follows:

```
uint256[50] __gap; // gap to reserve storage in the contract for future variable additions
```

In such an implementation, the size of the gap would be intentionally decreased each time a new variable was introduced, thereby avoiding overwriting preexisting storage values.

Additionally, instead of using contracts copied from the `OpenZeppelin/contracts` such as `Ownable`, consider using the `openzeppelin-contracts-upgradeable` package which already defines the forementioned gap. Using said package would also enable the system to benefit from any future changes implemented in this and any other contracts provided by the OpenZeppelin team.

[M03] Lack of event emission after sensitive actions

The following functions do not emit relevant events after sensitive actions.

- The `initialize` function of the `Governed` contract should emit a `DirectorySet` event.
- The `setContract` function of the `Governed` contract should emit a `SetContract` event, different from the one in the `Directory` contract.
- The `_setDependencies` function of the `Governed` contract should emit a `SetContract` event.
- The `setParameter` function of `CashMarket.sol` should emit a `ParametersSet` event, showing the new values of `G_MATURITY_LENGTH` and `G_NUM_MATURITIES`, or emit an individual event for each variable updated (e.g.: `MaturityLengthSet`, `NumMaturitiesSet`), showing the old and new values for each of them.
- The `setDependencies` function of the `Directory` contract should emit a `SetContract` event, as is being done in the `setContract` function of the same contract.
- In `Portfolios.sol`, when calling `settleMaturedAssets` a `SettleAccount` event is emitted if the account has any assets that were settled. However, when calling `settleMaturedAssetsBatch` no `SettleAccount` events are emitted. If the function is not `calledByEscrow`, then the `SettleAccountBatch` event is emitted, but it simply lists all accounts that were provided in the batch – with no way to distinguish which accounts were actually settled.

Consider emitting events after state-changing sensitive actions take place, to facilitate tracking and notify off-chain clients following the contracts' activity.

[M04] Contracts storage layout can get corrupted on upgradeable contracts

The Notional protocol uses a copy of some of the OpenZeppelin/SDK upgradeable contracts to manage upgrades in the system, which follows the unstructured proxy pattern.

This upgradeability system consists of a proxy contract which users interact with directly and that is in charge of forwarding transactions to and from a second contract. This second contract contains the logic, commonly known as the implementation contract.
When using this particular upgradeability pattern, it is important to take into account any potential changes in the storage layout of a contract, as there can be storage collisions between different versions of the same implementation.
Some possible scenarios are:

- When changing the order of the variables in the contract

- When removing the non-latest variable defined in the contract
- When changing the type of a variable
- When introducing a new variable before any existing one
- In some cases, when adding a new field to a struct in the contract

There is no certainty that the storage layout will remain safe after an upgrade. Violating any of these storage layout restrictions will cause the upgraded version of the contract to have its storage values mixed up, and can lead to critical errors in the contracts.

Consider checking whether there were changes in the storage layout before upgrading a contract by saving the storage layout of the implementation contract's previous version and comparing it with the storage layout of the new one. Additionally, consider using the openzeppelin/upgrades plugins which already cover some of these scenarios.

[M05] Invalid transaction fee encoding specifications

The `setFee` function that begins on line 150 of `CashMarket.sol` sets the liquidity and transaction fee rates for the market in which the function is called. In this context, the transaction fee is the percentage of a transaction that is taken by the protocol and moved to a designated reserve account. As the name suggests, transaction fees factor in to many of the essential transaction types performed within the system.

The encoding scheme information in the `setFee` function's NatSpec `@notice` tag specifies that a value of one percent should be encoded as `1.01e18`, but this leads to reversions in the `_takeCurrentCash` and `_takefCash` functions upon which the system depends. Additionally, none of the other documented encoding formats for similar values in the codebase can be used for the relevant `transactionFee` value.

The encoding scheme used for other fee-like values in the system, such as those used by the `setHaircuts` function, encode a value of one percent as `.99e18`. If applied here, that also leads to reversions in the same essential functions.

The unit tests dealing with `transactionFee` values encode them as basis points, so that one percent would be represented as `1e7` wherever `INSTRUMENT_PRECISION` values are `1e9`. While using this encoding scheme doesn't result in reversions, it does miscalculate the `fee` and any values dependent on it, including implied rates.

Since the system does not currently collect fees, this issue is not manifesting itself. To avoid potential issues going forward, consider adding validation logic to ensure that values supplied for `transactionFee` will yield the expected results. Also consider documenting the proper encoding format to use when supplying the relevant fee values and unit testing to verify that all fee calculations align with expectations.

Low

[L01] Anyone can call the `initializeDependencies` function in `CashMarket.sol`

The `initializeDependencies` function in the `CashMarket` contract initializes the addresses of the core contracts of the system in the `Governed` contracts data structure, so that the forementioned contract can interact with them within its functions.

Since this function is not restricted and can be called by anyone, it is possible that a malicious actor could monitor the mempool, waiting for a core contract address in the `Directory` contract to be modified. If that new core contract were to introduce any inconsistencies in the interactions between itself and the previously deployed `CashMarket` contract, the attacker could call `initializeDependencies` to force `CashMarket` to use the modified core contract. This could result in undesirable behavior in `CashMarket` functions that interact with the core contracts of the system.

Consider restricting the `initializeDependencies` function so that it can only be called by the owner of the contract.

Update: *Fixed in PR #6.*

[L02] Incomplete parameters in emitted events

Some events in the codebase do not show all relevant parameters when being emitted. *Some examples* are:

- The `UpdateRateFactors` event in `CashMarket.sol` should also emit the old values that are being overwritten.
- The `UpdateMaxTradeSize` event in `CashMarket.sol` should also emit the old `maxTradeSize` value.
- The `UpdateFees` event in `CashMarket.sol` should also emit the old values of the `liquidityFee` and `transactionFee` variables.

- The `SetContract` event in `Directory.sol` should also emit the old value of the contract address that is being updated.
- The `setReserveAccount` event in `Escrow.sol` should also emit the previous value of the `G_RESERVE_ACCOUNT` variable.
- The `SetMaxAssets` event in `Portfolios.sol` should emit the previous value of the `maxAssets` variable.
- The `SetHaircut` event in `Portfolios.sol` should emit the previous values of the `liquidityHaircut`, `fCashHaircut`, and `fCashMaxHaircut` variables.

When modifying a state variable in the system, consider emitting both its old and new value to notify off-chain clients monitoring the contracts' activity.

[L03] Inconsistent and incomplete core contracts initialization

There are some inconsistencies around how the `core contracts` of the system are initialized. For instance, the `CashMarket` contract implements the `initializeDependencies` function, which allows the owner of the contract to set the addresses for all of the core contracts upon which it depends. This is inconsistent with the `Escrow`, `ERC1155Token`, `ERC1155Trade`, and `Portfolios` contracts, where the `setDependencies` function defined in the `Directory` contract must be used to perform the equivalent function.

Moreover, there are several variables in these contracts that could be set on initialization rather than exclusively by means of an independent function. *Some examples* are:

- All the variables set in the `setParameters` function of the `CashMarket` contract.
- All the variables set in the `setHaircuts` function of the `Portfolios contract`.

To avoid confusion and to improve the overall readability and consistency of the code, consider setting these dependency addresses in each core contract from their respective initialization functions. Additionally, consider initializing all other relevant variables in those initialization functions rather than in a separate function. Lastly, consider emitting all the relevant events as mentioned in *Lack of event emission after sensitive actions* when initializing storage.

[L04] Lack of input validation

There are several instances of `external` functions failing to validate the input parameters they are provided. For example:

- In the `setParameters` function on line 94 of `CashMarket.sol` and the `createCashGroup` function on line 140 of `Portfolios.sol`, `maturityLength` can be set arbitrarily. In practice, a market with an extremely large maturity length would likely not have many participants. Even so, if `maturityLength` were too large, it would lead to erroneous cash ladders. If `maturityLength` were set to zero, it would lead to reversions caused by division by zero.
- In the `setFee` function on line 150 of `CashMarket.sol`, neither `liquidityFee` nor `transactionFee` are given upper bounds. Values that are too large will lead to reversions in several critical functions.
- In the `setParameters` function on line 94 of `CashMarket.sol`, `numMaturities` can be set to `0` which would cause reversions in several critical functions.
- In the `settleCashBalanceBatch` function on line 682 of `Escrow.sol`, the length of `values` and `payers` is not required to be equal. Unequal lengths will lead to a reversion after potentially burning non-negligible amounts of gas.
- In the `setHaircuts` function on line 100 of `Portfolios.sol`, the values passed in for the various "haircuts" can be arbitrarily large. This is in contradiction with the intention of the codebase and the comment provided in the NatSpec `@notice` tag of this same function.
- In the `updateCashGroup` function on line 182 of `Portfolios.sol`, the NatSpec comments list several guidelines for each input, but none of those guidelines are enforced in the code.

To avoid errors and unexpected system behavior, consider explicitly restricting the range of inputs that can be accepted for all externally-provided inputs via `require` clauses where appropriate.

[L05] Missing error messages in require statements

Throughout the codebase, there are several require statements which lack error messages. For example:

- On line 786 of `Escrow.sol`.
- On line 237 of `Liquidation.sol`.
- On line 478 of `Liquidation.sol`.
- On line 536 of `Liquidation.sol`.

- On line 540 of `Liquidation.sol`.
- On line 610 of `Liquidation.sol`.
- On line 162 of `Portfolios.sol`.
- On line 815 of `Portfolios.sol`.
- On line 176 of `RiskFramework.sol`.

Consider providing specific, informative, and user-friendly error messages with every require statement.

[L06] Multiple conditions in a single require statement

There are instances in the codebase where a single `require` statement contains multiple conditions. *Some examples* are:

- On line 123 in `CashMarket.sol` within the `setRateFactors` function.
- On line 205 in `Escrow.sol` within the `listCurrency` function.
- On line 509 in `Portfolios.sol` within the `mintfCashPair` function.

Consider isolating each condition in its own `require` statement where possible, so as to be able to include a more specific user-friendly error message for each required condition.

[L07] Not using SafeMath functions

Although most of the codebase employs SafeMath methods where appropriate, there are still a few instances of regular Solidity arithmetic operators being used. *Some* examples are:

- On line 81 of `RiskFramework.sol` `*` is used.
- On line 207 of `Escrow.sol` `++` is used.

These instances are not protected from potential overflows and may return unexpected values that could lead to data in `storage` being unintentionally overwritten. Consider always performing arithmetic operations with methods that protect the system from such possibilities, like the math libraries of OpenZeppelin contracts.

[L08] Setting ownerships directly rather than via API

The `Directory` and `Governed` contracts define an `initialize` function to initialize, among other things, the `owner` variable defined in the `Ownable` contract. However, this variable is being initialized manually rather than by using the `Ownable` contract's API, and therefore the `OwnershipTransferred` event is not emitted.

Since in **Adding new variables to multi-level inherited upgradeable contracts may break storage layout** it was recommended to use the `OwnableUpgradeSafe` contract present in the `openzeppelin-contracts-upgradeable` package, consider using its `__Ownable_init` function to initialize ownership.

[L09] Unbounded array lengths could lead to out of gas errors

In our prior audit, we raised the issue of the Notional system potentially running out of gas within functions that iterated over arrays crucial to the system. We specifically cited the 'portfolios' array of assets as one whose length should be bounded. While the system now limits the length of portfolios by setting a max number of assets, there are still some arrays that could grow too large to be iterated over in some cases.

For example, `maxCurrencyId` essentially has no reasonable upper bound to limit the number of currencies that could be listed. Since `maxCurrencyId` is explicitly used to set the size of several other arrays that are often iterated over, unbounded growth of this value could be problematic. If too many currencies were listed, several functions within the system could potentially fail due to out of gas errors.

To prevent encountering out of gas errors that would be difficult to remedy, consider putting an upper bound on the value of variables that are used to limit array growth, especially when those arrays will be iterated over.

[L10] Casting between types without overflow checks

In our prior audit of the Notional system, we raised an issue about unsafe casting between types. During their initial response to the prior audit, they partially addressed our concerns. The few persistent instances of the issue were to be removed prior to this audit.

However, the codebase is not yet entirely free of this issue. There are still a few instances of explicit casts that, in scenarios that may well be unlikely to happen,

could result in an undesirable truncation leading to unexpected values. *Some examples* are:

- Within the functions `_convertToETH` on line 49 of `ExchangeRate.sol` and `_convertETHTo` on line 80 of `ExchangeRate.sol`, the `int256` `balance` input is explicitly cast to a `uint128`.
- Within the `_calculateNotionalToTransfer` function on line 816 of `Portfolios.sol`.

Consider using the `SafeCast` library for the casting operations cited in the examples, and wherever else possible, to ensure those type casts cannot corrupt values and lead to undesirable system behavior.

[L11] Unused `struct`

The `struct` `CollateralCurrencyParameters` declared on line 33 of `Liquidation.sol` is never used elsewhere in the codebase. Consider removing unused code to improve overall legibility.

Notes

[N01] Confusing constant usage

`MAX_CASH_GROUPS` is a `uint8` `constant` defined on line 38 of `PortfoliosStorage.sol`. It is defined in hex format as `0xFE` which is decimal `254`.

Within the `createCashGroup` function, there is a `require` that checks that `currentCashGroupId <= MAX_CASH_GROUPS`. If the condition is satisfied, then `currentCashGroupId` is incremented. This allows for the unintuitive state where `currentCashGroupId` can increment to `255`, which is *greater* than `MAX_CASH_GROUPS`.

Consider altering the `require` condition to check that `currentCashGroupId < MAX_CASH_GROUPS` to more closely align the `constant` name with the implementation. Note that if such a change were implemented, `MAX_CASH_GROUPS` should also be set to `255` to retain the current range of `currentCashGroupId`s.

[N02] `decodeAssetId` is not the inverse of `encodeAssetId`

The function, `encodeAssetId` on line 225 of `Common.sol` encodes four attributes of an asset – `cashGroupId`, `instrumentId`, `maturity`, and

`assetType`. The related decode function, `decodeAssetId` on line 240 of `Common.sol` does not decode the `assetType`. Instead, there is a separate function, `getAssetType` on line 215 of `Common.sol` which decodes the `assetType` attribute. These two internal functions are most often used in immediate succession.

In order to reduce the number of function calls and overall code complexity, consider modifying `decodeAssetId` so that it decodes `assetType` as well.

[N03] Failure is delayed

There are a few instances in the codebase where `external` function calls do not fail quickly. Instead, they invariably fail with certain inputs or under certain conditions, but only after burning non-negligible amounts of gas. For instance:

- In `CashMarket.sol`, the `external` function `addLiquidity` can accept `0` as an input for `cash` or `maxfCash`. In either case, if the market has zero liquidity (`market.totalLiquidity == 0`), then those arguments are sent to the `internal` function `_addLiquidity`. There, `_addLiquidity` calls `_getImpliedRateRequire` which calls `_getImpliedRate` which calls `_getExchangeRate` where the call will revert when `cash` is `0`. If only `maxfCash` is `0`, then `_addLiquidity` calls `_getImpliedRateRequire` which calls `_getImpliedRate` which calls `_getExchangeRate` which then calls `_abdkMath` before the zero value for `maxfCash` leads to an inevitable reversion.
- In `Portfolios.sol`, the `createCashGroup` function and the `updateCashGroup` function both take an argument named `precision`. That value is eventually passed to the `cashMarket` contract's `setParameters` function where `precision` is required to be equal to `1e9`.
- In `Portfolios.sol`, the `raiseCurrentCashViaCashReceiver` function and the `raiseCurrentCashViaLiquidityToken` function both eventually make calls to the `_tradePortfolio` function. Only then is there a `require` present to check that the `calledByEscrow` function returns `true`.

Consider adding relevant `require` statements to the beginning of functions that currently have nested or deferred `require` statements. Checking conditions and failing early where possible can avoid unnecessary gas consumption and can also increase the legibility of the codebase.

[N04] Error prone lack of uniform rate encoding

Within the codebase, there are numerous rates and fees that all require being encoded in different manners. For instance, the `setHaircuts` function specifies "a 5% haircut will be set to 0.95e18" – while the `setDiscounts` function specifies "a 5% discount for liquidators will be set as 1.05e18". There are additional encoding methodologies expected in other parts of the code.

This lack of uniform encoding results in a more error prone system for administrators, users, and developers alike. Consider standardizing encoding methodologies where possible across the various rates and fees the system requires to reduce the likelihood for error.

[N05] Constants lacking explanation

There are several occurrences of literal values with unexplained meaning in the codebase. *Some examples* are:

- Each constant defined on lines 25 through 29 of `CashMarket.sol`.
- The literal `1e9` used on line 109 of `CashMarket.sol`.
- The values for the bit masks and shifts on lines 226 through 229 and lines 245 through 249 of `Common.sol`.

Literal values in the code base unaccompanied by explanation make the code harder to read, understand, and maintain; this negatively impacts the experience of developers, auditors and external contributors alike.

Where possible, consider defining a constant variable for every literal value used, and giving that variable a clear and self-explanatory name. Additionally, for complex values, inline comments explaining how they were calculated or why they were chosen are highly recommended. Following Solidity's style guide, constants should be named in UPPER_CASE_WITH_UNDERSCORES format, and specific public getters should be defined to read each one of them.

[N06] Missing, misleading, or incomplete inline documentation

Although most of the public and sensitive functions in the codebase have relevant docstrings, there are some instances where docstrings are missing, misleading, or incomplete. *Some examples* include:

- The NatSpec `@notice` tag of the `setFee` function on line 150 of `CashMarket.sol` specifies an encoding for `transactionFee` that would lead to reversions.

- The NatSpec `@notice` tag of the `_isValidBlock` function on line 1036 of `CashMarket.sol` states a requirement that `blockTime <= maxTime < maturity <= maxMaturity`. In fact, the code allows for `maxTime` to be greater than `maturity`.
- The `initialize` function on line 21 of `Directory.sol` is missing docstrings.
- The NatSpec `@notice` tag of the `setLiquidityHaircut` function on line 161 of `Escrow.sol` mentions setting a value on the `RiskFramework` contract, but no such action is taken.
- The `initialize` function on line 21 of `Governed.sol` is missing docstrings.
- The `setContract` function on line 33 of `Governed.sol` is missing docstrings.
- The `freeCollateralAggregateOnly` function on line 297 of `Portfolios.sol` is missing docstrings.
- The NatSpec `@dev` tag of the `freeCollateralView` function on line 336 of `Portfolios.sol` requires more context or is misplaced.
- The NatSpec `@return` tag of the `freeCollateralView` function on line 338 of `Portfolios.sol` lists two return values, but in the actual function three values are returned.
- The NatSpec `@notice` tag of the `mintfCashPair` function on line 497 of `Portfolios.sol` mentions "when cashGroup is set to zero" as if it is an allowed condition, but that would contradict the `require` on line 509.

There are also instances of missing or misleading inline comments. *Some examples* include:

- The inline comment on line 66 of `CashMarket.sol` states that only `G_NUM_MATURITIES` is mutable, but, in fact, `G_MATURITY_LENGTH` is as well.
- The `_quickSort` function on line 266 of `Common.sol` is essential for the system to function, but it only has a single inline comment that does very little to explain the function in detail.
- The inline comments on lines 112 through 113 in `RiskFramework.sol` do not hold true for for all `view` functions. This is reaffirmed by other comments on lines 129 through 131.

Clear inline documentation is fundamental to outlining the intentions of the code. Mismatches between it and the implementation can lead to serious misconceptions about how the system is expected to behave. Consider refining the inline documentation that has been identified above as misleading or incomplete. Also consider adding additional inline documentation wherever it is lacking. When writing docstrings, consider following the Ethereum Natural Specification Format (NatSpec).

[N07] Multiple SPDX license identifiers per file

In `IERC777.sol` and `IERC165.sol`, there are multiple SPDX license identifiers declared. This should cause the Solidity compiler to raise an error, but there is currently a compiler bug that results in the second SPDX license identifier being overlooked. Consider having only a single SPDX license identifier per file to avoid future compilation errors and to mitigate potential licensing confusion.

[N08] Naming issues

Throughout the codebase, there are functions and variables that could benefit from better naming. *Some* examples include:

- The `creditBalance` variable defined on line 600 of `Liquidation.sol` should be renamed to `hasCreditBalance` to emphasize that it is a `bool`.
- The `_fCashMaxHaircut` function on line 12 of `PortfoliosStorage.sol` should be renamed to `_getfCashMaxHaircut` to be consistent with its counterpart `_setfCashMaxHaircut`.
- The `_fCashHaircut` function on line 16 of `PortfoliosStorage.sol` should be renamed to `_getfCashHaircut` to be consistent with its counterpart `_setfCashHaircut`.
- The `_liquidityHaircut` function on line 20 of `PortfoliosStorage.sol` should be renamed to `_getLiquidityHaircut` to be consistent with its counterpart `_setLiquidityHaircut`.

There are also several instances of variable names being composed of all capital letters, a solidity convention recommended only for constants. They could benefit from being renamed to follow the solidity style guide. *Some* examples include:

- The `DIRECTORY` variable defined on line 18 of `Governed.sol` should be renamed to `directory`.
- The `G_RESERVE_ACCOUNT` variable defined on line 76 of `EscrowStorage.sol` should be renamed to `gReserveAccount`.
- The `G_MAX_ASSETS` variable defined on line 57 of `PortfoliosStorage.sol` should be renamed to `gMaxAssets`.

Consider making the suggested naming changes to improve overall code legibility.

[N09] Inconsistent argument type for `Portfolios` contract address

Throughout the codebase, functions that accept an argument to refer to the `Portfolios` contract specify that argument type inconsistently. Sometimes the argument type is specified as an `address`, and other times the argument type is specified as `IPortfoliosCallable`. In either case, the `Portfolios` contract address is generally cast between the two types along the call chains of the relevant functions. Consider keeping the `Portfolios` argument type consistent across functions in order to reduce the number of inline casts and improve overall code legibility.

[N 10] Account can potentially be settled without an event emission

Inside the `settlefCash` function on line 910 of `Escrow.sol`, by providing `0` for the input `valueToSettle`, the function will `return` instead of `revert` *after* calling `_freeCollateralFactors`, which in turn calls `freeCollateralFactors` in the `Portfolios` contract. There, `freeCollateralFactors` calls `_settleMaturedAssets` which potentially modifies storage values. This path to execution of `_settleMaturedAssets` will not emit any events, even if state is modified, which could cause off-chain clients monitoring the contract to miss relevant on-chain activity.

Consider using a `require` statement rather than a conditional `return` where possible. Alternatively, in this case, consider executing the relevant `return` statement *before* the call to `_freeCollateralFactors`.

[N11] Typographical errors

The code contains the following typos:

- Throughout the codebase, “a asset” should be “an asset”.
- On line 244 of `Common.sol`, “Instrument Group Id” should be “Cash Group Id”.
- On line 26 of `Escrow.sol`, “a account balances” should be “account balances”.
- On line 26 of `Escrow.sol`, “withdraws” should be “withdrawals”.
- On line 905 of `Escrow.sol`, “are denominated” should be “are denominated in”.
- On line 25 of `ExchangeRate.sol, “True of” should be “True if”.
- On line 47 of `ExchangeRate.sol, “buffer” should be “buffers”.
- On line 47 of `ExchangeRate.sol, “apporpriate” should be “appropriate”.
- On line 106 of `Liquidation.sol`, “token that” should be “token that is”.

- On line 345 of `Liquidation.sol`, “will not longer” should be “will no longer”.
- On line 597 of `Liquidation.sol`, “deterimine” should be “determine”.
- On line 131 of `Portfolios.sol`, “An cash” should be “A cash”.
- On line 134 of `Portfolios.sol`, “maturitys” should be “maturities”.

Consider correcting these typos to improve overall code readability.

[N12] Unnecessary conditionals

There are instances in the code where conditions that are redundant, mutually exclusive, or that could not exist are checked. *Some* examples include:

- On line 182 of Common.sol in the `isCashReceiver` function, the final conditional, `!isLiquidityToken()`, is redundant. If `assetType` were a liquidity token, it would be `false` from the start of the conditional at `isCash()`.
- The `if` on line 163 of `Portfolios.sol` checks for a condition that must necessarily be `true` given the `if` statement that proceeds it.
- The ternary statement on line 779 of `Portfolios.sol` checks for a condition that can never be `true` given the conditions of its parent `for` loop.

Consider removing unnecessary conditional statements where possible to improve code legibility and reduce execution costs.

[N13] Unnecessary sorting of single element arrays

The function `_sortPortfolio` on line 259 of `Common.sol` sorts an in-memory array of `Asset` `struct`s. There is a conditional on line 260 that ensures arrays with a `length` of `0` are not sorted. In fact, an array with only a single element, where `length` is equal to `1`, also does not need to be sorted.

To reduce the number of function calls, reduce gas usage, and more accurately reflect desired behavior, consider modifying the conditional so that only arrays with a `length` greater than `1` are sorted.

[N14] Unused argument in `setParameters` function from `CashMarket.sol`

In `CashMarket.sol` the `setParameters` function accepts an argument `precision`. However, that argument is checked to be a constant and never

used otherwise. Consider removing it.

[N15] Unused import statements

Within the codebase there are instances of files being imported unnecessarily. *Some examples* are:

- The `CashMarket.sol` import on line 10 of `ERC1155Token.sol`.
- The `IERC1155TokenReceiver.sol` import on line 8 of `ERC1155Trade.sol`.
- The `IAggregator.sol` import on line 17 of `Escrow.sol`.
- The `Governed.sol` import on line 8 of `RiskFramework.sol`.

Consider removing any unused import statements to improve overall code legibility.

[N16] Function visibilities too permissive

There are some functions that are not being accessed locally but are being declared as `public` instead of `external`. *Some examples* are:

- In `Escrow.sol`: `listCurrency` `depositsOnBehalf`, and `withdrawsOnBehalf`
- In `Portfolios.sol`: `freeCollateral`, `freeCollateralAggregateOnly`, and `freeCollateralFactors`
- In `ERC1155Trade.sol`: `batchOperation`, and `batchOperationWithdraw`

Moreover, *some examples* of functions that are only being accessed locally but are being declared as `internal` instead of `private` are:

- In `Escrow.sol`: `_depositEth`, `_withdrawEth`, `_deposit`, `_tokenDeposit`, `_withdraw`, `_tokenWithdraw`, `_settleCashBalance`, `_liquidate`, `_validateCurrencies`, `_finishLiquidateSettle`, `_freeCollateral`, `_freeCollateralFactors`, `_hasCollateral`, and `hasNoAssets`.
- In `Portfolios.sol`: `_freeCollateral`, `_settleMaturedAssets`, `_tradefCashLiquidator`, `_calculateNotionalToTransfer`, `_tradePortfolio`, `tradeLiquidityToken`, `_tradeCashReceiver`, `_upsertAsset`, and `_reduceAsset`
- In `CashMarket.sol`: `_addLiquidity`, `_removeLiquidity`, `_settleLiquidityToken`, `_takeCurrentCash`, `_takefCash`, `_calculateTransactionFee`, `_updateMarket`, `_isValidBlock`, and `_tradeCalculation`

- In `ERC1155Trade.sol`: `_batchTrade`, `_updateWithdrawsWithTradeRecord`, and `_calculateWithdrawAmount`

Consider limiting function visibility where possible to improve the overall clarity and readability of the code.

Conclusions

No critical or high severity issues were found. Some changes were proposed to follow best practices and reduce the potential attack surface. Some of the high-level issues regarding efficiency, usability, architecture, and readability of the code were greatly improved from the prior audit. The protocol is now in a more mature state, but there is still room for improvement.

Security Audits

- If you are interested in smart contract security, you can continue the discussion in our forum, or even better, join the team 🚀
- If you are building a project of your own and would like to request a security audit, please do so here.

RELATED POSTS

SECURITY AUDITS

Compound Audit

The Compound team asked us to review and audit their platform's smart contracts.

SECURITY AUDITS

Aave Protocol Audit

The Aave team asked us to review and audit a pre-production version of their

We examined their...

READ MORE

protocol. We looked at...

READ MORE

SECURITY AUDITS

Balancer Contracts Audit

Balancer is an automated portfolio manager. It allows anyone to create Balancer pools, each of...

READ MORE